ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Технологія проектування програмних систем

Електронний конспект лекцій для студентів спеціальностей 7.05010202 (спеціалісти) і 8.05010202 (магістранти) - “Системне програмування”

Затверджено на засіданні
кафедри системного програмування,
протокол №9 від 14 березня 2014 р.

Хмельницький 2014

Технологія проектування програмних систем: Електронний конспект лекцій для студентів спеціальностей 7.05010202 (спеціалісти) і 8.05010202 (магістранти) - “Системне програмування” денної форми навчання. Укл. О.В.Поморова, Т.О.Говорущенко – Хмельницький: ХНУ, 2014 – Укр. мовою, 384 с.

Укладачі: О.В.Поморова, доктор технічних наук, професор
Т.О.Говорущенко, кандидат технічних наук, доцент

Відповідальна за випуск: О.В.Поморова, д.т.н., професор, завідувач кафедри системного програмування

ВСТУП

Розроблення програмного забезпечення (англ. software engineering, software development) – це рід діяльності (професія) та процес, спрямований на створення та підтримку робото здатності, якості та надійності програмного забезпечення (ПЗ) з використанням технологій, методології та практики з керування проектами, математики, інженерії та інших галузей знань.

Виробництво ПЗ сьогодні — найбільша галузь світової економіки, в якій зайнято близько 3 млн фахівців (програмістів, розробників ПЗ та ін.). Ще кілька мільйонів осіб безпосередньо залежать від успішної діяльності виробників ПЗ.

У процесі проектування сучасного програмного забезпечення виникає ряд проблем: велика вартість та тривалі терміни проектування, складність вивчення та використання програмної системи, тривале навчання користувача, низька швидкість роботи користувача, значна кількість помилок у роботі користувача.

Сучасне ПЗ характеризується: складністю опису; наявністю сукупності тісно взаємодіючих компонентів з локальними завданнями та цілями; відсутністю повних аналогів, а відтак і відсутністю можливості використання будь-яких типових проектних рішень; необхідністю інтеграції наявного і розроблюваного ПЗ; функціонуванням у неоднорідному середовищі на різних апаратних платформах; відокремленістю та різнорідністю окремих груп розробників із різним рівнем кваліфікації; великими термінами виконання проекту.

Факти, що визначають специфіку ПЗ, як продукту: замовник не розуміє складності процесу розробки ПЗ і впливу зміни його вимог до ПЗ в процесі розробки; зміна та поява нових вимог в процесі розробки ПЗ є неминучими; ітеративність процесу розробки ПЗ, яка ускладнює його; рівень новизни та складності ПЗ є дуже високим; технології швидко змінюються, оновлюються та застарівають.

Глобально процес розроблення ПЗ може бути розділений на декілька процесів - формулювання вимог, формулювання цілей

проекту, аналіз прикладної галузі, створення функційної специфікації, проектування, реалізація. Проектування ПЗ наразі ведеться засобами автоматизованого розроблення ПЗ (CASE-засобами). Всі процеси розроблення ПЗ повинні виконуватись з відпрацюванням ряду стандартів (правил, угод), яких повинні дотримуватись всі учасники проекту.

В основу процесу розроблення ПЗ покладено фундаментальну ідею: проектування ПЗ є формальним процесом, який можна вивчати і вдосконалювати. Завдяки засвоєнню і правильному застосуванню методів і засобів створення ПЗ можна підвищити його якість, забезпечити керованість його процесу проектування і збільшити термін його життя. Але при тому, що складність деяких програмних розробок на сьогодні вже перевищує складність багатьох машинобудівних або інфраструктурних проектів, а наслідки помилок є катастрофічними, процес розроблення ПЗ залишається не забезпеченим фундаментальною теорією та методологією надбання знань і навичок. Всі дослідження у галузі програмної інженерії мають хаотичний, несистематизований характер. Безумовно, є ряд фундаментальних досліджень (роботи Боема, Дейкстри, Мейєра), але відсутня завершена, протестована та апробована теорія та методологія розроблення якісного ПЗ.

На сьогодні, для підготовки спеціалістів та магістрантів спеціальності “Системне програмування” у вищих навчальних закладах України введено дисципліну “Технологія проектування програмних систем”, яка формує понятійний аппарат проектування програмного забезпечення, а також систематизує принципи, методи і засоби проектування програмних систем.

Навчальна дисципліна "Технологія проектування програмних систем" має загальне навантаження 108 годин (3 кредити ECTS) для магістрантів та 198 годин (5,5 кредитів ECTS) для спеціалістів, в тому числі 34 години - лекції. Навчальна дисицпліна завершується екзаменом.

Даний конспект лекцій містить теоретичний матеріал для формування та систематизції понятійного апарату з навчальної дисицпліни "Технологія проектування програмних систем".

Лекція №1
ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА ВИБІР ЖИТТЄВОГО ЦИКЛУ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Огляд та порівняльний аналіз моделей життєвого циклу програмного забезпечення
2. Вибір прийнятної моделі життєвого циклу розроблення програмного забезпечення

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.
2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.
4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002
5. ISO/IEC 12207:2008. Systems and software engineering — Software life cycle processes
6. IEEE Standard 1074-1995 - IEEE Standard for Developing Software Life Cycle Processes
7. Why the Waterfall Model Doesn’t Work // [Електронний ресурс] - Режим доступу: http://www.infoq.com/resource/articles/scaling-software-agility/en/resources/ch02.pdf
8. Boehm B. A Spiral Model of Software Development and Enhancement - ACM SIGSOFT Software Engineering Notes, ACM, 11(4):14-24, August 1986 // [Електронний ресурс] - Режим доступу: http://weblog.erenkrantz.com/~jerenk/phase-ii/Boe88.pdf

9. Guide to the Software Engineering Body of Knowledge (SWEBOK) - A project of the IEEE Computer Society Professional Practices Committee, 2004 // [Електронний ресурс] - Режим доступу: http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/otros-recursos-1/SWEBOK_Guide_2004.pdf

10. IEEE Standard 1517-99. IEEE Standard for Information Technology – Software Lifecycle Process – Reuse Processes

Вступ

Одним з ключових понять в галузі розроблення програмного забезпечення є життєвий цикл. *Життєвий цикл розроблення ПЗ* - це проектна діяльність по розробленню та розгортанню програмного забзпечення. Життєвий цикл програмного забезпечення супроводжується розробленням, обігом та використанням програмної документації.

Життєвий цикл програмного забезпечення (ЖЦ ПЗ) - це сукупність окремих етапів робіт, що проводяться у заданому порядку протягом періоду часу, який починається з вирішення питання про розроблення програмного забезпечення і закінчується припиненням використання програмного забезпечення [ISO/IEC 12207:2008].

В загальному випадку, життєвий цикл визначається моделлю й описується у формі методології (методу). Модель або парадигма життєвого циклу визначає загальну організацію і, як правило, основні його фази та принципи переходу між ними. Методологія (метод) визначає комплекс робіт, їх детальний зміст і рольову відповідальність фахівців на всіх етапах моделі.

Модель життєвого циклу ПЗ – це структура, що складається із процесів, робіт та задач, які включають в себе розроблення, експлуатацію і супровід програмного продукту; охоплює життя системи від визначення вимог до неї до припинення її використання.

Модель ЖЦ ПЗ залежить від специфіки, масштабу і складності проекту та особливостей умов, за яких система створюється та функціонує.

Модель ЖЦ конкретного ПЗ визначає характер процесу створення цього ПЗ, що означає сукупність упорядкованих у часі, об'єднаних у стадії (етапи) робіт. Етап створення ПЗ - це частина процесу створення ПЗ, що обмежена певними часовими рамками і завершується випуском конкретного продукту (моделей ПЗ, програмних компонентів, документації).

Враховуючи завдання, що покладаються на модель ЖЦ, необхідно зробити правильний вибір процесів, їхніх завдань та дій для побудови моделі ЖЦ ПЗ, яка задовольнятиме концептуальну ідею проектованого ПЗ з урахуванням його складності та масштабу робіт. У модель ЖЦ обов'язково включаються процеси реалізації робіт і завдань, що забезпечують створення проміжного продукту і перехід до наступного процесу моделі. При виборі схеми моделі ЖЦ для конкретної предметної області, вирішуються питання включення важливих для створюваного продукту видів робіт або не включення несуттєвих робіт.

1. Огляд моделей життєвого циклу ПЗ

Найбільшого поширення набули наступні моделі ЖЦ ПЗ: каскадна (водоспадна), інкрементна, спіральна, еволюційна моделі та модель стандартизованого ЖЦ ПЗ.

Однією з перших стала застосовуватися *каскадна модель ЖЦ ПЗ* (waterfall model), в якій кожна робота виконується один раз і в тому порядку, як це представлено у моделі (рис.1.1). Каскадна модель була запропонована у 1970 році У.Ройсом. Основна властивість каскадної моделі полягає у завершенні кожного етапу до початку наступного, тобто у моделі робиться припущення, що кожна робота буде виконана настільки ретельно і вірно, що після її завершення і переходу до наступного етапу повернення до попереднього етапу не буде потрібне. Розробник перевіряє проміжний результат різними відомими методами верифікації та фіксує його в якості готового еталону для наступного процесу.

Відповідно до цієї моделі ЖЦ, роботи та завдання процесу розроблення зазвичай виконуються послідовно, як це представлено у схемі. Однак допоміжні та організаційні процеси (контроль вимог, управління якістю) зазвичай виконуються паралельно з

процесом розроблення. У даній моделі повернення до початкового процесу передбачається після супроводу та виправлення помилок.

Рис.1.1 - Каскадна модель ЖЦ програмного забезпечення

Переваги реалізації ПЗ за допомогою каскадної моделі наступні:

- всі завдання підсистем та ПЗ реалізуються одночасно, що сприяє встановленню стабільних зв'язків між ними;

- повністю розроблене ПЗ з документацією на нього легше супроводжувати, тестувати, фіксувати помилки і вносити зміни не безладно, а цілеспрямовано, починаючи з вимог і повторити процес.

Недоліки та фактори ризику каскадної моделі ЖЦ ПЗ відображені на рис.1.2.

Каскадну модель можна розглядати як модель ЖЦ, придатну для створення першої версії ПЗ з метою перевірки реалізованих в ній функцій. При супроводі та експлуатації можуть бути виявлені різного роду помилки, виправлення яких потребують повторного виконання всіх процесів, починаючи з уточнення вимог.

Рис.1.2 - Недоліки та фактори ризику каскадної моделі

В основу *інкрементної моделі ЖЦ ПЗ* (рис.1.3) покладено наступну концепцію: перша створювана проміжна версія ПЗ (випуск 1) реалізує частину вимог, в наступну версію (випуск 2) додають додаткові вимоги і так доти, доки не будуть остаточно виконані всі вимоги та вирішені завдання розроблення ПЗ. Для кожної проміжної версії на етапах ЖЦ виконуються необхідні процеси, роботи та завдання, в тому числі аналіз вимог та створення нової архітектури, які можуть бути виконані одночасно.

Фактори ризику, які можуть мати місце при застосуванні інкрементної моделі ЖЦ ПЗ наведені на рис.1.4.

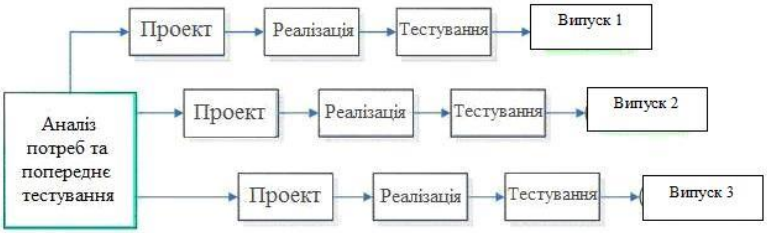


Рис.1.3 - Інкрементна модель ЖЦ програмного забезпечення

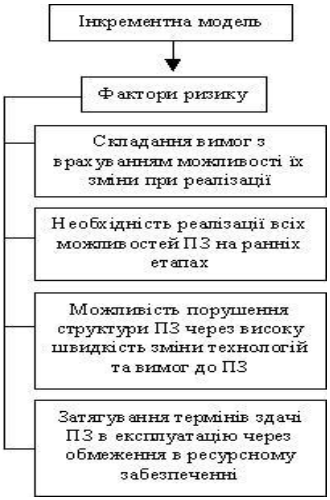


Рис.1.4 - Фактори ризику інкрементної моделі

Наразі інкрементна модель частіше розглядається в контексті поступового нарощування функціональності створюваного продукту.

Інкрементну модель ЖЦ доцільно використовувати у випадках, коли:

- бажано реалізувати деякі можливості ПЗ швидко за рахунок створення проміжної версії продукту;

- система розбивається на окремі складові частини, які можна реалізовувати як деякі самостійні проміжні або готові продукти;

- можливе збільшення фінансування на розроблення окремих частин ПЗ.

Виходячи з можливості внесення змін, як до процесу, так і в створюваний проміжний продукт була створена *спіральна модель ЖЦ ПЗ* (spiral model) - рис.1.5. Спіральна модель розроблена Б.Боемом у середині 80-х років XX століття. Внесення змін у спіральній моделі орієнтоване на задоволення потреби користувачів одразу, як тільки буде встановлено, що створені артефакти або елементи документації (опис вимог проекту, коментарі різного виду і т.і.) не відповідають дійсному стану розроблення.

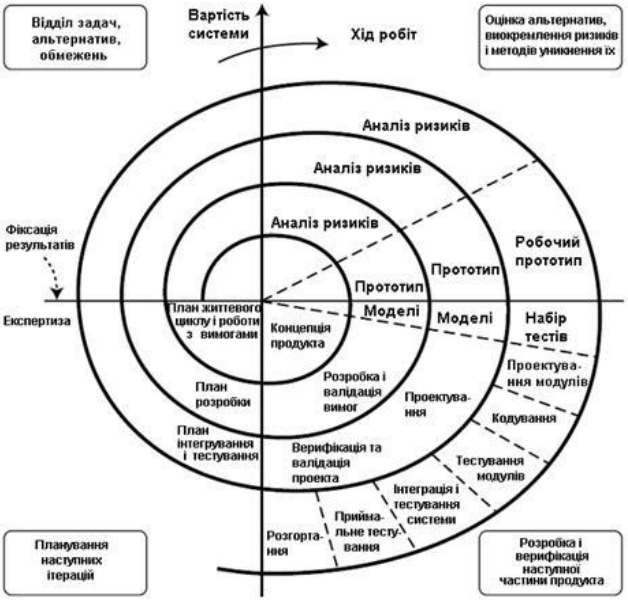


Рис.1.5 - Спіральна модель ЖЦ програмного забезпечення

Дана модель ЖЦ припускає аналіз продукту на витку розроблення, його перевірку, оцінювання правильності та прийняття рішення переходити на наступний виток або спуститися на попередній виток для доопрацювання на ньому проміжного продукту.

Відмінність спіральної моделі від каскадної полягає у можливості забезпечувати багаторазове повернення до процесу формулювання вимог і до повторного розроблення будь-якого процесу виконання робіт. На зображеній моделі (рис.1.5), кожен виток спіралі відповідає одній з версій розроблення ПЗ.

Недоліком спіральної моделі є важкість визначення моменту переходу розроблення на наступний етап.

Серед переваг спіральної моделі слід відмітити спрощення внесення змін в проект при зміні вимог замовника, оскільки елементи ПЗ інтегруються поступово. Спіральна модель життєвого циклу дає можливість визначати якість вже на етапі проектування.

У випадку *еволюційної моделі ЖЦ* програмне забезпечення розробляється у вигляді послідовності блоків структур (конструкцій). На відміну від інкрементної моделі ЖЦ, вимоги встановлюються частково і уточнюються в кожному наступному проміжному блоці структури ПЗ. Використання еволюційної моделі припускає проведення дослідження предметної області для вивчення потреб замовника проекту та аналізу можливості застосування цієї моделі для реалізації. Модель застосовується для розробки нескладних і не критичних систем, для яких головною вимогою є реалізація функцій системи. При цьому вимоги не можуть бути визначені одразу і повністю.

Розвитком цієї моделі є *модель еволюційного прототипування* (рис.1.6). В літературі вона часто називається моделлю швидкого розроблення додатків *RAD* (Rapid Application Development). У даній моделі наведені дії, пов'язані з аналізом її застосування для конкретного виду ПЗ, а також спостереження про замовника для визначення потреб користувача з метою розроблення плану створення прототипу. У моделі є дві головні ітерації розроблення функціонального прототипу, проектування та реалізації ПЗ. Перевіряється, чи задовольняє ПЗ всім функціональним і не функціональним вимогам. Основною ідеєю цієї моделі є моделювання окремих функцій ПЗ в прототипі і поступове еволюційне його доопрацювання для виконання всіх заданих функціональних вимог.

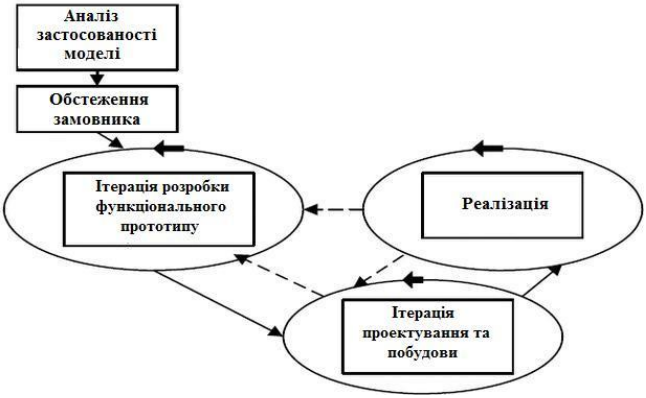


Рис.1.6 - Модель еволюційного прототипування

При використанні еволюційної моделі слід враховувати фактори ризику, наведені на рис.1.7.

Рис.1.7 - Фактори ризику еволюційної моделі

Переваги застосування даної моделі ЖЦ наступні:

- швидка реалізація деяких функціональних можливостей ПЗ та перевірка їх роботоздатності;
- використання проміжного продукту в наступному прототипі;
- виділення окремих функціональних частин для реалізації їх у вигляді прототипу;

- зворотній зв'язок із замовником для уточнення функціональних вимог;

- спрощення внесення змін в зв'язку з заміною окремої функції;

- дозволяє знизити ступінь невизначеності із завершенням кожної ітерації циклу;

- тестування продукту можна починати вже на ранніх стадіях життєвого циклу.

Отже, ЖЦ є моделлю створення і використання ПЗ, яка відображає його різні стани, починаючи з моменту виникнення необхідності в даному програмному продукті і закінчуючи моментом його повного виходу із вжитку у всіх користувачів. Для сучасного ПЗ найбільш прийнятною та універсальною є спіральна модель ЖЦ ПЗ, а для нескладного і некритичного ПЗ, в якому найбільш важливими є функціональні можливості, може бути застосована еволюційна модель ЖЦ ПЗ. Здавалось би, програмна індустрія нарешті прийшла до загальної "правильної" моделі. Але каскадна модель, нещадно розкритикована теорією і практикою, подовжує зустрічатись в реальному житті. Спіральна модель є яскравим представником еволюційного погляду, але в той же час представляє собою єдину модель, яка приділяє увагу аналізу та попередженню ризиків.

2. Вибір прийнятної моделі життєвого циклу ПЗ

Вибір прийнятної моделі життєвого циклу розроблення ПЗ для проекту може здійснюватись за наступним алгоритмом:

1) Аналіз таких відмінних категорій проекту (таблиці 1-4), як: вимоги, колектив розробників, колектив користувачів, тип проекту та ризики;

2) Відповіді на питання кожної категорії;

3) Розташування категорій та питань за ступенем важливості відносно проекту, для якого обирається модель ЖЦ.

Категорія *вимог* (таблиця 1.1) складається з питань відносно вимог, які висуває користувач до проекту, тобто відносно властивостей програмної системи, яка підтримується даним проектом.

Таблиця 1.1 – Вибір моделі ЖЦ на основі вимог

Питання	**Каскадна**	**Спіральна**	**RAD**	**Інкрементна**
Чи є вимоги добре відомими або такими, що легко визначаються?	Так	Ні	Так	Ні
Чи можуть вимоги визначатись у циклі?	Так	Ні	Так	Так
Чи часто змінюватимуться вимоги в циклі?	Ні	Так	Ні	Ні
Чи потрібно демонструвати вимоги?	Ні	Так	Так	Ні
Чи потрібна перевірка концепції для демонстрації можливостей?	Ні	Так	Так	Ні
Чи будуть вимоги відображати складність програмної системи?	Ні	Так	Ні	Так
Чи володіє вимога функціональними властивостями на ранньому етапі?	Ні	Так	Так	Так

При можливості у склад *колективу розробників* краще відібрати персонал ще до того, як буде обиратись модель ЖЦ. Характеристики такого колективу (таблиця 1.2) відіграють важливу роль у процесі вибору, оскільки саме цей колектив несе відповідальність за вдале виконання циклу.

Таблиця 1.2 – Вибір моделі ЖЦ на основі характеристик учасників колективу розробників

Питання	**Каскадна**	**Спіральна**	**RAD**	**Інкрементна**
Чи є проблеми предметної галузі проекту новими для більшості розробників?	Ні	Так	Ні	Ні
Чи є технологія предметної галузі проекту	Так	Так	Ні	Так

новою для більшості розробників?				
Чи є інструменти проекту новими для більшості розробників?	Так	Так	Ні	Ні
Чи змінюються ролі учасників проекту?	Ні	Так	Ні	Так
Чи можуть розробники проекту навчатись?	Ні	Ні	Так	Так
Чи є структура більш значущою для розробників, ніж гнучкість?	Так	Ні	Ні	Так
Чи буде менеджер проекту суворо відстежувати прогрес команди?	Так	Так	Ні	Так
Чи важлива легкість розподілу ресурсів?	Так	Ні	Так	Так
Чи сприймає колектив рівноправні огляди?	Так	Так	Ні	Так

На початкових фазах проекту можна одержати чітке уявлення про *колектив користувачів* (таблиця 1.3) та його майбутній взаємозв'язок з колективом розробників протягом всього проекту. Таке уявлення допоможе при виборі прийнятної моделі ЖЦ, оскільки деякі моделі вимагають посиленої участі користувачів у процесі розроблення та вивчення проекту.

Таблиця 1.3 – Вибір моделі ЖЦ на основі характеристик колективу користувачів

Питання	**Каскадна**	**Спіральна**	**RAD**	**Інкрементна**
Чи є присутність користувачів обмежена?	Так	Так	Ні	Так
Чи будуть користувачі знайомі з визначенням програмної системи?	Ні	Так	Ні	Так
Чи будуть користувачі ознайомлені з проблемами предметної галузі?	Ні	Ні	Так	Так
Чи користувачі задіяні на всіх етапах ЖЦ?	Ні	Ні	Так	Ні
Чи відстежуватиме замовник хід проекту?	Ні	Так	Ні	Ні

Уточнимо тепер, що собою являють *тип проекту та ризики* (таблиця 1.4), які є елементами, визначення яких виконується на

етапі планування. В деяких моделях передбачений менеджмент ризиків високого степеня, в той час як в інших він не передбачений взагалі. Вибір моделі, яка робить можливим менеджмент ризиків, не означає, що не потрібно складати план дій, спрямований на мінімізацію виявлених ризиків. Така модель просто забезпечує схему, за якої можно обговорити та виконати даний план дій.

Таблиця 1.4 – Вибір моделі ЖЦ на основі характеристик типу проекту та ризиків

Питання	**Каскадна**	**Спіральна**	**RAD**	**Інкрементна**
Чи буде проект ідентифікувати новий напрямок продукту для організації?	Ні	Так	Ні	Так
Чи буде проект мати системну інтеграцію?	Ні	Так	Так	Так
Чи є проект розширенням існуючого ПЗ?	Ні	Ні	Так	Так
Чи буде фінансування проекту стабільним?	Так	Ні	Так	Ні
Чи очікується тривала експлуатація програмної системи в організації?	Так	Так	Ні	Так
Чи потрібен високий ступінь надійності?	Ні	Так	Ні	Так
Чи буде ПЗ змінюватись непередбаченими методами на етапі супроводу?	Ні	Так	Ні	Так
Чи є графік обмеженим?	Ні	Так	Так	Так
Чи є «прозорими» інтерфейсні модулі?	Так	Ні	Ні	Так
Є повторно використовувані компоненти?	Ні	Так	Так	Ні
Чи є достатніми ресурси?	Ні	Так	Ні	Ні

Проблема використання тієї чи іншої моделі життєвого циклу програмного забезпечення полягає у тому, що не всі етапи життєвого циклу відпрацьовані повноцінно через відставання або відсутність повноцінної методології для виконання певного етапу. Так, виконання етапів формулювання вимог та тестування базуються на практичних методиках. Тому вибір прийнятної моделі – це лише перша стадія застосування моделі життєвого циклу в

процесі певного проекту. Наступна стадія полягає у підлаштуванні обраної моделі ЖЦ відповідно до потреб даного проекту. Згідно моделі SEI CMM, не існує якихось чітких рекомендацій чи вказівок щодо підлаштування моделі ЖЦ. Життєві цикли та їхні етапи можна використовувати в якості відправної точки при визначенні тих циклів, етапів та дій, які потрібні в даний момент часу. Після завершення підлаштування модель має більшу значущість для колективу розробників та колективу користувачів.

А що ж відбуватиметься у випадку, коли при виконанні проекту вносяться певні зміни, які змушують колектив задуматись, що інша модель ЖЦ була би більш дієвою? Чи можна змінити модель в процесі виконання проекту? Відповідь на це питання майже завжди позитивна, але заміну моделі слід виконувати з обов'язковим врахуванням можливих наслідків такої зміни для проекту. Але в будь-якому випадку краще замінити модель, ніж намагатись використовувати ту, яка не відповідає в достатньому ступені потребам проекту.

Висновки

Існує чимало різних моделей або представлень життєвого циклу розроблення ПЗ. Всі вони представляють собою логічно побудовану послідовність дій, починаючи з визначення потреби і закінчуючи виробництвом ПЗ. Кожна модель представляє собою процес, який структурно складається з з етапів, спрямованих на забезпечення цілісності відповідних дій. Кожний повністю відпрацьований етап знижує ступінь ризику при виконанні проекту завдяки застосуванню критеріїв входу та виходу для визначення подальших дій. Життєві цикли розроблення ПЗ є методиками, які охоплюють всі стандарти і процедури, які впливають на планування, збір і аналіз вимог, розроблення проекту, конструювання та впровадження програмної системи. З метою забезпечення ефективності життєвого циклу його слід ретельно обрати та підлаштувати відповідно до задач і цілей певного проекту. Популярні узагальнені моделі надають лише початкові схеми. Кожна така схема має недоліки та переваги, які й визначають можливість її застосування для певних типів проектів.

Лекція №2
СУЧАСНИЙ СТАН СФЕРИ ВИРОБНИЦТВА ПРОГРАМНИХ ЗАСОБІВ. РОЗПОВСЮДЖЕНІ ПРОЦЕСИ ТА ЕТАПИ РОЗРОБЛЕННЯ ПРОГРАМНИХ СИСТЕМ

План:

1. Сучасний стан сфери виробництва програмних засобів
2. Розповсюджені процеси та етапи розроблення програмних систем

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.
2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.
4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002
5. Мацяшек, Лешек. А. Анализ и проектирование систем. Разработка инфорамционных систем с использованием UML. –М.: Издательский дом “Вильямс”.
6. Вендров, А.М. Проектирование программного обеспечения экономических информационных систем / А.М. Вендров. - М.: Финансы и статистика, 2002.
7. С.Макконнелл. Совершенный код. Мастер-класс - М.: Издательство "Русская редакция", 2013 - 896 с.
8. Мищенко В.О., Поморова О.В., Говорущенко Т.А. CASE-оценка критических программных систем. В 3-х томах. Том 1. Качество / Под ред. Харченко В.С. - Харьков: Нац.аэрокосмический университет "ХАИ", 2012. - 201 с.
9. ISO/IEC 12207:2008. Systems and software engineering — Software life cycle processes

10. IEEE Standard 1074-1995 - IEEE Standard for Developing Software Life Cycle Processes

11. Guide to the Software Engineering Body of Knowledge (SWEBOK) - A project of the IEEE Computer Society Professional Practices Committee, 2004 // [Електронний ресурс] - Режим доступу: http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/otros-recursos-1/SWEBOK_Guide_2004.pdf

12. IEEE Standard 610.12-90. Standard Glossary of Software Engineering Terminology // [Електронний ресурс] - Режим доступу: http://web.ecs.baylor.edu/faculty/grabow/Fall2011/csi3374/secure/Standards/IEEE610.12.pdf

13. IEEE Standard 1219-98. IEEE Standard for Software Maintenance // [Електронний ресурс] - Режим доступу: http://www.cs.uah.edu/~rcoleman/CS499/CourseTopics/IEEE_Std_1219-1998.pdf

14. IEEE Standard 14764-2006 (ISO/IEC 14764). Standard for Software Engineering - Software Maintenance // [Електронний ресурс] - Режим доступу: http://webstore.iec.ch/preview/info_isoiec14764%7Bed2.0%7Den.pdf

15. ISO 9001: 2008-12. Quality management systems – Requirements // [Електронний ресурс] - Режим доступу: http://nads.gov.ua/sub/data/upload/publication/cherkaska/ua/6331/iso-9001_2008_dvuyazychnyj.pdf?s398224032=63e9aac82dc234cb077145d99b22b9aa

Вступ

Виробництво ПЗ сьогодні — найбільша галузь світової економіки, в якій зайнято близько 3 млн фахівців (програмістів, розробників ПЗ та ін.). Ще кілька мільйонів осіб безпосередньо залежать від успішної діяльності виробників ПЗ.

В кінці 90-х років минулого століття знання та досвід, накопичені в індустрії ПЗ за попередні 30-35 років, а також за більш ніж 15-річні спроби застосування різних моделей розроблення, були оформлені у дисципліну, яку було названо програмною інженерією. Формування дисципліни відбувалось і

відбувається без фундаментальної теорії та методології, а лише на основі практичного досвіду.

Саме тому, хоча програмне забезпечення розробляють вже понад п'ятдесяти років, за цей період задачі, які воно може вирішувати, рівень їх складності та форми представлення отриманих результатів кардинально змінилися, дотепер розроблення якісних програмних продуктів не стала нормою. Також потребують розвитку та вдосконалення методології розроблення надійного ПЗ з відповідними витратами та в межах заданого часу. Джерела несправностей сучасного ПЗ вкрай різноманітні, і це лише ускладнює проблему.

На думку Д.Паттерсона, світова гонитва за продуктивністю призвела до залежності людини від технологій, і людству пора створювати такі інформаційні технології, на які світ дійсно може покластися, повністю довіряючи їм.

Звісно, хаотичний період розвитку ПЗ, коли значно більше уваги приділялось саме програмному коду, а не його якості, став відходити у минуле. За останні роки програмна індустрія досягла такого рівня розвитку, при якому вимоги до забезпечення якості стали обов'язковим пунктом договорів на предмет розроблення програмних систем.

1. Сучасний стан сфери виробництва ПЗ

Криза у галузі розроблення ПЗ була помітною ще більше 50 років тому - великі проекти стали виконуватися з відставанням від графіка або з перевищенням кошторису витрат, розроблений продукт не мав необхідних функціональних можливостей, продуктивність його була низька, якість програмного забезпечення не влаштовувала споживачів. При наявності ряду методів та засобів, залученні кращих фахівців для розроблення технологій та стандартів проектування та розроблення програмних комплексів, якість ПЗ, як і раніше, залежить від знань та досвіду розробників.

Сучасне ПЗ характеризується:

1) складністю опису;

2) наявністю сукупності тісно взаємодіючих компонентів з локальними завданнями та цілями;

3) відсутністю повних аналогів, а відтак і відсутністю можливості використання будь-яких типових проектних рішень;

4) необхідністю інтеграції наявного і розроблюваного ПЗ;

5) функціонуванням у неоднорідному середовищі на різних апаратних платформах;

6) відокремленістю та різнорідністю окремих груп розробників із різним рівнем кваліфікації;

7) великими термінами виконання проекту.

Отже, наразі істотною і невід'ємною властивістю програмних систем є їх складність. Постійне зростання складності функцій ПЗ неминуче призводить до збільшення їх обсягу та трудомісткості створення.

Щільність помилок, поява яких очікується в ПЗ різного розміру, представлена у таблиці 2.1.

Таблиця 2.1 - Типова щільність помилок для ПЗ

Розмір ПЗ	**Типова щільність помилок**
Менше 2 К	0-25 помилок на 1000 рядків коду
2К-16К	0-40 помилок на 1000 рядків
16К-64К	0,5-50 помилок на 1000 рядків
64К-512К	2-70 помилок на 1000 рядків
Більше 512К	4-100 помилок на 1000 рядків

З таблиці 2.1 слідує, що сучасне ПЗ обсягом в мільйони рядків коду в принципі не може бути безпомилковим. Проблема полягає в тому, щоб забезпечити потрібну якість ПЗ з врахуванням того, що деяка невідома кількість помилок та дефектів у ПЗ завжди залишається, а їх негативна дія повинна бути блокована або скорочена до допустимого рівня.

За наближеними оцінками витрати на розроблення ПЗ складають близько 275 мільярдів доларів, але лише 72% програмних проектів досягають етапу впровадження і всього 26% програмних проектів завершуються успіхом.

Статистика успішності проектів по розробленню програмного забезпечення відображена на рис.2.1.

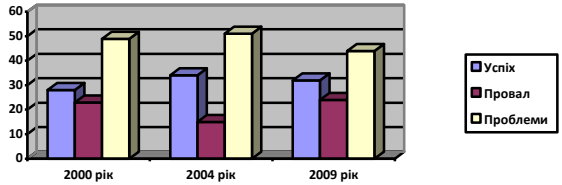


Рис.2.1 - Статистика успішності програмних проектів

За статистикою, 18% програмних проектів ніколи не завершуються; 53% проектів по розробленню ПЗ завершуються з перевитратами на 56% і перевищенням термінів на 84%; лише 29% проектів вкладаються в терміни та бюджет.

Програмні проекти часто зазнають невдач через помилки на ранніх етапах життєвого циклу ПЗ, а саме:

1) неадекватне формулювання вимог;
2) невдале проектування або неефективне планування;
3) невірне розуміння або недостатній аналіз специфікації;
4) нереалістичні проектні плани.

Проаналізуємо помилки вбудованого ПЗ, що призвели до відомих катастроф та інцидентів, які були внесені на ранніх етапах життєвого циклу. Такий аналіз наведено у таблиці 2.2.

Таблиця 2.2 - Аналіз помилок вбудованого ПЗ та їхніх наслідків

Подія	**Причина**	**Наслідки**
Етап формулювання вимог		
У 1971 р. "Космос-419" не стартував до Марсу	Невірність специфікації	Втрата апарату
У 1971 р. станція "Марс 2" не змогла відстикуватись від корабля	Невірність специфікації	Невиконання завдання
Аварія гелікоптера Chinook у 1994 році	Невірність специфікації	Загинуло 29 чоловік

"Смертельні" сеанси радіаційної терапії із застосуванням Therac-25	Неповнота специфікації	6 пацієнтів одержали смертельну дозу опромінення
Вибух ракети-носія Ariane 5 у1996 році	Протиріччя вимог щодо необхідності забезпечення надійності та максимально допустимого навантаження	Вартість обладнання та розроблення - 7,5 млрд. доларів, "упущена вигода" – 2 млрд. доларів
Етап проектування		
"Смертельні" сеанси радіаційної терапії із застосуванням Therac-25	Помилки у розробленні та постановці проекту; некоректні процедури оцінки та прогнозування ризиків	6 пацієнтів одержали смертельну дозу опромінення
Помилка процесора Intel Pentium у 1994 році	Відхилення від специфікації	Втрати компанії – 475 млн. доларів
Аварія станції Mars Climate Orbiter та зникнення зв'язку із Mars Polar Lander у 1999 році	Помилка в проекті через використання різних одиниць вимірювання	327,6 млн. доларів - апарати та 91,7 млн. доларів - запуски
Аварійне падіння ракетоносія "Рокот" у 2005 році	Логічна помилка в алгоритмі системи керування	Втрата європейського наукового супутнику CryoSat
Аварія аеробуса A330 у жовтні 2008 року	Помилка у системному алгоритмі обробки даних	110 пасажирів та 9 членів екіпажу були поранені
Падіння ракети-носія "Зенит-3SL" у 2013 році	Невірний проект системи керування	Втрата супутника "Intelsat-27"

Проаналізуємо тепер, які помилки прикладного ПЗ з катастрофічними наслідками були внесені на ранніх етапах життєвого циклу (таблиця 2.3).

Таблиця 2.3 - Аналіз помилок прикладного ПЗ та їхніх наслідків

Подія	**Причина**	**Наслідки**
Етап формулювання вимог		
Збій у системі Нью-Йоркського банку	Недостатність пам'яті через невірні вимоги	32 млрд. доларів
У 1990 р. на AT&T відбулась 9-годинна аварія	Проблеми з граничними умовами у специфікації	75 млн. нездійснених дзвінків, втрата 60 млн. доларів
У 1998 р. на AT&T відбулась 26-годинна аварія	Проблеми з прихованими граничними умовами у специфікації	Непрацездатність служб, пов’язаних з передачею даних
Помилка Y2K - помилка двоцифрового збереження року у даті (1999 р.)	Невірність або неповнота специфікації	Втрати - 500 млрд. доларів
Етап проектування		
У 1983 р. на станції "Серпухов-15" спрацювала система виявлення	Невірно спроектована система розпізнавання	Світ був на межі ядерної війни
У 1991 р. система протиракетної оборони Patriot не перехопила іракську ракету	Помилка заокруглення - некоректний розрахунок місцезнаходження ракети, що наближалась	Загинули 28 американських солдат та близько 100 чоловік одержали поранення

Помилка Y2K - помилка двоцифрового збереження року у даті	Невірний проект	Втрати 500 мільярдів доларів
"Смертельна" терапія у Національно-му онкологічному інституті у Панама-Сіті в 2001 році	Некоректне обчисення доз радіації у ПЗ компанії Multidata Systems International	28 пацієнтів зазнали надмірного опромінення, декілька хворих померли
Аварія на заводі по переробці урану у Західній Австралії в 2001 році	Логічна помилка у алгоритмі	Викид радіоактивної речовини

До помилок вбудованого ПЗ із катастрофічними наслідками також належить і відмова системи енергопостачання супутника "Коронас-Фотон" внаслідок помилкового проекту та неповноти тестування, яка призвела до втрати зв'язку із ним (2009 рік). Варто згадати також наступні помилки у прикладному ПЗ, які призвели до катастрофічних наслідків: 1) невірний розрахунок навантаження на ПЗ на етапі формулювання вимог призвів до падіння рейтингу та втрати 500 мільярдів доларів компанією Dow Jones Industrial Average на біржевих торгах (1987 р.); 2) невірно спроектована система розпізнавання Patriot призвела до помилкової ідентифікації британського бомбардувальника Tornado як ворожої ракети (2003 р.) - загинули 2 пілоти.

Аналіз таблиць 2.2 і 2.3 показує, що чимало помилок вбудованого і прикладного ПЗ, які спричинили катастрофи та інциденти, були внесені на ранніх етапах життєвого циклу.

Таблиця 2.4 відображає відсоткову кількість помилок, які виникають на етапах формулювання вимог, проектування та реалізації ПЗ різного обсягу.

З таблиці 2.4 видно, що помилки формулювання вимог та проектування складають 25-55% всіх помилок, причому чим більший обсяг ПЗ, тим більше помилок вноситься на ранніх етапах.

Таблиця 2.4 - Розподіл помилок, припущених на різних етапах життєвого циклу

Етап ЖЦ	**Обсяг ПЗ**				
	2К	8К	32К	128К	512К
Формулювання вимог	До 10%	До 15%	До 20%	До 22%	До 23%
Проектування	До 15%	До 19%	До 25%	До 28%	До 32%
Конструювання	До 75%	До 66%	До 55%	До 50%	До 45%

Слід врахувати також і факт, що вартість виправлення помилки проектування в 2-4 рази вища вартості виправлення помилки конструювання. Залежність вартості виправлення помилок від етапу ЖЦ ПЗ наведено на рис.2.2.

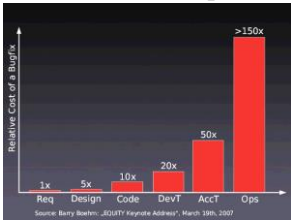


Рис.2.2 - Залежність вартості виправлення помилок від етапу життєвого циклу програмного забезпечення

Дослідження останніх 25 років переконливо довели високу вартість внесення змін, яких можна було уникнути. Так вчені з компаній Hewlett-Packard, IBM, Hughes, Aircraft, TRW виявили, що *виправлення помилки до початку конструювання (реалізації) в 10-100 разів дешевше, ніж її усунення в кінці роботи над проектом, під час тестування або після його випуску* - рис.2.3.

	Час виявлення дефекту				
Час внесення дефекту	Формулювання вимог	Проектування архітектури	Конструювання (кодування)	Тестування	Після випуску ПЗ
Формулювання вимог	1	3	5-10	10	10-100
Проектування архітектури	-	1	10	15	25-100
Конструювання (кодування)	-	-	1	10	10-25

Рис.2.3 - Середня вартість виправлення дефектів в залежності від часу їх внесення та виправлення (в умовних одиницях)

Із збільшенням інтервалу між моментами внесення та виявлення дефекту вартість його виправлення сильно зростає. Чим довше помилка зберігається у ланцюгу розроблення ПЗ, тим сильніше вона проникає в інші частини ПЗ, тим більше шкоди завдає на наступних етапах і тим більше коштів доведеться витратити на її усунення. Наприклад, *один дефект у вимогах може призвести до одного або декількох дефектів у проекті, які, в свою чергу, можуть призвести до появи множини дефектів у коді.*

Отже, забезпечення можливості раннього виявлення помилок та оцінювання якості проекту дали б можливість зменшити витрати на розроблення ПЗ, а то й уникнути ряду катастроф та інцидентів, причини яких були внесені на етапах формулювання вимог та проектування.

Статистика використання методологій проектування програмного забезпечення наведена на рис.2.4.

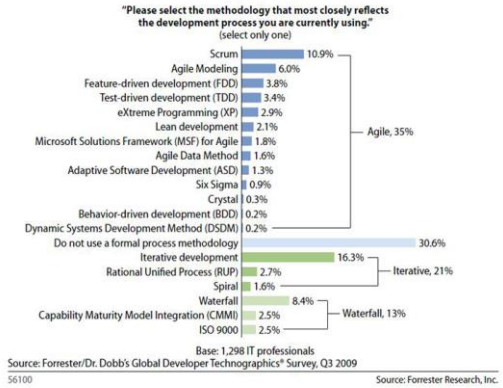


Рис.2.4 - Статистика використання методологій проектування ПЗ

Наразі в більшості програмних проектів, в більшості методологій проектування ПЗ основна частина зусиль з виявлення та виправлення помилок все ще виконується на етапі тестування, а то й після випуску ПЗ, отже, *на відлагодження та переробку йде близько 50% часу типового циклу розроблення ПЗ.* В десятках компаній було виявлено, що *політика раннього виправлення дефектів може в кілька разів знизити фінансові та часові*

витрати на розроблення ПЗ, що є вагомим аргументом на користь якомога більш раннього виявлення та усунення дефектів. Дослідження 50 проектів, на які витрачено понад 400 людинороків і які включили майже 3000000 рядків коду, проведені у Лабораторії проектування ПЗ NASA, показали, що підвищена увага до раннього контролю якості дозволяє істотно знизити рівень помилок, але не підвищує загальних витрат на розроблення.

В основу процесу розроблення ПЗ покладено фундаментальну ідею: проектування ПЗ є формальним процесом, який можна вивчати і вдосконалювати. Завдяки засвоєнню і правильному застосуванню методів і засобів створення ПЗ можна підвищити його якість, забезпечити керованість його процесу проектування і збільшити термін його життя. Але при тому, що складність деяких програмних розробок на сьогодні вже перевищує складність багатьох машинобудівних або інфраструктурних проектів, а наслідки помилок є катастрофічними, процес розроблення ПЗ залишається не забезпеченим фундаментальною теорією та методологією надбання знань і навичок. Безумовно, є ряд фундаментальних досліджень (роботи Боема, Дейкстри, Мейєра), але відсутня завершена, протестована та апробована теорія та методологія розроблення якісного ПЗ.

Софтверні компанії суб'єктивно обирають власний шлях до успіху програмних проектів:

1) Google - самокеровані невеликі команди, легкі орієнтовані на людей Agile-процеси;

2) NASA - високоформалізовані процеси, постійне вдосконалення процесів;

3) Microsoft - MSF, в деяких підрозділах – SEI PSP;

4) Boeing, Northrop-Grumman, Lockheed-Martin - SEI PSP/TSP, «зрілі» процеси CMM (5 level).

2. Розповсюджені процеси та етапи розроблення програмних систем

Стандарт [ISO/IEC 12207:2008] описує основні (Primary Processes), допоміжні (Supporting Processes) та організаційні (Organizational Processes) процеси ЖЦ ПЗ:

1) основні процеси:
- замовлення (Acqusition);
- постачання (Supply);
- розроблення (Development);
- експлуатація (Operation);
- супровід (Maintenance);

2) допоміжні процеси:
- документування (Documentation);
- керування конфігурацією (Configuration Management);
- верифікація (Verification);
- атестація (Validation);
- сумісний аналіз (Joint Review);
- вирішення проблем (Problem Resolution).

3) організаційні процеси:
- керування (Management);
- створення інфраструктури (Infrastructure);
- вдосконалення (Improvement);
- навчання (Training).

Дерево процесів життєвого циклу є структурою декомпозиції життєвого циклу на відповідні процеси (групи процесів). *Декомпозиція процесів будується на двох найважливіших принципах*, що визначають правила розбиття (partitioning) ЖЦ на складові процеси:

1) *принцип модульності* (задачі в процесі є функціонально зв'язаними; зв' язок між процесами – мінімальний; якщо функція використовується більше, ніж одним процесом, вона сама є процесом; якщо процес Y використовується процесом X і лише ним, то процес Y належить (є його частиною або його задачею) процесу X, за винятком випадків потенційного використання процесу Y іншими процесами в майбутньому);

2) *принцип відповідальності* (за кожний процес несе відповідальність особа, призначена для контролю заданого процесу життєвого циклу; функція, частини якої знаходяться в компетенції різних осіб, не може розглядатися як самостійний процес).

В загальному випадку, розбиття процесу базується на поширеному PDCA-циклі: P (Plan) - планування; D (Do) - виконання; C (Check) - перевірка; A (Act) - дія.

Основні процеси. Замовлення – визначає роботи і задачі замовника. Складається з наступних робіт:

- Inititation – ініціювання (підготовка);
- Request-for-proposal preparation – підготовка запиту (заявки);
- Contract preparation and update – підготовка та коригування договору;
- Supplier monitoring – моніторинг постачальника (нагляд);
- Acceptance and completion – прийом та завершення.

Постачання – визначає роботи та задачі постачальника. Включає наступні роботи:

- Inititation – ініціювання (підготовка);
- Preparation of response – підготовка пропозиції;
- Contract – розробка контракту (договору);
- Planning – планування;
- Execution and control – виконання та контроль;
- Review and evaluation – перевірка та оцінювання;
- Delivery and completion – постачання та завершення

Розроблення – визначає роботи та задачі розробника. Складається з наступних робіт:

- Process implementation – визначення процесу;
- System requirements analysis – аналіз системних вимог;
- System design – проектування системи (системної архітектури);
- Software requirements analysis – аналіз вимог до ПЗ;
- Software architectural design – проектування архітектури ПЗ;
- Software detailed design – детальне проектування програмної системи;
- Software coding and testing – кодування та тестування;
- Software integration – інтеграція програмної системи;
- Software qualification testing – кваліфікаційні випробування програмних засобів;

- System integration – інтеграція системи в цілому;
- System qualification testing – кваліфікаційні випробування системи;
- Software installation – встановлення (введення в дію);
- Software acceptance support – забезпечення приймання програмних засобів.

Експлуатація – визначає роботи і задачі оператора служби підтримки. Включає наступні роботи:

- Process implementation – визначення процесу;
- Operational testing – операційне тестування (експлуатаційні випробування);
- System operation – експлуатація системи;
- User support – підтримка користувача.

Супровід – визначає роботи і задачі, які виконуються фахівцями служби супроводу. Включає наступні роботи:

- Process implementation – визначення процесу;
- Problem and modification analysis – аналіз проблем та змін;
- Modification implementation – внесення змін;
- Maintenance review/acceptance – перевірка та приймання при супроводі;
- Migration – міграція (перенос);
- Software retirement – виведення ПЗ з експлуатації.

Стандарт ISO 12207 не визначає послідовність та розбиття виконання процесів у часі, адресуючи це питання також роботам з адаптації стандарту до конкретних умов та оточенню і застосуванню обраних моделей, практик, технік і т.і.

У 2004 році інститутом IEEE було сформовано Керівництво до зводу знань з програмної інженерії (Guide to the Software Engineering Body of Knowledge - SWEBOK), який став резльтатом консенсусу провідних представників програмної індустрії та визнаних авторитетів у галузі програмної інженерії SWEBOK описує *10 основних етапів розроблення програмних систем*:

1) вимоги до ПЗ (Software requirements);
2) проектування ПЗ (Software design);
3) конструювання ПЗ (Software construction);

4) тестування ПЗ (Software testing);

5) експлуатація (підтримка, супровід) ПЗ (Software maintenance);

6) конфігураційне керування ПЗ (Software configuration management);

7) управління у програмній інженерії (Software engineering management);

8) процеси програмної інженерії (Software engineering process);

9) інструменти та методи програмної інженерії (Software engineering tools and methods);

10) якість ПЗ (Software quality).

Програмні вимоги - властивості ПЗ, які повинні бути належним чином представлені у ньому для розв'язку конкретних практичних задач. Дана галузь знань стосується питань видобування (збирання), аналізу, специфікування та затвердження вимог. Питання, пов'язані з керуванням вимогами, здійснюють критично-важливий вплив на програмні проекти, а в певному ступені - на факт можливості успішного завершення проектів. Тільки систематична робота з вимогами дозволяє коректним чином забезпечити моделювання задач реального світу та формулювання необхідних приймальних тестів для впевненості у відповідності створюваного ПЗ критеріям, заданим реальними практичними потребами.

Якість процесу роботи із вимогами повинна ретельно оцінюватись та покращуватись в зв'язку із значущістю роботи з вимогами та з ключовою роллю цього процесу для визначення вартісних та часових ресурсів, необхідних для реалізації програмного проекту. Отже, після формулювання вимог слід визначати атрибути та метрики якості для: оцінювання покриття процесів роботи з вимогами з точки зору стандартів та моделей покращення процесів; вимірювання та кількісного оцінювання (benchmarking) процесів роботи з вимогами; планування та реалізації процесу покращення.

Етап визначення вимог тісно пов'язаний з процедурами верифікації (перевірки) та валідації (атестації, затвердження).

Вимоги до ПЗ вважаються описаними не повністю, якщо для них не задано правила верифікації і валідації (V&V). На жаль, часто у великих софтверних організаціях замість повноцінної перевірки суті та змісту документів, ці перевірки зводяться до так званого "нормоконтролю" - перевірка документів вимог полягає у перевірці на дотримання прийнятих стандартів зовнішнього оформлення документу, але аж ніяк не в оцінюванні якості вимог. Такий "нормоконтроль" не є повноцінною перевіркою вимог.

Для перевірки вимог існують наступні методи та практики:

1) огляд вимог (Requirements Review);

2) прототипування (Prototyping);

3) затвердження моделі (Model Validation);

4) приймальні випробування (Acceptance Tests).

З практичної точки зору, корисно мати певні методики та інструменти для кількісного оцінювання обсягу вимог для розроблюваного програмного продукту. Така кількісна оцінка корисна для досліджень масштабів змін у вимогах, оцінювання вартісних характеристик розроблення та підтримки ПЗ, зокрема оцінювання продуктивності розроблення та ефективності підтримки на етапах реалізації вимог та внесення змін.

Процес визначення архітектури, компонентів, інтерфейсів та інших характеристик ПЗ або його компонентів називається проектуванням. *Проектування* - це інженерна діяльність в рамках життєвого циклу, під час якої належним чином аналізуються вимоги для створення опису внутрішньої структури ПЗ, яка є основою для конструювання ПЗ. Результатом етапу проектування є архітектура ПЗ, тобто декомпозиція ПЗ у вигляді організованої структури компонентів та інтерфейсів між компонентами. Найважливішою характеристикою завершеності етапу проектування є той рівень деталізації компонентів, який дозволяє зайнятись їхнім конструюванням.

Потоки етапу проектування програмного забезпечення представлені на рис.2.5.

Рис.2.5 - Потоки етапу проектування

Конструювання ПЗ - детальне створення робочого ПЗ комбінуванням кодування, верифікації, модульного тестування, інтеграційного тестування та налагодження. Процес конструювання ПЗ пов'язаний із важливими аспектами діяльності по проектуванню і тестуванню - відштовхується від результатів проектування, а з його результатами працює тестування. У процесі конструювання створюється більша частина активів програмного проекту - конфігураційних елементів.

Більша частина результатів та процес конструювання можуть бути виміряні, в тому числі кількісно. Ці вимірювання, названі результатами аудиту коду або метриками коду, корисні як для оцінювання ризиків та якості, так і для вибору операцій по зниженню ризиків і підвищенню якості.

Основна відмінність етапу конструювання полягає у концентруванні уваги на програмному коді та інших артефактах, тісно пов'язаних з кодом, зокрема, на детальних моделях.

Тестування ПЗ - це діяльність, яка виконується для оцінювання та покращення якості ПЗ. Ця діяльність базується на виявленні дефектів та проблем у ПЗ. Тестування полягає у динамічній верифікації поведінки ПЗ на кінцевому наборі тестів, які обрані відповідним чином із звичайно виконуваних дій прикладної галузі та забезпечують перевірку відповідності очікуваній поведінці ПЗ. Тестування - це спостереження за виконанням програми із заданими параметрами, за заданим сценарієм або з іншимим заданими початковими умовами або цілями тестування. Ефективність тестування визначається лише в контексті заданих умов.

Методики тестування:

1) методики на основі інтуїції та досвіду інженера (Techniques based on the software engineer's intuition and experience);

2) методики на основі специфікації ПЗ (Specification-based techniques);

3) методики, орієнтовані на код (Code-based techniques);

4) методики, орієнтовані на дефекти (Fault-based techniques);

5) методики на основі умов використання (Usage-based techniques);

6) методики на основі природи додатку (Techniques based on the nature of the application).

Наразі тестування не розглядається як діяльність, яка починається лише після завершення фази конструювання, а доведено необхідність проводити тестування протягом всього процесу розроблення та супроводу. Планування тестування повинне починатись на ранніх стадіях роботи з вимогами. Очевидно, що легше попередити проблему, ніж боротися з її наслідками. Тестування, нарівні з керуванням ризиками, і є тим інструментом, який дозволяє ефективно діяти саме в такому ключі. Адекватна увага, приділена питанням тестування, якісно знижує ризик виникнення помилки на етапі експлуатації, забезпечуючи більш високе задоволення користувачів.

Етап супроводу ПЗ - сукупність діяльності, необхідної для забезпечення ефективної (з точки зору витрат) підтримки ПЗ.

Супровід ПЗ - це модифікація програмного продукту після передачі в експлуатацію для усунення збоїв, покращення показників продуктивності та/або інших характеристик або адаптації продукту для використання в модифікованому оточенні.

Супровід необхідний для забезпечення того, щоб ПЗ протягом всього періоду експлуатації задовольняло вимогам користувачів. Роботи по супроводу повинні проводитись для вирішення наступних задач:

1) усунення збоїв;
2) покращення дизайну;
3) реалізація розширень функціональних можливостей;
4) створення інтерфейсів взаємодії з іншими зовнішінми системами;
5) адаптація до інших апаратних та програмних платформ;
6) міграції успадкованого ПЗ;
7) виведення ПЗ з експлуатації.

Роботи по супроводу споживають значну частину фінансових ресурсів ЖЦ ПЗ. Загальне розуміння супроводу передбачає лише усунення збоїв. Але дослідження та опитування протягом багатьох років показують, що більше 80% зусиль по супроводу пов'язані не стільки з усуненням збоїв, скільки з іншими роботами, не пов'язаними з виправленням дефектів.

На жаль, питанням супроводу приділяється значно менша увага, ніж іншим етапам ЖЦ. Ігнорування оцінки вартості супроводу призводить до перевищення бюджетів, браку ресурсів і частого провалу використання таких продуктів. Неготовність розглядати ЖЦ в часі як триваючий і після передачі ПЗ в експлуатацію, непропрацьованість відповідних процедур коригування ПЗ після його випуску - основна проблема і для бізнес-середовищ, для яких ПЗ є лише інструментом, і для компаній-інтеграторів, які "забувають" про необхідність розвитку успіху після впровадження ПЗ, і для незалежних постачальників ПЗ, які, випускаючи одну версію ПЗ, починають працювати над новою версією, приділяючи недостатню увагу підтримці та оновленню вже існуючих версій.

Конфігураційне керування в галузі ПЗ - один з допоміжних процесів ЖЦ, який підтримує проектний менеджмент, діяльність з розроблення та супроводу, забезпечення якості, а також замовників та користувачів кінцевого продукту. Конфігураційне керування - це ідентифікація конфігурації ПЗ в певні моменти часу з метою систематичного контролю змін конфігурації, а також підтримки та супроводу цілісної та відстежуваної конфігурації протягом всього ЖЦ ПЗ.

На жаль, конфігураційне керування в багатьох проектних командах зводиться лише до контролю версій текстів програм і, в кращому випадку, документації на створюване ПЗ (не проектної документації!). Спроба обмежити конфігураційне керування лише питаннями контролю версій є результатом нерозуміння того, що результати процесу розроблення ПЗ - це не лише код, модулі та документація користувача, а й все те, що створювалось протягом всього процесу розроблення (описи бізнес-процесів, архітектурні моделі, вимоги, план проекту, задачі, запити на зміни і т.і.). Спрощення конфігураційного керування до рівня керування версіями вигідне багатьом постачальникам відповідних програмних засобів. В деяких випадках (малі проекти, тимчасово використовувані додатки) такий спрощений погляд цілком обгрунтований. Однак для іншого ПЗ таке спрощення абсолютно неприпустиме, оскільки конфігураційне керування передбачає постійно діючий процес, а не комплекс певних періодично виконуваних операцій. Тільки сприйняття діяльності з конфігураційного керування як інфраструктурної основи процесів ЖЦ може забезпечити ефективність керування програмними проектами. В той же час, конфігураційне керування - необіхдна, але не достатня умова, оскільки тільки сукупність процесів ЖУ визначає весь комплекс робіт по створенню ПЗ.

Управління у програмній інженерії - це додаток питань управління (планування, координації, кількісного оцінювання, моніторингу, контролю та звітності) до інженерної діяльності для систематичного, впорядкованого та кількісно вимірюваного забезпечення розроблення і супроводу ПЗ. Можливо управляти програмною інженерією так, як і будь-яким іншим комплексним

процесом, але в той же час існують аспекти, специфічні для програмних продуктів, а процеси ЖЦ ПЗ ускладнюють досягнення необхідного рівня ефективності управління. Серед ускладнюючих факторів: відсутність розуміння замовниками та споживачами складності, яка виникає внаслідок впливу змінюваних вимог та в силу природи програмної інженерії; неминуча зміна або поява нових клієнтських вимог, в результаті чого ПЗ змушене будуватись із застосуванням ітеративних процесів замість послідовного виконання; рівень новизни та складності ПЗ надзвичайно високий; застосовувані технології мають високу швидкість зміни, оновлення та застарівання.

Процеси програмної інженерії - це технічна та управлінська діяльність протягом процесів ЖЦ ПЗ, пов'язана із визначенням, реалізацією, оцінюванням, вимірюванням, керуванням, зміною та увдосконаленням процесів ЖЦ ПЗ. Основним джерелом знань щодо процесів програмної інженерії є стандарт ISO/IEC 12207:2008. Оцінювання процесів відбувається з використанням відповідних моделей та методів оцінювання якості ПЗ, які розглядатимуться пізніше.

Інструменти та методи програмної інженерії дозволяють автоматизувати певні повторювані дії, зменшуючи перевантаження інженерів рутинними операціями і допомагаючи їм сконцентруватись на творчих, нестандартних аспектах реалізації виконуваних процесів.

Інструменти часто проектуються з метою підтримки конкретних методів програмної інженерії і можуть варіюватись від підтримки окремих задач і до охоплення всього ЖЦ.

Методи представляють відповідні нотації, словник термінів, процедури виконання охоплюваних методом задач і рекомендації з оцінювання та перевірки виконуваного процесу і одержуваного в його результаті продукту, відтак накладають певні структурні обмеження на діяльність в рамках програмної інженерії з метою приведення цієї діяльності у відповідність до заданого систематичного підходу та досягнення успіху.

Методи програмної інженерії поділяються на: 1) евристичні (Heuristic Methods): структурні, орієнтовані на дані, об'єктно-

орієнтовані та орієнтовані на область застосування методи; 2) формальні (Formal Methods) - математичні методи, які включають ряд операцій, в тому числі створення формальної специфікації ПЗ, аналіз та доведення специфікацій, реалізація ПЗ на основі перетворення формальної специфікації у програми і верифікація ПЗ; 3) методи прототипування (Prototyping Methods): стилі прототипування, цілі прототипування, методики оцінювання/дослідження результатів протипування.

Історія програмної інженерії показала, що при наявності чималої кількості розроблених методів дисципліна не забезпечена фундаментальною методологією. Всі розроблені методи розрізнені, використовуються на розсуд розробників, оскільки відсутні чіткі критерії використання того чи іншого методу в кожному конкретному випадку. Програмна інженерія, як і будь-яка інженерна дисципліна, вимагає чітких та строгих математичних методів, але наразі використання математичних методів у програмній інженерії досить обмежене (математичні методи використовуються лише у внутрішній реалізації конкретних програмних продуктів, тобто неявно для розробників), хоча більше 35 років проводяться дослідження саме з використання математичних методів у процесі створення ПЗ. Така ситуація складається через небажання софтверних компаній використовувати математичні методи через їхню трудомітскість, а також через відсутність чіткої методології та критеріїв вибору методів.

Якість ПЗ - це ступінь відповідності присутніх характеристик вимогам. Якість ПЗ є постійним об'єктом турботи програмної інженерії. Програмні вимоги визначають потрібні характеристики якості ПЗ, а також впливають на методи кількісного оцінювання та сформульовані для оцінювання цих характеристик критерії приймання. Вартість якості може бути диференційована на вартість попередження дефектів, вартість оцінювання, вартість внутрішніх та зовнішніх збоїв. Якість ПЗ може підвищуватись за рахунок ітеративного процесу постійного покращення. Це вимагає контролю, координації та зворотнього зв'язку в процесі управління багатьма одночасно виконуваними

процесами (процесами ЖЦ, процесами виявлення, усунення та попередження збоїв/дефектів та процесів покращення якості). Детальніше поняття, основи та процеси якості ПЗ розглядатимуться пізніше.

На планування та вибір методів управління якістю впливають різні фактори: 1) область застосування ПЗ; 2) системні та програмні вимоги; 3) компоненти, використовувані у ПЗ; 4) використовувані стандарти програмної інженерії; 5) використовувані методи та інстрмунети програмної інженерії; 6) бюджет, персонал, плани та розклад діяльності з розроблення ПЗ; 7) цільові користувачі та призначення ПЗ; 8) рівень цілісності.

До програмної інженерії застосовують теорії та концепції, які лежать в основі вдосконалення якості - попередження та раннє діагностування помилок, постійне вдосконалення, увага до вимог замовника. Але власна теорія та методологія в галузі забезпечення якості ПЗ, яка б враховувала вплив різних факторів при управлінні якістю, наразі відсутня, а використовувані концепції є суб'єктивно залежними, оскільки вони засновані на роботах експертів.

Висновки

Процес розроблення ПЗ тісно пов’язаний з процесом аналізу та оцінювання значущих характеристик ПЗ. До характеристик ПЗ належать: вартість ПЗ, захист ПЗ, повнота реалізації вимог, обсяг файлів ПЗ, вимоги до системного програмного забезпечення та технічних засобів, обсяг потрібної оперативної та дискової пам'яті. Але найважливішими характеристиками ПЗ з точки зору розробника є його складність, а з точки зору користувача - його надійність та якість.

Лекція №3
СЕРТИФІКАЦІЯ Й ОЦІНЮВАННЯ ПРОЦЕСІВ СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Сертифікація процесів створення програмного забезпечення
2. Оцінювання процесів створення програмного забезпечення

Рекомендована література:

1. Липаев В.В.Сертификация программных средств. Учебник. – М.: СИНТЕГ, 2010. – 348 с.
2. Документирование программного обеспечения и эффективный процесс разработки // [Електронний ресурс] – Режим доступу: http://creograf.ru/?messPress_ShowR_161=1
3. Скляр В.В. Оценка качества и экспертиза программного обеспечения. Лекционный материал. - Харьков: НАУ "ХАИ", 2008. - 204 с.
4. Харченко В.С., Скляр В.В., Гордеев А.А. Верификация программного обеспечения. - Харьков: НАУ "ХАИ", 2006. - 132 с.
5. Крайер Э. Успешная сертификация на соответствие нормам ИСО серии 9000. Пер. с нем. – М.: ИЗДАТ, 1999
6. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

Вступ

Процес розробки програмного продукту подібний до інших інженерних задач. Його найбільш вдала адаптація для програмістів називається Уніфікований процес розробки програмного забезпечення (USDP- Unified Software Development Process). Інститутом IEEE було розроблено стандарти документації такого процесу розробки програмного забезпечення (ПЗ). Згідно цих

стандартів, документація проекту складається з наступних складових, розроблюваних в порядку перерахування:

- План експертизи програмного забезпечення (SVVP – Software Verification and Validation Plan): документ описує, яким чином і в якій послідовності повинні перевірятись стадії проекту і сам продукт на відповідність вимогам. Іншими словами, цей документ обумовлює те, як замовник і розробник переконаються в тому, що вони зробили те, що хотіли зробити, але не описує, що і як вони робитимуть.

- План контролю якості ПЗ (SQAP – Software Quality Assurance Plan) - показує, яким чином проект повинен досягти відповідності встановленому рівню якості;

- План управління конфігураціями ПЗ (SCMP – Software Configuration Management Plan) - описує, де і як розробник зберігатиме версії документації, програмного коду і як визначатиме, яким чином код пов'язаний з документацією;

- План управління програмним проектом (SPMP – Software Project Management Plan) - описує, як розробник керуватиме проектом створення програмного продукту, щоб довести розробку до вдалого фіналу;

- Специфікація вимог до програмного забезпечення (SRS – Software Requirements Specification) - це найважливіший документ. Складається з двох частин: перша включає те, що вимагає замовник, друга - те, що зроблять програмісти;

- Проектна документація ПЗ (SDD – Software Design Document) - це архітектура і деталі проектування додатку;

- Документація по тестуванню ПЗ (STD – Software Test Documentation) - опис того, як повинно проводитись тестування, хто при цьому присутній, результати тестування і т.д.

1. Сертифікація процесів створення ПЗ

Сучасні контракти і плани на створення складних програмних засобів для інформаційних систем готуються та оцінюються часто некваліфіковано, на основі неформалізованих уявлень замовників та розробників про необхідні функції та їхні характеристики. В технічних завданнях та реалізованих

програмних продуктах систематично замовчуються та/або недостатньо повно формалізуються вимоги, властивості та метрики якості продукту, якими характеристиками вони повинні описуватись, як їх слід вимірювати та порівнювати з вимогами, відображеними иу контракті, технічному завданні чи специфікації ПЗ. Крім того, деякі характеристики часто взагалі відсутні у вимогах замовника та, відповідно, й у документах на ПЗ, що призводить до довільного їх врахування або до пропущення їх при випробуваннях. Розмаїття галузей застосування комп'юетрів стає все ширшим, їхня коректна робота часто є визначальною для якісного управління об'єктами, для успішності підприємств або для безпеки людини. Тому ретельне специфікування та оцінювання необхідних характеристик ПЗ – ключовий фактор забезпечення його адекватного застосування. Це може бути досягнуто на основі виділення та визначення прийнятних характеристик з врахуванням цілей використання та функціональних задач ПЗ.

Сертифікація ПЗ – це процедура підтвердження відповідності ПЗ вимогам, за допомогою якої незалежне від розробника та споживача підприємство юридично засвідчує в письмовій формі, що стан продукції та/або виробництва і системи менеджменту якості здатний забезпечити стабільність характеристик продукції, що виготовляється, і відповідає встановленим замовником вимогам і стандартам. При сертифікаційних випробуваннях якості готового ПЗ можуть використовуватись стандартизовані кількісні та якісні критерії і характеристики, які безпосередньо відбивають функції та властивості продукції, які цікавлять замовника та споживачів, їх можна виміряти та достовірно встановити. Сертифікація ПЗ – це стандартизований, апробований механізм цілеспрямованих регламентованих випробувань виробництва та продуктів, який повинен бути в максимальному ступені спрямований на протидію різним загрозам та нейтралізацію їхніх негативних наслідків.

Сертифікація може виконуватись двома способами:

1) Сертифікація процесів виробництва (технологій забезпечення життєвого циклу ПЗ);

2) Сертифікація продуктів виробництва (готового ПЗ та експлуатаційної документації).

Для забезпечення високої якості ПЗ часто використовують сумісно обидва способи сертифікації. Така комплексна сертифікація забезпечує контроль реалізації вимог алгоритмічної та функціональної коректності ПЗ, що особливо важливо для забезпечення безпеки критичних систем, а також для скорочення випадкових дефектів та помилок програм. Сертифікація технологічних процесів відносно слабо пов'язана з конкретною функціональністю продукту і повинна забезпечувати, в основному, його конструктивну безпеку – мінімум ризиків, дефектів та технічних помилок.

Основною метою сертифікації технологій проектування та виробництва програмних засобів і систем є захист інтересів користувачів, державних та відомчих інтересів на основі контролю якості продукції, забезпечення їхніх високих споживацьких властивостей, підвищення ефективності витрат у сфері їх виробництва, експлуатації та супроводу, підвищення об'єктивності оцінок характеристик та забезпечення конкурентоздатності кінцевого продукту.

При аналізі та організації процесів сертифікаційних випробувань технологій та/або об'єктів системи і комплексу програм слід враховувати ряд базових компонентів методології сертифікації, які підлягають розгляду, застосуванню та затвердженню для конкретного проекту: 1) цілі сертифікації – правові, економічні, формальні; 2) проблеми, які необхідно розв'язувати для забезпечення високої ефективності та достовірності результатів сертифікаційних випробувань; 3) початкові дані і документи, необхідні для проведення сертифікації: стандарти та нормативні документи, їх структура та зміст; 4) характеристики та класифікація продуктів та/або процесів сертифікації та їхні показники якості; 5) ресурси забезпечення випробувань – фінансові, кадри фахівців, апаратна оснащеність, нормативні та інструментальні засоби.

Економічну рентабельність сертифікації кількісно визначити неможливо. Її ефект зосереджений на підвищенні таких

показників якості, як адекватність функціонування, надійність, своєчасність представлення інформації, повнота, достовірність, конфіденційність, безпека застосування ПЗ, які часто важко представити та оцінити конкретними економічними категоріями.

В залежності від області застосування ПЗ, його призначення і класу програмних продуктів їхня сертифікація може бути обов'язковою чи добровільною. Обов'язкова сертифікація необхідна для програмних продуктів, які виконують особливо важливі функції, в яких недостатня якість, помилки або відмови можуть завдати великих збитків або небезпечні для життя і здоров'я людей (авіація, космос, атомна енергетика, військові системи, системи державного керування, фінансові, банківські, транспортні системи). Необхідність проведення обов'язкової сертифікації, як правило, визначає замовник або споживач ПЗ для одержання формальних гарантій досягнення виробником заданих значень показників якості та безпеки ПЗ. *Добровільна сертифікація* застосовується з метою підвищення конкурентоздатності продукції, розширення сфери її використання та одержання додаткових економічних переваг (велика тиражність ПЗ, велика тривалість ЖЦ ПЗ з множиною версій, зниження податків за високу якість, збільшення прибутку розробників та постачальників ПЗ, скорочення рекламацій користувачів). Результати сертифікації повинні виправдати витрати на її проведення внаслідок одержання користувачами продукції більш високої та гарантованої якості при деякому підвищенні її вартості. Таким сертифікаційним випробуванням підлягають компоненти операційних систем і пакети прикладних програм широкого застосування. Необхідність добровільної сертифікації зазвичай визначає розробник або постачальник, за ініціативою яких формується сукупність характеристик якості та безпеки продукції, а також її призначення.

При позитивному результаті сертифікації заявник одержує сертифікат відповідності продукції, який видається на основі оцінювання відповідності технології або ПЗ діючим та/або спеціально розробленим документам: 1) міжнародним та державним стандартам на технології створення ПЗ та конкретну продукцію; 2) стандартам на супроводжуючу ПЗ документацію з

врахуванням необхідності та достатності номенклатури документів, семантичної повноти та однозначності розуміння вмісту документів; 3) нормативним та експлуатаційним документам на конкретне ПЗ – технічним умовам, технічним описам, специфікаціям вимог; 4) діючим міжнародним та національним стандартам на тестування, випробування, атестацію програм, вимоги яких не нижче вимог, регламентованих вітчизняними документами.

Сертифікаційні випробування є найбільш формалізованим та регламентованим етапом тестування як продуктів, так і процесів їх створення, який підтримується значною кількістю документів.

Наразі виникла і вимагає вирішення проблема забезпечення *сертифікації функціональної безпеки ПЗ* як однієї з найважливіших споживацьких властивостей продукту (функціональна безпека – властивість запобігання реалізації потенційних загроз, спрямованих на порушення штатного режиму та зниження якості функціонування систем, а також на нейтралізацію наслідків їх негативного впливу).

Сертифікація процесів розроблення ПЗ передбачає перевірку організації та керування на підприємстві регламентованих та стандартизованих технологічних процесів життєвого циклу комплексів програм.

Для підготовки до сертифікації процесів виробництва ПЗ керівники та фахівці повинні засвоїти та розуміти: 1) методичні основи життєвого циклу та технології виробництва ПЗ; 2) концептуальні та організаційні основи адміністративного керування життєвим циклом і якістю ПЗ на основі принципів стандартів ISO 9000:2000; 3) мету системного проектування – підготувати, обґрунтувати та узгодити задуми замовника і рішення розробника; 4) мету керування проектом – раціональне використання та попередження втрати ресурсів шляхо збалансованого розподілу їх за частковими роботами протягом всього ЖЦ ПЗ; 5) стратегічне планування ЖЦ проекту повинно містити довготривалі цілі розвитку та модифікації ПЗ; 6) методи та засоби техніко-економічного обґрунтування трудомісткості та тривалості виробництва ПЗ, а також кількості необхідних фахівців;

7) підготовку контракту (договору) на детальне проектування або на весь життєвий цикл ПЗ.

Вибір процесів виробництва та всього життєвого циклу залежить від характеристик конкретного проекту, використовуваних стандартів та стану технології.

Організація сертифікаційних випробувань ПЗ на відповідність вимогам передбачає:

1) формулювання цілей, задач та процесів сертифікаційних випробувань;

2) формулювання стратегій, планування, формування графіків тестування та визначення ресурсів попередніх випробувань на відповідність вимогам;

3) оцінювання витрат на випробування ПЗ.

Основою для формування функціональних показників, які оцінюються при сертифікації, є аналіз властивостей, які характеризують функціонування створюваного ПЗ з врахуванням технологічних можливостей розробника. Під функціонуванням передбачається множина властивостей, які обумовлюють придатність ПЗ забезпечувати надійне та своєчасне представлення потрібної інформації для її подальшого функціонального використання (відповідність реальних функціональних можливостей ПЗ декларованим у програмній документації; відповідність реальних функціональних можливостей апаратних засобів декларованим у технічній та експлуатаційній документації; забезпечення точності оброблення безпомилкової та актуальної початкової інформації за допомогою ПЗ для одержання в результаті достатньо достовірної використовуваної інформації; відповідність вимогам функціональних стандартів в галузі взаємодії, супроводжуваності та можливості переносу ПЗ на різні обчислювальні платформи у його життєвому циклі; відповідність вимогам нормативних документів щодо виконання мір забезпечення захищеності інформаційних та програмних ресурсів від несанкціонованого доступа).

Підготовка сертифікаційних випробувань ПЗ передбачає:

1) висування вимог до кваліфікації випробувачів ПЗ;

2) підготовку тестів для випробування ПЗ;

3) висування вимог до тестів для випробування модулів та компонентів;

4) висування вимог до генерації та компонентів динамічних тестів зовнішнього середовища для випробувань продукту в реальному часі;

5) вибір засобів обробки результатів та оцінювання ефективності динамічної генерації тестів для випробувань в реальному часі.

При сертифікації найбільше значення має повнота відповідності початкових вимог до функцій і характеристик та просторів виконаних перевірок при тестуванні. *Сертифікаційні випробування ПЗ на відповідність вимогам* передбачають:

1) встановлення порядку сертифікаційних випробувань ПЗ;

2) розроблення програми та методики випробувань ПЗ;

3) випробування відповідності функціональних та конструктивних характеристик ПЗ заданим вимогам;

4) випробування надійності, функціональної безпеки та динамічних характеристик ПЗ.

Засвідчення якості та завершення сертифікаційних випробувань ПЗ на відповідність вимогам передбачають:

1) випробування для скорочення та ліквідації небезпечних ризиків;

2) випробування експлуатаційної документації на відповідність вимогам до ПЗ;

3) завершення випробувань та оформлення документації результатів, затвердження акту сертифікаційних випробувань;

4) тиражування та впровадження сертифікованих версій ПЗ.

Для сертифікації ПЗ, перш за все, необхідно розуміти його основні властивості, вміти виділяти та формалізувати потрібні характеристики. *Ретельне специфікування вимог до ПЗ – ключовий фактор забезпечення його сертифікації та адекватного застосування.* Це може досягатись на основі виділення та визначення підходящих характеристик ПЗ з врахуванням цілей використання та функціональних задач ПЗ.

2. Оцінювання процесів створення ПЗ

При оцінюванні вартості проекту та кількості часу, потрібного для його виконання, слід виконати два основних етапи: 1) оцінювання розміру проекту; 2) визначення загального обсягу трудовитрат та, відповідно, вартості робіт.

Одержання оцінки можливого *розміру ПЗ* значно спрощується в процесі виконання робіт в рамках програмного проекту. Якщо є потреба у визначенні розміру на початковому етапі життєвого циклу, коли для оперування доступним є лише невеликий обсяг відомостей, то ця оцінка буде мати прогнозний характер. Але саме на базі такої ранньої оцінки робиться загальний висновок, чи варто займатись даним проектом надалі, на яких умовах слід заключати контракт на його виконання. Саме завдяки ранній оцінці можливого розміру ПЗ можна говорити про розмір планованих трудовитрат.

В якості одиниць вимірювання *розмірів програмного проекту* можуть виступати різні величини, вибір яких залежить від конкретного проекту та потреб організації. Розмір ПЗ визначається у термінах рядків коду (LOC – Lines of Code), функціональних точок та точок властивостей. Також можна враховувати кількість «жирних» точок на діаграмі потоку даних, оцінювати кількісні характеристики специфікацій процесу/керування, кількість основних записів та/або взаємозв'язків на діаграмі взаємозв'язку сутностей (ERD).

Для знаходження *очікуваної LOC-оцінки* - за кожною функцією експерти надають краще, гірше та імовірне значення LOC-оцінки, тоді очікувана LOC-оцінка функції обчислюється за формулою: $LOC_{оч_i} = (LOC_{кращ_i} + LOC_{гірш_i} + 4 \times LOC_{імов_i})/6$. Відповідно, очікувана LOC-оцінка всієї програми буде сумою очікуваних LOC-оцінок всіх її функцій.

Показник LOC залежить від мови програмування. Це оціночна і необов'язкова метрика. Вона може призвести до оптимізації кількості рядків коду, а не проекту в цілому. Дозволяє спрогнозувати трудовитрати, час і вартість розробки, а також необхідну кількість програмістів для виконання проекту. Є вільно поширювані інструменти очікуваної LOC-оцінки [http://www.construx.com]. LOC-оцінка впливає на оцінку величини змін обсягу коду в часі.

Одним із шляхів оцінювання розміру ПЗ на етапі проектування є порівняння його функціональних властивостей з вже існуючими аналогами і відповідне прогнозування LOC-оцінки за аналогією. Звісно, такий метод не є надто точним, оскільки до розроблюваного ПЗ додаватимуться нові функціональні можливості, воно може розроблятись іншою мовою і т.і.

Холстед пропонує наступну *міру довжини модуля:* $N = n_1 \log_2(n_1) + n_2 \log_2(n_2)$, де n_1 - кількість різних операторів, n_2 - кількість різних операндів, а також *обсяг модуля* як кількість символів для запису всіх операторів і операндів тексту програми: $V = N \times \log_2(n_1 + n_2)$.

Основу метрик Холстеда складають 4 вимірювані характеристики програми: $NUOprtr$ - кількість унікальних операторів програми включно з іменами підпрограм (словник операторів), $NUOprnd$ - кількість унікальних операндів програми (словник операндів), $NOprtr$ - загальна кількість операторів в програмі, $NOprnd$ - загальна кількість операндів програми.

На основі цих характеристик обчсилюються *наступні оцінки*: словник програми - $HPVoc = NUOprtr + NUOprnd$; довжина програми - $HPLen = NOprtr + NOprnd$ (впливає на оцінку величини змін обсягу коду в часі); обсяг програми - $HPVol = HPLen \times \log_2 HPVoc$ (впливає на оцінку величини змін обсягу коду в часі); складність програми - $HDiff = (NUOprtr/2) \times (NOprnd/NUOprnd)$; оцінка зусиль програміста при розробці програми $HEff = HDiff \times HPVol$ - вказує, наскільки ефективна праця розробника. Такі оцінки (особливо оцінка зусиль програміста) впливають на точність прогнозів оцінки трудомісткості та на розуміння того, наскільки інтелектуально-витратною для розробника буде та чи інша функція.

Альтернативою LOC-оцінки є *прогнозована кількість операторів програми* - $N_{прогн} = NF \times N_{од}$, де NF - кількість функцій або вимог в специфікації програми, $N_{од}$ - одиничне значення кількості операторів (середня кількість операторів, які доводяться на одну функцію або вимогу).

Суть можливостей майбутньої програми відображає *функційний розмір (FP)*. Для обчислення функційного розміру: ідентифікуються очікувані від програмного додатку функції за критеріями International Function Point Users Group (IFPUG). *Метод визначення функційного розміру опишемо покроково наступним чином*:

- виділити функції додатку;
- для кожної потенційної виділеної функції слід порахувати кількість зовнішніх входів EI, які по-різному впливають на виконувану функцію; кількість зовнішніх виходів EO, для істотно різних алгоритмів і нетривіальної функційності; кількість зовнішніх запитів EIN; кількість внутрішніх логічних файлів або унікальних логічних груп користувацьких даних ILF; кількість зовнішніх логічних файлів або унікальних логічних груп користувацьких даних ELF;
- кожен з визначених на попередньому кроці факторів множиться на коефіцієнт, який визначається складністю даного фактору в програмному проекті (EI - 3 або 4 або 6 (від простого до складного), EO - 4 або 5 або 7, EIN - 3 або 4 або 6, ILF - 7 або 10 або 15, ELF - 5 або 7 або 10). Ці добутки додаються за кожною функцією;
- визначити ваги для 14 загальних характеристик проекту (від 0 до 5); ваги вказуються в формі діапазонів, що відображає невпевненість відносно функцій; призначення ваг вимагає певного досвіду у використанні методу функційного розміру;
- обчислити уточнений функційний розмір (ФР) за формулою:

$$Уточн.ФР = Наближений\_функц\_розмір \times$$

$$\times[0.65 + 0.01 \times (Сума\_загальних\_характеристик)]$$

Якщо до проекту не висувається жодних спеціальних вимог (всі загальні характеристики дорівнюють 0), то неуточнений функційний розмір слід зменшити на 35%, інакше слід збільшити на 1% на кожну одиницю значень загальних характеристик.

Функційний розмір використовується як відносна метрика для порівняння з попередніми проектами, за його допомогою можна обчислити кількість рядків коду, що дозволяє визначити загальну трудомісткість та терміни проекту. Є вільно поширювані

інструменти для обчислення функційного розміру [http://ifpug.org/home/docs/freebies.html].

На основі розміру проекту (в основному, на основі LOC-оцінок) можна обчислити вартість, трудовитрати та тривалість програмного проекту.

Очікувана вартість розробки кожної функції: $ВАРТІСТЬ_i = LOC_{оч_i} \times ВАРТІСТЬ_{рядка_i}$; вартість рядка є константою і не змінюється від реалізації до реалізації. Відповідно вартість ПЗ є сумою вартостей розробки всіх його функцій. *Прогнозовані витрати на розробку кожної функції*: $ВИТРАТИ_i = (LOC_{оч_i} / ПРОДУКТ_i)$. І відповідно витрати на ПЗ є сумою витрат на розроблення всіх його функцій.

Прогнозована оцінка трудовитрат та тривалості проекту за моделлю Боема -трудовитрати на розробку програмних додатків зростають швидше, ніж розмір додатків. Для представлення даного співвідношення використовується експоненційна функція. Тривалість проекту за моделлю Боема зростає експоненційно разом з докладеними до проекту зусиллями.

Боем оцінив параметри розглядуваних вище співвідношень:

$$Трудовитрати = a \times KLOC^b;$$

$$Тривалість = c \times Трудовитрати^d = c \times (a \times KLOC^b)^d = c \times a^d \times KLOC^{b \times d},$$

де $KLOC$ - кількість тисяч рядків коду, a, b, c, d - коефіцієнти COCOMO. Для органічного (самостійного) програмного проекту: $a = 2.4, b = 1.05, c = 2.5, d = 0.38$. Для вбудованих програмних проектів (інтеграція апаратного та програмного забезпечення): $a = 3.6, b = 1.20, c = 2.5, d = 0.32$. Для проміжних програмних проектів (не органічні, але й не жорстко вбудовані): $a = 3.0, b = 1.12, c = 2.5, d = 0.35$.

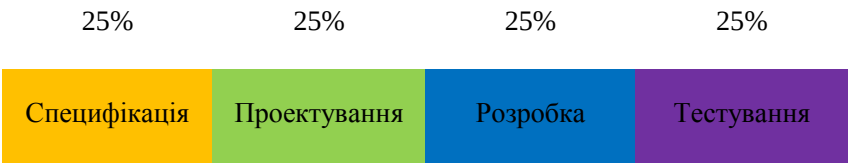
Висновки

На початкових етапах життєвого циклу для оперування доступним є лише невеликий обсяг відомостей, тому оцінки розміру, вартості, тривалості, трудовитрат програмного проекту мають прогнозний характер. Але саме на основі таких ранніх

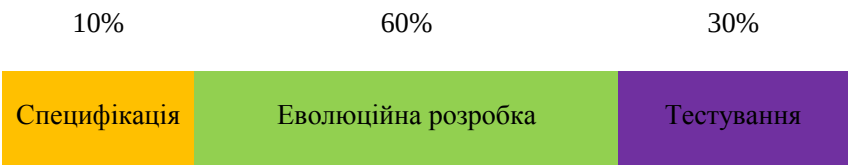
оцінок робиться загальний висновок, чи варто займатись даним проектом надалі.

Розподіл часу та витрат на різні етапи життєвого циклу ПЗ представлений нижче.

Окремий розрахунок витрат на кожен етап виробництва ПЗ:



Структура витрат при використанні еволюційного ЖЦ



Врахування вартості супроводу контрактного ПЗ

ПЗ для ПК

Лекція №4
СУЧАСНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Визначення технології проектування ПЗ (ТППЗ). Загальні вимоги, пропоновані до ТППЗ. Приклади ТППЗ
2. Моделі систем. Прототипування програмних систем

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.
2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.
4. А.М. Вендров. Современные технологии создания программного обеспечения // [Електронний ресурс] – Режим доступу: http://citforum.ru/programming/application/program/
5. Ю. Ю. Якунин. Технологии разработки программного обеспечения - Красноярск: ИПК СФУ, 2008
6. Д.А.Чернев. Технология разработки програмного обеспечения – Ташкент, 2004 – 224 с.
7. IEEE 1209-1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools.
8. IEEE 1348-1995. IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools.
9. ISO/IEC 14102:1995(E). Information technology - Guideline for the evaluation and selection of CASE Tools.

Вступ

Керування розробкою ПЗ – це галузь знань, в якій визначаються аспекти керування і кількісних оцінок інженерії ПЗ.

Факти, що визначають специфіку ПЗ як продукту: замовник не розуміє складності процесу розробки ПЗ і впливу зміни його вимог до ПЗ в процесі розробки; зміна та поява нових вимог в процесі розробки ПЗ є неминучими; ітеративність процесу розробки ПЗ, яка ускладнює його; рівень новизни та складності ПЗ є дуже високим; технології швидко змінюються, оновлюються та застарівають.

Керування розробкою програмного забезпечення здійснюється на 3-х рівнях:

- організаційне керування та керування інфраструктурою;
- керування проектами;
- планування і контроль програм кількісного оцінювання показників ПЗ.

У керуванні програмною інженерією важливими є знання щодо: керування інтеграцією проекта (project integration management); керування змістом проекта (project scope management); керування термінами проекта (project time management); керування вартістю проекта (project cost management); керування якістю проекта (project quality management); керування людськими ресурсами проекта (project human resource management); керування комунікаціями проекта (project communication management); керування ризиками проекта (project risk management); керування постачанням проекта (project procurement management).

Всі специфічні факти, рівні, важливі знання, заходи, операції, прийоми щодо керування розробкою ПЗ включає в себе технологія проектування програмного забезпечення.

1. Визначення технології проектування (ТП). Загальні вимоги, пропоновані до ТППЗ. Приклади ТППЗ

Технологія – комплекс організаційних заходів, операцій та прийомів, спрямованих на виготовлення, обслуговування, ремонт та/або експлуатацію виробу з номінальною якістю та оптимальними витратами.

Технологія проектування (розроблення) програмного забезпечення (ТППЗ) – це комплекс організаційних заходів,

операцій та прийомів, спрямованих на розроблення програмних продуктів високої якості в рамках відведеного бюджету і в термін. ТППЗ – це впорядкована сукупність взаємозв'язаних технологічних процесів в межах життєвого циклу ПЗ.

ТППЗ в загальному випадку можна описати наступною системою понять. *Технологічний процес* – сукупність взаємозв'язаних технологічних операцій. *Технологічна операція* – основна одиниця роботи, виконувана певною роллю, яка: передбачає чітко визначену відповідальність ролі; дає чітко визначений результат (набір робочих продуктів), який базується на певних початкових даних (іншому наборі робочих продуктів); представляє собою одиницю роботи з жорстко визначеними межами, які встановлюються при плануванні проекту. *Робочий продукт* – це інформаційна або матеріальна сутність, яка створюється, модифікується або використовується в деякій технологічній операції (модель, документ, код, тест і т.і.). Робочий продукт визначає область відповідальності ролі і є об'єктом керування конфігурацією. *Роль* – визначення поведінки та обов'язків окремої особи або групи осіб в середовищі організації-розробника ПЗ, які здійснюють діяльність в межах деякого технологічного процесу та відповідають за певні робочі продукти. *Керівництво* – це практичне керівництво по виконанню однієї або сукупності технологічних операцій. Керівництва містять методичні матеріали, інструкції, нормативи, стандарти та критерії оцінювання якості робочих продуктів. *Інструментальний засіб (CASE-засіб)* – це програмний засіб, який забезпечує автоматизовану підтримку діяльності, виконуваної в межах технологічних операцій.

Основною вимогою, яка висувається до сучасних ТППЗ, є їхня відповідність стандартам та нормативним документам, пов'язаним з процесами ЖЦ ПЗ та оцінкою технологічної зрілості організацій-розробників (ISO 12207, ISO 9000, CMM та ін.). Згідно цих нормативів, ТППЗ повинна підтримувати наступні процеси: керування вимогами, аналіз та проектування ПЗ, розроблення ПЗ, експлуатація, супровід, документування, керування конфігурацією та змінами, тестування, керування проектом. Повнота підтримки процесів ЖЦ ПЗ повинна підтримуватись комплексом

інструментальних засобів. Відповідність стандартам означає також, зокрема, використання загальноприйнятих стандартних нотацій та угод, стандартних методів моделювання, які повинні бути оформлені у вигляд нормативів до початку процесу проектування. Недотримання проектних стандартів ставить розробників у залежність від фірми-виробника даного засобу, ускладнює формальний контроль коректності проектних рішень, зменшує можливості залучення додаткових колективів розробників, заміни виконавців та відчуження проекту через обмежену кількість фахівців, знайомих з даним методом (нотацією).

Іншою важливою вимогою є адаптованість до умов застосування, яка досягається за рахунок поставки технології в електронному вигляді разом із CASE-засобами та бібліотеками процесів, шаблонів, методів, моделей та інших компонентів, призначених для побудови ПЗ того класу систем, на який орієнтована технологія. Електронні технології повинні містити засоби, які забезпечують їх адаптацію та розвиток за результатами виконання конкретних проектів. Процес адаптації полягає у видаленні непотрібних процесів та дій ЖЦ ПЗ, в заміні неприйнятних або у додаванні власних процесів та дій, а також методик, стандартів та керівництв.

При впровадженні ТППЗ слід керуватись рекомендаціями стандартів [IEEE 1209-1992, IEEE 1348-1995, ISO/IEC 14102:1995], які відбивають досвід, накопичений багатьма поколіннями користувачів та розробників ТППЗ. Під впровадженням ТППЗ розуміють всі дії – від оцінювання початкових потреб до повномасштабного використання ТППЗ ту різних підрозділах організації. *Процес впровадження складається з наступних етапів*: 1) визначення потреб у ТППЗ, характеристик об’єкту впровадження та проектів створення ПЗ; 2) визначення вимог, які висуваються до ТППЗ (аналіз характеристик об’єкту впровадження та проектів, обґрунтування вимог до ТППЗ, визначення пріоритетів вимог); 3) оцінювання варіантів ТППЗ – попереднє експертне оцінювання, яке полягає у аналізі доступних ТППЗ на предмет відповідності вимогам, та деталізоване оцінювання, яке полягає у формуванні детального опису кожної ТППЗ-претендента; 4) вибір ТППЗ на

основі порівняльного аналізу технологій та з врахуванням експертної оцінки; 5) адаптація ТППЗ до умов застосування шляхом формування конкретної робочої конфігурації ТППЗ, адаптованої до умов об'єкту впровадження. Під час впровадження ТППЗ накопичується статистика та оцінюється ефективність її впровадження з точки зору ряду критеріїв (мінімум трудомісткості супроводу ПЗ, мінімум витрат на супровід ПЗ та ін.).

Метою процесу оцінювання ТППЗ є визначення функціональності та якості ТППЗ для наступного вибору. Оцінювання виконується відповідно до конкретних критеріїв, його результати містять як об'єктивні, так і суб'єктивні дані по кожній ТППЗ. Процеси оцінювання й вибору тісно взаємопов'язані. За результатами оцінювання цілі вибору та/або критерії вибору, а також їхні вагові коефіцієнти можуть вимагати модифікації. В таких випадках може знадобитись повторне оцінювання. Коли аналізуються кінцеві результати оцінювання та до них застосовуються критерії вибору, то може бути рекомендовано придбання технології. Альтернативою може стати відсутність адектваної технології, в такому випадку рекомендується розробити нову технологію, модифікувати існуючу або відмовитись від впровадження.

Процес вибору ТППЗ включає в себе наступні дії:

1) формулювання задач вибору, включаючи цілі, припущення та обмеження;

2) виконання всіх необхідних дій по вибору, включаючи визначення та ранжування критеріїв, визначення технологій-кандидатів, збір необхідних даних та застосування ранжованих критеріїв до результатів оцінювання для визначення засобів з найкращими показниками;

3) виконання необхідної кількості ітерацій з метою вибору (або відхилення) технології, яка має подібні до інших показники.

Типовий процес оцінювання та/або вибору може використовувати набір критеріїв різних типів. Кожний критерій повинен бути обраний та адаптований експертом з врахуванням особливостей конкретного процесу. *Початковими даними для оцінювання та вибору* є набір параметрів ТППЗ: 1) функціональні

характеристики, орієнтовані на процеси ЖЦ ПЗ (керування проектом, керування вимогами, керування конфігурацією та змінами, аналіз та проектування ПЗ і т.і.); 2) функціональні характеристики застосування (середовище функціонування, сумісність з іншими ТППЗ, відповідність технологічним стандартам); 3) характеристики якості (надійність, зручність використання, ефективність, супроводжуваність, можливість переносу); 4) загальні характеристики (витрати на технологію, ліцензійна політика, оціночний ефект від впровадження ТППЗ, потрібна для впровадження інфраструктура, доступність та якість навчання, сертифікація постачальника, підтримка постачальника). На основі даного набору параметрів аналізуються та класифікуються існуюча ТППЗ. В результаті виконаного оцінювання може виявитись, що жодна доступна технологія не задовольняє в потрібній мірі всіх критеріїв і не покриває всі потреби проекту. В такому випадку може застосовуватись набір засобів, який дозволяє побудувати єдине технологічне середовище.

Загальний набір критеріїв, застосовуваних для оцінювання ТППЗ, наведено у таблиці 4.1.

Таблиця 4.1 – Критерії оцінювання ТППЗ

Критерій	**Визначення**
Мінімум трудомісткості створення ПЗ	Кількість людино-місяців, які витрачаються на створення ПЗ з використанням ТППЗ
Максимум продуктивності	Обсяг роботи (вимірюваний в кількості рядків коду або функціональних точок), який доводиться на одиницю трудомісткості (людино-місяць) при використанні даної ТППЗ
Максимум якості створюваного ПЗ	Кількість дефектів у робочих продуктах при використання даної ТППЗ
Повернення інвестицій	(Прибуток від використання ПЗ – Витрати на створення та супровід ПЗ) / (Витрати на створення та супровід ПЗ)
Мінімум витрат на супровід ПЗ	Відношення вартості супроводу ПЗ при використанні даної ТППЗ до сукупних витрат на інформаційні ТППЗ в організації

Мінімум часу впровадження ТППЗ	Часовий інтервал від початку впровадження ТППЗ до виходу на беззбитковий рівень (початок повернення інвестицій в ТППЗ)
Мінімум витрат на впровадження ТППЗ	Сумарна вартість придбання, навчання та супроводу ТППЗ
Мінімальний термін окупності витрат на впровадження ТППЗ	Часовий інтервал від початку впровадження ТППЗ до повної окупності витрат на її впровадження

Перед повномасштабним впровадженням обраної ТППЗ виконується *пілотний проект*, метою якого є експериментальна перевірка вірності рішень, прийнятих на попередніх етапах, та підготовка до впровадження. Важливою функцією пілотного проекту є прийняття рішення стосовно придбання або відмови від використання ТППЗ. Провал пілотного проекту дозволяє уникнути подальших більш значних і коштовних невдач. Пілотний проект повинен мати наступні *характеристики*: 1) типовість предметної галузі; 2) масштабованість; 3) презентабельність, показовість; 4) критичність; 5) авторитетність; 6) готовність проектної групи. В процесі оцінювання пілотного проекту організація повинна визначити свою позицію за наступними трьома питаннями: 1) чи доцільно впроваджувати ТППЗ?; 2) які конкретні особливості пілотного проекту призвели до його успіху (або невдачі)?; 3) які проекти або підрозділи організації могли б отримати вигоду від використання ТППЗ? Можливим рішенням може бути одне з наступних: 1) впровадити ТППЗ; 2) виконати додатковий пілотний проект; 3) відмовитись від ТППЗ; 4) відмовитись від використання ТППЗ взагалі.

Процес переходу до практичного використання ТППЗ починається з розроблення та наступної реалізації плану переходу, який відбиває поетапний підхід до переходу – від ретельно обраного пілотного проекту до проектів з розмаїттям характеристик, яке істотно зросло. План переходу повинен визначати початкову практику застосування та процедури використання засобів. Реальне застосування будь-якої ТППЗ в конкретних організації і проекті неможливе без відпрацювання

ряду стандартів, правил, угод, яких повинні дотримуватись всі учасники проекту. Для успішного впровадження ТППЗ істотною є послідовність в її застосуванні, а також характер майбутнього використання ТППЗ як окремими розробниками, так і групами.

Приклади провідних ТППЗ різних компаній-постачальників:

1) Rational Unified Process (RUP);

2) Oracle;

3) Borland;

4) Computer Associates.

Розглянемо детально кожну з вищеназваних ТППЗ.

Rational Unified Process (RUP) – одна з найбільш досконалих технологій, яка претендує на роль світового корпоративного стандарту. Це програмний продукт, розроблений компанією Rational Software (www.rational.com), яка наразі входить до складу IBM. RUP в значній мірі відповідає стандартам та нормативним документам, зв'язаним з процесами ЖЦ ПЗ та оцінкою технологічної зрілості організацій-розробників. Її основні принципи представлені на рис.4.1.

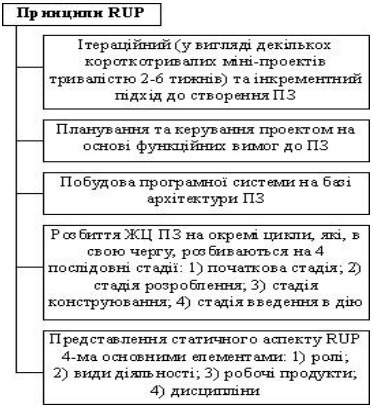


Рис.4.1 - Основні принципи технології RUP

Перший принцип є визначальним. Відповідно до нього розроблення системи виконується у вигляді декількох короткотривалих міні-проектів фіксованої тривалості (2-6 тижнів), які називаються ітераціями. Кожна ітерація включає свої власні

етапи аналізу вимог, проектування, реалізації, тестування, інтеграції та завершується створення працюючої системи. Ітераційний цикл базується на постійному розширенні та доповненні системи в процесі декількох ітерацій з періодичним зворотнім зв'язком та адаптацією подаваних модулів до ядра ПЗ.

Згідно RUP, *ЖЦ ПЗ розбивається на окремі цикли*, в кожному з яких створюється нове покоління продукту. Кожний *цикл*, в свою чергу, *розбивається на чотири послідовні стадії*: 1) початкова стадія (inception); 2) стадія розроблення (elaboration); 3) стадія конструювання (construction); 4) стадія введення в дію (transition). Кожна стадія завершується в чітко визначеній контрольній точці. В цей момент часу повинні досягатись важливі результати і прийматись критично важливі трішення про подальше розроблення.

Початкова стадія - це всебічне вивчення всіх можливостей реалізації проекту, яке може тривати місяці. Під час початкової стадії розробляється бізнес-план проекту – визначається, скільки він коштуватиме і який прибуток принесе, визначаються межі проекту і виконується деякий початковий аналіз для оцінювання розмірів проекту. Результати початкової стадії: загальний опис системи; початкова модель варіантів використання; початковий проектний глосарій; початковий бізнес-план; план проекту зі стадіями та ітераціями; один або декілька прототипів.

На *стадії розроблення* виявляються більш детальні вимоги до системи, виконується високорівневий аналіз предметної області та проектування для побудови базової архітектури системи, створюється план конструювання та усуваються найбільш ризиковані елементи проекту. Найважливішим результатом стадії розроблення є опис базової архітектури (модель предметної області та технологічна платформа), яка є основою подальшого розроблення, тобто проектом для наступних стадій. Результати стадії розроблення: модель варіантів використання; перелік додаткових вимог із нефункційними вимогами включно; опис базової архітектури майбутньої системи; працюючий прототип; уточнений бізнес-план; план розроблення всього проекту з усіма ітераціями та критеріями оцінювання кожної ітерації. Стадія

розроблення займає близько п'ятої частини загальної тривалості проекту. Основними ознаками завершення стадії розроблення є дві події: 1) розробники можуть оцінити з досить високою точністю, скільки часу потрібно на реалізацію кожного варіанту використання; 2) ідентифіковані всі найбільш серйозні ризики, і ступінь розуміння найбільш важливих з них така, що відомо, як із ними впоратись.

Результатом *стадії конструювання* є продукт, готовий до передачі кінцевим користувача. Як мінімум, він має містити: ПЗ, інтегроване на потрібних платформах; керівництва користувача; опис поточної реалізації.

Призначенням *стадії введення в дію* є передача готового продукту у розпорядження користувачів. Дана стадія включає: бета-тестування, паралельне функціонування з існуючою системою, яка підлягає заміні; конвертування баз даних; оптимізацію продуктивності; навчання користувачів та фахівців служби супроводу.

Статичний аспект RUP представлений 4-ма основними елементами: ролі (поведінка та відповідальність особи або групи осіб); види діяльності (одиниця виконуваної конкретним виконавцем роботи; відповідають поняттю технологічної операції); робочі продукти; дисципліни (відповдає поняттю технологічного процесу і представляє собою послідовність дій, яка призводить до одержання значного результату). RUP передбачає 6 основних дисциплін (побудова бізнес-моделей, визначення вимог, аналіз та проектування, реалізація, тестування, розгортання) та 3 допоміжних дисципліни (керування конфігурацією та змінами, керування проектом, створення інфраструктури). Кожна з перерахованих дисциплін підтримується певним інструментальним засобом комплексу Rational Suite.

RUP спирається на *інтегрований комплекс інструментальних засобів Rational Suite*:

- Rational Suite AnalystStudio – визначення та керування повним набором вимог до розроблюваної системи;

- Rational Suite DevelopmentStudio – проектування та реалізація ПЗ;

- Rational Suite TestStudio – автоматичне тестування додатків;

- Rational Suite Enterprise – підтримка повного ЖЦ ПЗ.

До складу Rational Suite, крім самої технології RUP як продукта, входять наступні компоненти:

- Rational Rose – засіб візуального моделювання мовою UML;

- Rational XDE – засіб аналізу та проектування;

- Rational Requisite Pro – засіб керування вимогами, призначений для організації сумісної роботи групи розробників;

- Rational Rapid Developer – засіб швидкого розроблення додатків на платформі Java 2 Enterprise Edition;

- Rational ClearCase – засіб керування конфігурацією ПЗ;

- Rational SoDA – засіб автоматичної генерації проектної документації;

- Rational ClearQuest – засіб для керування змінами та відстежування дефектів у проекті;

- Rational Quantify – засіб кількісного визначення «вузьких» місць, які впливають на загальну ефективність роботи системи;

- Rational Purify – засіб для локалізації помилок часу виконання програми, які важко виявляються;

- Rational PureCoverage – засіб ідентифікації ділянок коду, пропущених при тестуванні;

- Rational TestManager– засіб планування функційного та навантажувального тестування;

- Rational Robot – засіб запису та відтворення тестових сценаріїв;

- Rational TestFactory – засіб тестування надійності;

- Rational Quality Architect – засіб генерації коду для тестування.

Методичну основу *ТППЗ корпорації Oracle* (www.oracle.com) складає метод Oracle (Oracle Method), який охоплює більшість процесів ЖЦ ПЗ. До складу комплекса входять:

- CDM (Custom Development Method) – розроблення прикладного ПЗ;

- PJM (Project Management Method) - керування проектом;

- AIM (Application Implementation Method) – впровадження прикладного ПЗ;

- BPR (Business Process Reengineering) - реінжиніринг бізнес-процесів;

- OCM (Organizational Change Management) – керування змінами.

Метод CDM оформлено у вигляді консалтингового продукту CDM Advantage – бібліотеки стандартів та керівництв. По суті, це методичне керівництво по розробленню прикладного ПЗ з використанням інструментального комплексу Oracle Developer Suite. Відповідно до CDM, ЖЦ ПЗ формується з певних етапів (фаз) проекту та процесів, кожний з яких виконується протягом декількох етапів (рис.4.2):

- стратегія (визначення вимог);

- аналіз (формулювання детальних вимог до системи);

- проектування (перетворення вимог у детальні специфікації системи);

- реалізація (написання та тестування додатків);

- впровадження (встановлення нової прикладної системи, підготовка до початку експлуатації);

- експлуатація.

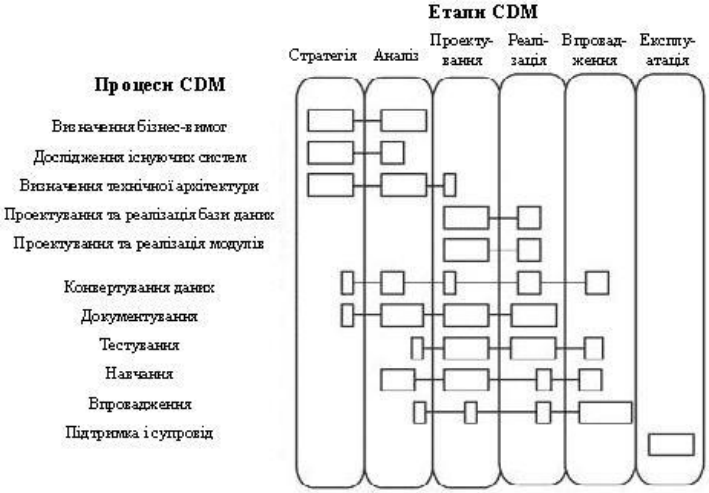


Рис.4.2 – Етапи і процеси CDM

На етапі стратегії визначаються цілі створення системи, пріоритети та обмеження, розробляється системна архітектура та складається план розроблення. На етапі аналізу будуються модель інформаційних потреб (діаграма «сутність-зв'язок»), діаграма функціональної ієрархії, матриця перехресних посилань та діаграма потоків даних. На етапі проектування розробляється детальна архітектура системи, проектуються схема реляційної БД та програмні модулі, встановлюються перехресні посилання між компонентами системи. На етапі реалізації створюється БД, будуються прикладні системи, відбувається їх тестування, перевірка якості та відповідності вимогам користувачів, створюється системна документація, матеріали для навчання та керівництва користувачів. На етапах впровадження та експлуатації аналізуються продуктивність та цілісність системи, виконується підтримка та, за необхідності, модифікація системи.

Процеси CDM:

- визначення бізнес-вимог, або постановка задачі (Business Requirements Definition);
- дослідження існуючих систем для розуміння стану існуючого технічного та програмного забезпечення (Existing Systems Examination);
- визначення технічної архітектури (Technical Architecture);
- проектування та реалізація бази даних (Database Design and Build);
- проектування та реалізація модулів (Module Design and Build) – основний процес проекту;
- конвертування даних (Data Conversion);
- документування (Documentation);
- тестування (Testing);
- навчання (Training);
- впровадження або перехід до нової системи (Transition);
- підтримка та супровід (Post-System Support).

У CDM виділяються два основних підходи до розроблення:

1) класичний підхід – етапи такого підходу представлені на рис.4.2. Застосується для найбільш складних та масштабних проектів, передбачає послідовний та детермінований порядок

виконання задач. Для таких проектів характерна велика кількість бізнес-правил, розподілена архітектура, критичність додатків. Застосування класичного підходу також рекомендується при нестачі досвіду у розробників, непідготовленості користувачів, нечітко визначеній задачі.Тривалість таких проектів від 8 до 36 місяців;

2) підхід швидкого розроблення (Fast Track) – це ітераційний підхід, заснований на методі DSDM (Dynamic Systems Development Method). Складається з 4-х етапів – стратегія, моделювання вимог, проектування та генерація системи, впровадження в експлуатацію. Підхід використовується для реалізації невеликих та середніх проектів з нескладною архітектурою, гнучкими термінами та чіткою постановкою задач. Тривалість проекту від 4 до 16 місяців.

PJM – це певна дисципліна ведення проекту, яка дозволяє гарантувати, що цілі проекту, чітко визначені на його початку, залишаються в центрі уваги протягом всього проекту. Метод керівництва проектом у PJM представляється у вигляді чітко визначеної операційної схеми, в якій виділяються процеси, етапи, задачі, результати розв'язку задач та залежності між задачами:

- керування проектом та надання звітності (Control and Reporting);
- керування роботою (Work Management);
- керування ресурсами (Resource Management);
- керування якістю (Quality Management);
- керування конфігурацією (Configuration Management).

Комплекс Oracle Developer Suite містить набір інтегрованих засобів розроблення для швидкого створення додатків. Він включає в себе засоби моделювання, програмування на Java, розроблення компонентів, бізнес-аналізу та складання звітів. Всі ці засоби використовують спільні ресурси, що довзоляє сумісно працювати над одним проектом групи розробників. Oracle Developer Suite інтегрований з Oracle Database та Oracle Application Server, утворюючи єдину платформу для створення та вставновлення додатків.

Oracle Developer Suite включає:

- Oracle Designer – засіб моделювання та генерації додатків;
- Oracle Forms – засіб швидкого розроблення додатків;
- Oracle Reports – візуальний засіб для розроблення звітів;
- Oracle JDeveloper – засіб візуального програмування мовою Java;
- Oracle Discoverer – засіб для розроблення аналітичних додатків;
- Oracle Warehouse Builder – система для побудови сховищ даних;
- Oracle Portal – засіб розроблення інформаційного порталу організації.

Компанія *Borland* (www.borland.com) в результаті розвитку власних розробок та придбання ряду компаній надала інтегрований комплекс інструментальних засобів, які реалізують керування повним життєвим циклом додатків (*Application Life Cycle Management - ALM*). Відповідно до технології Borland процес створення ПЗ включає 5 основних етапів – рис.4.3.

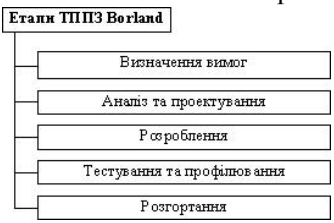


Рис.4.3 - Основні етапи технології Borland

Виконання всіх етапів координується процесом керування конфігурацією та змінами.

Визначення вимог реалізується за допомогою системи керування вимогами CaliberRM, яка зберігає вимоги у БД, створює документи з їх описом на базі заданих шаблонів, підтримує різні методи візуалізації залежностей між вимогами, а також виконує оцінку трудовитрат, ризиків та витрат на основі вимог.

Засіб аналізу та проектування Together ControlCenter за основу використовує один з варіантів підходу швидкого розроблення ПЗ - Feature Driven Development (FDD). Це

інтегроване середовище проектування та розроблення, яке підтримує візуальне моделювання на UML з наступним написанням додатків для платформ J2EE (Java) та .Net (C#, C++, Visual Basic).

Технологія LiveSource забезпечує синхронізацію між проекто додатку та змінами – при внесенні змін у тексти змінюється модель програми, а при зміні моделі належним чином змінюється текст мовою програмування.

Засоби Optimizeit Suite 5, Optimizeit Profiler for .NET та Optimizeit ServerTrace – інструментальні засоби тестування. Перші дві системи дозволяють виявити потенційні проблеми використання апаратних ресурсів – пам'яті та процесорних потужностей. Третя система призначена для керування продуктивністю серверних J2EE-додатків з точки зору досягнення заданого рівня обслуговування та збирання контрольних даних з віртуальних Java-машин.

Сутність концепції ALM зосереджена у системі керування конфігурацією та змінами: саме вона об'єднує основні фази ЖЦ ПЗ. Такою системою є StarTeam, яка виконує функції контролю версій, керування змінами, відстежування дефектів, керування вимогами, керування потоком задач та керування проектом.

В технології Borland виділяється три рівня інтеграції: 1) функційна (touch-point) – дозволяє звернутись з однієї системи до функцій іншої; така інтеграція дає можливість розділяти інформацію між системами, але не забезпечує єдиного робочого простору, змушує користувачів перемикатись між вікнами і призводить до дублювання процесів керування структурою проекта; 2) вбудована (embedded) – забезпечує роботу з однією системою безпосередньо у середовищі іншої; 3) синергетична (synergistic) – найвищий рівень інтеграції, який дозволяє сполучати функції двох різних продуктів непомітно для розробників.

Компанія *Computer Associates* (www.ca.com) пропонує комплекси інструментальних засобів підтримки різних процесів ЖЦ ПЗ, зокрема комплекс AllFusion Modeling Suite – інтегрований комплекс CASE-засобів, який складається з наступних продуктів:

- AllFusion Process Modeler (BPwin) – функційне моделювання;

- AllFusion ERwin Data Modeler (ERwin) – моделювання даних;

- AllFusion Component Modeler (Paradigm Plus) – об'єктно-орієнтований аналіз та проектування за допомогою UML і можливістю генерації коду;

- AllFusion Model Manager (Model Mart) - організація сумісної роботи колективу розробників;

- AllFusion Data Model Validator (ERwin Examiner) – перевірка структури та якості моделей даних.

AllFusion Change Management Suite - комплекс засобів керування конфігурацією та змінами.

AllFusion Process Management Suite – засоби керування процесами та проектами для різних типів додатків.

CASE-засіб BPwin - засіб моделювання бізнес-процесів, яке реалізує метод IDEF0, а також підтримує діаграми потоків даних та IDEF3. Підтримує функційно-вартісний аналіз.

CASE-засіб ERwin - набір засобів концептуального моделювання даних, які використовують метод IDEF1X. Реалізує проектування схеми БД, генерацію її опису мовою цільової СУБД та реверсний інжиніринг існуючої БД.

Для керування груповим розробленням використовується засіб Model Mart, який забезпечує багатокористувацький доступ до моделей. Цей засіб задовольняє ряду вимог: 1) сумісне моделювання; 2) створення бібліотек рішень; 3) керування доступом.

2. Моделі систем. Прототипування програмних систем

Модель ПЗ – це формалізований опис системи ПЗ на певному рівні абстракції. Кожна модель визначає конкретний аспект системи, використовує набір діаграм та документів заданого формату, а також відбиває точку зору та є об'єктом діяльності різних людей з конкретними інтересами, ролями або задачами. *Графічні (візуальні) моделі* – це засоби для візуалізації, опису, проектування та документування архітектури системи. Моделі є

основою взаємодії учасників проекту і гарантують коректність архітектури. Оскільки складність систем зростає, важливо мати гарні методи моделювання. Хоча й існує безліч факторів, від яких залежить успіх проекту, але наявність суворого стандарту мови моделювання є вельми істотним.

Склад моделей, використовуваних в кожному конкретному проекті, та ступінь їх деталізації в загальному випадку залежать від наступних факторів:

- складності проектованої системи;
- необхідності повноти її опису;
- знань та навичок учасників проекту;
- часу, відведеного на проектування.

Візуальне моделювання сильно вплинуло на розвиток ТППЗ взагалі та CASE-засобів зокрема. *CASE-технологія* – це сукупність методів проектування ПЗ, а також набір інструментальних засобів, які дозволяють наочно моделювати предметну галузь, аналізувати цю модель на всіх стадіях розроблення і супроводу ПЗ, а також розробляти додатки відповідно до інформаційних потреб користувача на основі аналізу специфікацій.

При структурному аналізі та проектуванні використовуються різні моделі, які описують:

- функційну структуру системи;
- послідовність виконуваних дій;
- передачу інформації між функційними процесами;
- відношення між даними.

Найбільш поширеними моделями перших трьох груп є:

- функційна модель SADT (Structured Analysis and Design Technique);
- модель IDEF3;
- діаграми потоків даних DFD (Data Flow Diagrams).

Метод SADT – це сукупність правил та процедур, призначених для побудови функційної моделі об’єкту будь-якої предметної галузі. Функційна модель SADT відображає функцій ну структуру об’єкту, тобто дії, які він виконує, та зв’язки між цими діями. Моделі SADT (IDEF0) традиційно використовуються для моделювання організаційних систем (бізнес-процесів). Слід

зазначити, що метод SADT успішно працює лише при описі добре специфікованих та стандартизованих бізнес-процесів у корпораціях, тому принятий у США в якості типового. Перевагами застосування моделей SADT є: повнота опису бізнес-процесу, жорсткі вимоги методу (як результат – моделі стандартного вигляду), відповідність підходу до опису процесів стандартам ISO 9000. Оскільки на Україні бізнес-процеси почали формуватись і розвиватись порівняно недавно, вони слабко типізовані, тому розумніше орієнтуватись на менш жорсткі моделі.

Метод моделювання IDEF3 призначений для таких моделей процесів, в яких важливо зрозуміти послідовність виконання дій та взаємозалежності між ними. Основою моделі IDEF3 є так званий сценарій процесу, який виділяє послідовність дій та підпроцесів аналізованої системи.

Діаграми потоків даних (DFD) – це ієрархія функційних процесів, зв'язаних потоками даних. Мета такого представлення – продемонструвати, як кожний процес перетворює свої вхідні дані у вихідні, а також виявити залежності між цими процесами. Модель системи визначається як ієрархія діаграм потоків даних, які описують асинхронний процес перетворення інформації від її ведення у систему до видачі результатів споживачу. Діаграми потоків даних спочатку створювались як засіб проектування інформаційних систем (SADT – засіб моделювання систем взагалі) і мають більший набір елементів, які адекватно відображають специфіку таких систем.

Розглянуті моделі приблизно однакові з точки зору можливостей зображувальних засобів моделювання. Основним критерієм вибору того чи іншого методу є ступінь володіння ним з боку консультанта або аналітика, грамотність вираження своїх думко мовою моделювання.

Найбільш поширеним засобом моделювання даних (предметної галузі) є *модель «сутність-зв'язок»* (Entity-Relationship Model - ERM). Ця модель представляє собою підмножину об'єктної моделі предметної галузі.

Концептуальною основою *об'єктно-орієнтованого аналізу та проектування (ООАП)* ПЗ є об'єктна модель. Її основні

принципи – абстрагування, інкапсуляція, модульність та ієрархія; основні поняття – об'єкт, клас, атрибут, операція, інтерфейс. Більшість сучасних методів ООАП засновані на використанні уніфікованої мови моделювання UML (Unified Modeling Language). Мову UML використовують всі крупні компанії-виробники ПЗ (Microsoft, Oracle, IBM, Hewlett-Packard, Sybase). Крім того, практично всі світові виробники CASE-засобів підтримують UML у своїх продуктах. Повний опис UML можна знайти на сайтах http://www.omg.org та http://www.rational.com.

Стандарт UML містить наступний набір діаграм:

1) структурні (structural) моделі:

- діаграми класів (class diagrams);
- діаграми компонентів (component diagrams);
- діаграми розташування (deployment diagrams);

2) поведінкові (behavioral) моделі:

- діаграми варіантів використання (use case diagrams);
- діаграми взаємодії (interaction diagrams): діаграми послідовності (sequence diagrams) та кооперативні діаграми (collaboration diagrams);
- діаграми станів (statechart diagrams);
- діаграми діяльності (activity diagrams).

UML має механізми розширення, призначені для можливості адаптації мови моделювання розробниками до своїх конкретних потреб, не змінюючи при цьому його метамодель. Наявність механізмів розширення принципово відрізняє UML від таких засобів моделювання, як IDEF0, IDEF1X, IDEF3, DFD та ERM. Перераховані мови є сильно типізованими, оскільки вони не допускають довільної інтерпретації семантики елементів моделей. UML, допускаючи таку інтерпретацію, є слабко типізованою мовою. До його *механізмів розширення* належать:

- стереотипи – новий тип елементу моделі, який визначається на основі вже існуючого елементу; це механізм, який дозволяє розділяти класи на категорії;
- теговані (іменовані) значення – це пара рядків «тег=значення» або «ім'я=вміст», в яких зберігається додаткова інформація про будь-який елемент системи;

- обмеження – це семантичне обмеження, яке має вигляд текстового виразу природньою або формальною мовою, яке неможливо виразити мовою UML.

Моделювання бізнес-процесів є важливою складовою частиною проектів по створенню крупномасштабних систем ПЗ. Відсутність таких моделей є однією з головних причин невдач багатьох проектів. Моделі бізнес-процесів є самостійним результатом з великим практичним значенням. Наразі у моделюванні бізнес-процесів переважає процес ний підхід. Його основний принцип полягає у структуруванні діяльності організації відповідно до її бізнес-процесів, а не організаційно-штатної структури. Процесний підхід може використовувати будь-які з вище перерахованих засобів моделювання. Однак наразі спостерігається тенденція інтеграції різноманітних методів моделювання та аналізу систем, яка проявляється у формі створення інтегрованих засобів моделювання. Одним з таких засобів є продукт під назвою ARIS - Architecture of Integrated Information System – це комплекс засобів аналізу та моделювання діяльності підприємства на основі сукупності різних методів моделювання. ARIS підтримує чотири типи моделей: організаційні моделі (структура системи); функційні моделі (ієрархія цілей системи); інформаційні моделі (структура інформації системи); моделі керування (комплексний погляд на реалізацію бізнес-процесів в межах системи). Під час моделювання кожний аспект діяльності підприємства спочатку розглядається окремо, а після детального припрацювання всіх аспектів будується інтегрована модель, яка відбиває всі зв'язки між різними аспектами. Моделювання бізнес-процесів передбачає побудову двох моделей: модель бізнес-процесів (Business Use Case Model) та модель бізнес-аналізу (Business Analysis Model). Для моделювання бізнес-процесів можуть бути використані діаграми мови UML, але при моделюванні діяльності великої компанії, яка не лише виробляє продукцію, але й надає послуги, необхідно застосовувати різні методики моделювання.

Прототипування ПЗ – це етап розроблення ПЗ, процес створення прототипу програми – макету (чорнової, пробної версії)

програми, як правило, з метою перевірки придатності пропонованих для застосування концепцій, архітектурних та/або інших технологічних рішень, а також для представлення програми замовнику на ранніх етапах процесу розроблення. Прототип дозволяє також одержати зворотній зв'язок від майбутніх користувачів, причому, саме тоді, коли це найбільш необхідно: на початку проекту, коли ще є можливість виправити помилки проектування практично без втрат. Прототипування спрямовано на перевірку концепції та мінімізацію ризиків на етапі розроблення програмного забезпечення.

Цілі прототипування:

1) перевірка концепції та моделювання процесів – прототип дозволяє максимально наблизити бачення майбутньої системи до реального функціонування, включаючи емуляцію робочих процесів з використанням тестових даних, а також оцінити зручність використання; робота з прототипом дозволяє своєчасно скоригувати вимоги до майбутнього ПЗ і передавати у розроблення лише перевірені та ретельно деталізовані завдання;

2) керування інвестиціями та мінімізація ризиків – верифікація концепції та деталізація вимог, яка досягається в процесі створення та оцінювання прототипу, дозволяє мінімізувати ризики при інвестуванні у розроблення ПЗ шляхом завчасного виявлення потенційних «вузьких» місць, точної пріоритезації задач та реалістичного планування бюджетів та термінів. Такий підхід забезпечує коректну реалізацію проектних вимог та ідей, закладених в концепцію майбутньої системи.

Процес створення прототипу складається з таких кроків:

1) визначення початкових вимог;

2) розроблення першого варіанту прототипу, який містить лише інтерфейс користувача системи;

3) вивчення прототипу замовником та кінцевими користувачами, одержання зворотнього зв'язку про необхідні зміни та доповнення;

4) перероблення та покращення прототипу: з врахуванням одержаних зауважень та пропозицій змінюються як специфікації, так і прототип; після цього кроки 3 ф 4 можуть повторюватись.

Два основних *типи прототипування*:

- *швидке прототипування* (rapid або throwaway prototyping) – створюється макет, який на певному етапі буде залишений і не стане частиною готової системи. Основна перевага – у швидкості: у відповідь на свої вимоги замовник практично одразу одержує прототип інтерфейсу і одразу може уточнювати свої вимоги. Вартість зміни вимог на цьому етапі дуже низька, оскільки немає коду, який потрібно переписувати. Швидке прототипування виконується в найкоротші терміни, не обов'язково в межах тієї платформи та тих технологій, як і розроблювана система;

- *еволюційне прототипування* (evolutionary prototyping) – має за мету послідовно створювати макети системи, які будуть все ближче й ближче до реального продукту. Перевага такого підходу – на кожному кроці наявна робоча систем, яка нехай і не володіє всією необхідною функційністю, але яка покращується з кожною ітерацією. При цьому немає витрат на код, який не буде використовуватись. Еволюційний підхід до прототипування обирається, якщо всі необхідні вимоги до моменту початку розроблення невідомі і будуть визначатись по мірі створення програми, тоді на кожному етапі реалізуються наявні та ясні вимоги. В деяких випадках, коли розробляється новітня система, аналогів якої немає, користувачі починають використовувати систему ще до того, як вона буде повністю дописана, адже система з неповною функційністю краще, ніж її повна відсутність.

Переваги застосування прототипування:

- зменшення часу, вартості, ризиків: прототипування покарщує якість специфікацій; чим пізніше проводяться зміни у специфікації, тим вони дорожчі, тому уточнення «чого ж замоники хотять насправді» на ранніх стадіях розроблення знижує загальну вартість;

- залучення користувача у процес розроблення: прототипування залучає майбутніх користувачів до процесу розроблення і дозволяє їм бачити те, як саме виглядатиме майбутня програма, що дозволяє позбавитись від можливих розбіжностей в уявленні про програму між розробниками та користувачами.

Недоліки застосування прототипування:

- недостатній аналіз: концентрація зусиль на обмеженому прототипі може відволікати розробників від необхідного аналізу вимог на повну систему;

- змішування прототипу та готової системи в уявленні користувачів: користувачі можуть подумати, що прототип, який пропонується відкинути і є основою майбутньої системи, тому можуть вимагати від прототипу більш точної поведінки або розчаруватись у можливостях розробників;

- надмірний час на створення прототипу: ключова властивість прототипу – короткий час його створення; якщо ж розробники не приймають це до уваги, то вони витрачають час на створення надто складного прототипу та втрачають переваги від застосування прототипування взагалі.

Спірним є питання, чи застосовне прототипування взагалі, у тій чи іншій формі, до всіх типів проектів. Однак відомо, що найбільші переваги прототипування дає при розробленні систем, які мають розвинутий інтерфейс користувача. Інакше прототипування майже не дає реальних переваг.

Варіанти використання прототипів:

- як інструмент видобування, перевірки та затвердження вимог;

- як основу для написання специфікації вимог до ПЗ (SRS) та технічного завдання на етапі проектування;

- як техніку перевірки програмного дизайну на етапі проектування;

- як об’єкт дослідження юзабіліті-тестування (тестування на зручність використання);

- як зразок для розробників на етапі реалізації (конструювання);

- як зразок системи на етапі комерційної пропозиції;

- як зразок при тестуванні готового ПЗ;

- як зразок при прийманні-передачі роботи;

- як приклад рішення для демонстрації потенційним замовникам.

Висновки

За прогнозами IDC, ринок ТППЗ, який зазнав певної кризи у 2002 році, в найближчі 5 років очікує стійке зростання в середньому на 6,3% на рік. Визначальним фактором для розвитку цієї тенденції є прагнення компаній-розробників підвищити продуктивність своєї роботи, скоротити терміни виходу нових продуктів на ринок, контролювати витрати та швидко отримувати віддачу від інвестицій. Досягненню цих цілей сприяє використання середовищ розроблення, які дозволяють знизити складність процесів створення ПЗ, збільшити їх ефективність, зменшити витрати на розроблення та максимально використовувати потенціал нових технологій. Аналітики мають думку, що основний напрямок розвитку інструментальних засобів – це їх наскрізна інтеграція, перехід від частково інтегрованих засобів до інтегрованих комплексів, які поєднують можливості керування вимогами, моделювання, розроблення, тестування, керування конфігурацією та змінами, а також розгортання додатків. Найближчим часом такі комплекси включатимуть і засоби керування потоками робіт та проектами. Ринок таких інструментальних засобів очікує глобальна консолідація, яка принесе значні вигоди розробникам. В той же час проблема обгрунтованого вибору та ефективного застосування ТППЗ у великомасштабних проектах залишається актуальною. Неможливо досягти задовільних результатів від застосування навіть найдосконаліших технологій, якщо вони застосовуються безсистемно, розробники не мають необхідної кваліфікації для роботи із ними, і сам проект виконується і керується хаотично. Систематичний, обгрунтований підхід до вибору та застосуванню ТППЗ може скоротити час і підвщити якість розроблення ПЗ, забезпечити високий ступінь його незалежності від конкретних розробників, а також знизити витрати на розроблення та супровід програмного забезпечення.

Лекція №5
ОСНОВНІ ФАЗИ, СТАНДАРТИ ТА ЗАСОБИ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Основні фази розроблення ПЗ: формулювання вимог, формулювання цілей проекту, аналіз прикладної галузі, створення функційної специфікації, проектування, реалізація
2. Стандарти в галузі розроблення ПЗ
3. Програмні засоби підтримки життєвого циклу

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.
2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.
4. А.М. Вендров. Современные технологии создания программного обеспечения // [Електронний ресурс] – Режим доступу: http://citforum.ru/programming/application/program/
5. Джек Гринфилд, Кит Шорт, Стив Кук, Стюарт Кент, Джон Крупи Фабрики разработки программ (Software Factories): потоковая сборка типовых приложений, моделирование, структуры и инструменты — М.: «Диалектика», 2006
6. Д.А.Чернев. Технология разработки програмного обеспечения – Ташкент, 2004 – 224 с.
7. IEEE 1209-1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools.
8. IEEE 1348-1995. IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools.
9. ISO/IEC 14102:1995(E). Information technology - Guideline for the evaluation and selection of CASE Tools.

Вступ

Методології, технології та інструментальні засоби проектування ПЗ складають основу проекту будь-якої програмної системи. Методологія реалізується через конкретні технології та стандарти, які їх підтримують, а також через методики та інструментальні засоби, які забезпечують виконання процесів ЖЦ.

Застосування будь-якої технології проектування, розроблення та супроводу ПЗ в конкретній організації і конкретному проекті неможливе без відпрацювання ряду стандартів (правил, угод), яких повинні дотримуватись всі учасники проекту. До таких стандартів належать наступні: 1) стандарт проектування; 2) стандарт оформлення проектної документації; 3) стандарт інтерфейсу користувача.

Сучасні CASE-засоби охоплюють обширну галузь підтримки багатьох ТППЗ: від простих засобів аналізу та документування до повномасштабних засобів автоматизації, які покривають весь життєвий цикл ПЗ. Найбільш трудомісткими етапами розроблення ПЗ є етапи аналізу та проектування, в процесі яких CASE-засоби повинні забезпечувати якість прийнятих технічних рішень та підготовку проектної документації. До CASE-засобів відносять програмні засоби, які автоматизують ту чи іншу сукупність процесів ЖЦ ПЗ.

1. Основні фази розроблення ПЗ: формулювання вимог, формулювання цілей проекту, аналіз прикладної галузі, створення функціональної специфікації, проектування, реалізація

Формулювання вимог до ПЗ є однією з найважливіших фаз розроблення ПЗ, оскільки визначає успіх всього проекту. Дана фаза включає етапи:

1) планування робіт, яке попереджає роботи над проектом. Основні задачі:

- визначення цілей розробки;
- попередня економічна оцінка проекту;
- побудова план-графіку виконання робіт;
- створення та навчання сумісної робочої групи;

2) проведення обстеження діяльності автоматизваного об'єкта (організації), в межах якого здійснюється:

- попереднє виявлення вимог до майбутньої системи;
- визначення структури організації;
- визначення переліку цільових функцій організації;
- аналіз розподілу функцій за підрозділами та співробітниками;
- виявлення функційних взаємодій між підрозділами, інформаційних потоків всередині підрозділу і між ними, зовнішніх по відношенню до організації об'єктів та зовнішніх інформаційних взаємодій;
- аналіз існуючих засобів автоматизації діяльності;

3) побудова моделей діяльності організацій, яка передбачає обробку матеріалів обстеження та побудову двох видів моделей:

- модель "AS-IS" («як є»), яка відбиває існуючий на момент обстеження стан справ в організації та дозволяє зрозуміти, яким чином функціонує дана організація, а також виявити «вузькі» місця і сформулювати пропозиції щодо покращення ситуації;
- модель "TO-BE" («як має бути»), яка відбиває уявлення про нові технології роботи організації.

Кожна модель включає в себе повну функцій ну та інформаційну модель діяльності організації, а також, у випадку необхідності, модель, яка описує динаміку поведінки організації.

Перехід від моделі "AS-IS" до моделі "TO-BE" може виконуватись двома способами: 1) вдосконаленням існуючих технологій на основі оцінювання їх ефективності; 2) радикальною зміною технологій та перепроектуванням бізнес-процесів (реінжиніринг бізнес-процесів).

Побудовані моделі мають самостійне практичне значення. Наприклад, модель "AS-IS" дозволяє виявити «вузькі» місця в існуючих технологіях та пропонувати рекомендації по вирішенню проблем незалежно від того, пропонується на даному етапі подальше розроблення ПЗ чи ні. Крім того, модель полегшує навчання співробітників конкретним напрямкам діяльності організації за рахунок використання наочних діаграм («один малюнок вартий тисячі слів»).

Формулювання цілей проекту – під час постановки цілей програмного проекту варто розрізняти мету – створення ПЗ і мету, заради досягнення якої створюється ПЗ. Дуже рідко ПЗ створюється «заради самого себе». Як виключення можна вказати лише навчальні проекти, коли ПЗ розробляється заради демонстрації можливостей розробників, або ж академічні проекти, коли ПЗ розробляється заради доказу принципової можливості розроблення. Але в більшості випадків програмні продукти розробляються з метою наступного використання, тому дуже важливо не випустити з поля зору основну мету розроблення, яка формулюється замовниками і користувачами ПЗ, і не замінити цю основну мету власними цілями розробників ПЗ. Сучасне визначення якості ПЗ звучить наступним чином: «якість – це ступінь задоволеності користувачів, або ступінь відповідності ПЗ висунутим до нього вимогам користувачів». Якщо цілі проекту не відповідають потребам користувачів, то програмний продукт неможливо буде визнати якісним, навіть якщо при його розробленні були використані сучасні технології та були задіяні найкваліфікованіші розробники.

Цілі проекту повинні бути описані явно та задовольняти певним критеріям:

1) конкретність – мета проекту повинна бути конкретною, вираженою в термінах проекту, з вказанням умов та термінів. Наприклад, мета «досягти максимального ступеня задоволеності користувачів продукту» не конкретна. В той час, як мета «одержати позитивний відгук на продукт від 90% користувачів за перші півроку продажів» досить конкретна для проекту по розробленню «коробкового» продукту;

2) реалістичність – мета повинна бути реалістичною, тобто досягенною з вархуванням наявних ресурсів. Наприклад, мета «розробити програму автоматичного художнього перекладу поезії з англійської мови на російську за рік» нереалістична для компанії середнього розміру. В той час, як мета «розробити програму підрядкового перекладу технічного тексту з англійської мови на російську за рік» може бути цілком реалістична, якщо є досвід та певні напрацювання;

3) вимірюваність – повинен бути вказаний ефективний критерій, який дозволяв би визначити, чи досягнуто мету і в якому ступені. Наприклад, мета «розробити програму, яка істотно підвищує ефективність процесу продажів» не вимірювана: неясно, як вимірюється ефективність і яке підвищення можна вважати істотною. В той час, як мета «розробити програму, яка прискорює процес оформлення заявки в середньому на 10%» вимірювана;

4) несуперечливість – цілі не повинні бути внутрішньо суперечливими і взаємно виключаючими одна одну. Наприклад, цілі «розробити продукт з максимальним набором функцій» та «витратити мінімальну кількість ресурсів на розроблення» суперечать одна одній (і до того ж, неконкретні). Мінімальна кількість ресурсів (нуль) буде витрачено, якщо розроблення не проводити, але тоді й можливостей у продукта не буде. В той час, як мета «розробити продукт, який має не менше функцій, ніж конкурент X, і витратити на розроблення не більш Y людиногодин» не є суперечливою (але може виявитись нереалістичною).

Сформульовані цілі проекту необхідно підтвердити, тобто заручитись явною згодою з цілями всіх сторін, зацікавлених в успішності проекту. Сторонами, зацікавленими в успішності проекту, можуть бути: 1) розробники, які виконують проект; 2) інвестори, які його фінансують; 3) користувачі, які будуть застосовувати результати на практиці. Зацікавлені сторони проекту – всі особи і організації, які прямо або непрямо приймають участь в проекті та зацікавлені в його успішності. Ясно, що цілі, переслідувані сторонами, можуть бути досить різні. Наприклад, розробники можуть переслідувати мету засвоєння нової технології при виконанні проекту. Інвестори, швидше за все, зацікавлені у поверненні коштів та одержанні прибутку. Користувачі сподіваються, що розроблюване ПЗ зробить їх працю більш продуктивною. Підтвердження цілей проекту – це пошук компромісу, який задовольняв би всі зацікавлені сторони.

В процесі виконання проекту його цілі можуть змінитись. Причини зміни цілей різноманітні і навряд чи можуть бути враховані наперед. Наприклад, причинами змін цілей можуть бути: 1) зміни в бізнес-процесі, для автоматизації якого розробляється

ПЗ; 2) зміни кон'юнктури ринку, зокрема, поява нового конкуруючого продукту; 3) поява нової інформаційної технології, яка впливає на цільову предметну галузь. В процесі виконання проекту постійно повинен проводитись моніторинг цілей для перевірки їх актуальності. У випадку необхідності цілі слід переглядати. Іноді перегляд цілей тягне за собою перегляд плану всього проекту.

Аналіз прикладної галузі. Щоб розробити програмну систему, яка приносить реальні вигоди певним користувачам, необхідно спочатку з'ясувати, які ж задачі вона повинна вирішувати для цих людей і які властивості мати. Вимоги до ПЗ визначають, які властивості та характеристики воно повинно мати для задоволення потреб користувачів та інших зацікавлених осіб. Однак сформулювати вимоги до складної системи не так просто. В більшості випадків майбутні користувачі можуть перерахувати набір властивостей, який вони хотіли б бачити, але ніхто не дасть жодних гарантій, що це – вичерпний список. Крім того, часто саме формулювання цих властивостей буде незрозумілим більшості програмістів. Щоб ПЗ було дійсно корисним, важливо, щоб воно задовольняло реальні потреби людей та організацій, які часто відрізняються від безпосередньо виражених користувачами побажань. Для виявлення цих потреб, а також для виявлення змісту висказаних вимог доводиться проводити досить велику додаткову роботу, яка називається аналізом предметної галузі або бізнес-моделюванням, якщо мова йде про потреби організації.

В результаті цієї діяльності розробники повинні навчитись розуміти мову користувачів та замовників, з'ясувати цілі їх діяльності, визначити набір задач, вирішуваних ними. На додаток варто з'ясувати, які взагалі задачі потрібно вміти вирішувати для досягнення цих цілей, з'ясувати властивості результатів, які хотілось би одержати, а також визначити набір сутностей, з якими доводиться мати справу при вирішенні цих задач. Крім того, аналіз предметної галузі дозволяє виявити місця можливих покращень і оцінити наслідки прийнятих рішень про реалізацію тих чи інших функцій. Після цього можна визначати галузь відповідальності майбутньої програмної системи і точніше формулювати вимоги.

Аналізом прикладної галузі займаються системні аналітики або бізнес-аналітики, які передають одержані ними знання іншим членам проектної команди, формулюючи їх більш зрозумілою розробникам мовою. Для передачі цих знань існує деякий набір моделей у вигляд графічних схем і тектосвих документів. Аналіз діяльності великої організації дає величезні обсяги інформації, з якої потрібно обирати суттєву та знаходити у ній «пробіли». Отже, одержану інформацію слід якимось чином систематизувати, для чого застосовується схема Захмана або архітектурна схема підприємства.

В основі схеми Захмана лежить наступна ідея: діяльність навіть дуже великої організації можна описати, використовуючи відповіді на прості запитання – навіщо, хто, як, де і коли – і різні рівні розгляду:

- цілі організації та базові правила, за якими вона працює;
- персонал, підрозділи та інші елементи організаційної структури, зв'язки між ними;
- сутності та дані, з якими має справу організація;
- виконувані організацією та різними її підрозділами функції та операції над даними;
- географічний розподіл елементів організації та зв'язки між географічно розділеними її частинами;
- часові характеристики та обмеження на діяльність організації, значущі для її діяльності події.

Також виділені декілька рівнів розгляду, з яких при бізнес-моделюванні особливо важливі три верхніх:

- найкрупніший – рівень організації в цілому, розглянутий в її розвитку спільно з оточенням, рівень загального планування її діяльності. Цей рівень містить тривалі цілі і задачі організації як цілісної системи, основні зв'язки організації з зовнішнім світом та основні види її діяльності;
- рівень бізнесу, на якому організація розглядається в усіх аспектах як окрема сутність, яка має певну структуру, відповідає її основним задачам;
- системний рівень, на якому визначаються концептуальні моделі всіх аспектів організації, без прив'язки до конкретних її

втілень та реалізацій, наприклад, логічна модель даних у вигляді набору сутностей та зв'язків між нимим, логічна архітектура системи автоматизації у вигляді набору вузлів з прив'язаними до них функціями.

Найбільш зручною формою представлення інформації при аналізі предметної області є графічні діаграми різного роду. Вони дозволяють досить швидко зафіксувати одержані знання, швидко відновлювати їх у пам'яті і успішно спілкуватись із замовниками та іншими зацікавленими особами.

Часто для опису поведінки складних систем та діяльності крупних організацій використовуються діаграми потоків даних, які містять 4 види графічних елементів: процеси (трансформації даних в межах описуваної системи), сховища даних (зовнішні по відношенню до системи), сутності та потоки даних (між елементами трьох попередніх видів).

Використовуються декілька систем позначень для перерахованих елементів, найбільш відомі нотація Йордана-ДеМарко (Yourdon-DeMarco) і нотація Гейна-Сарсона (Gane-Sarson). В нотації Йордана-ДеМарко: процеси зображені колами, зовнішні сутності – прямокутниками, сховища даних – двома горизонтальними паралельними лініями. В нотації Гейна-Сарсона: процеси – прямокутники з заокругленими кутами, зовнішні сутності – прямокутники з тінню, сховища даних – витягнуті горизонтально прямокутники без правого ребра.

Фаза створення функційної специфікації. Функційна специфікація (Functional Specification) – це формальний опис, який пояснює, що і як буде робити програма. Вона достатньо детально показує будову всіх модулів та їх взаємодію з врахуванням проектних обмежень. Функційна специфікація – це документ, який описує потрібні характеристики програмної системи (функційність), а також необхідні для користувача системи вхідні та вихідні параметри.

Функційні специфікації допомагають усунути дублювання та невідповідності, дозволяють точно оцінити необхідні дії та ресурси, виступають в якості узгоджуючого та довідкового документів про внесені зміни, надають документацію про

конфігурацію, дають можливість взаємодії осіб, які працюють з вісьмома основними функціями системного проектування. Вони дають точне уявлення про вирішення проблеми, підвищуючи ефективність розроблення системи та оцінюючи вартість альтернативних шляхів проектування. Вони служать вказівкою для випробувачів при верифікації (якісному оцінюванні) кожної технічної вимоги.

Функційна специфікація не визначає операцій, які відбуваються всередині даної системи і яким чином буде реалізовано її функцію. Замість цього, вона розглядає взаємодію із зовнішніми агентами (персонал, периферійні пристрої), які можуть взаємодіяти із системою.

Приклад з типової функційної специфікації: «Коли користувач натискає кнопку ОК, діалогове вікно закривається і в фокусі залишається головне вікно, яке було відкрите до появи діалогу». Така вимога описує взаємодію зовнішнього агента (користувач) і програмної системи.

Специфікація може бути неформальною, тоді її можна сприймати як план або керівництво користувача з точки зору розробника, або формальною, тоді вона визначає математичні або програмні умови.

З врахуванням призначення функційної специфікації та важких наслідків помилок у цьому документі, функційна специфікація повинна бути математично точною. Це означає, що вона повинна базуватись на поняттях, побудованих як математичні об’єкти, та твердженнях, однозначно зрозумілих для розробника ПЗ. Досить часто специфікація формулюється природньою мовою. Але використання математичних методів та формалізованих мов при розробленні функційної специфікації дуже бажане.

Функційна специфікація складається з 3-х частин:

1) опис зовнішнього інформаційного середовища, до якого повинні застосовуватись програми розроблюваної системи – на концептуальному рівні визначаються всі використовувані канали введення-виведення, всі інформаційні об’єкти, до яких застосовується розроблюване ПЗ, а також істотні зв’язки між цими

об’єктами (наприклад, концептуальна схема бази даних або опис мережі приладів, якими керує розроблюване ПЗ);

2) визначення функцій програмної системи, визначених на множині станів цього інформаційного середовищі (зовнішні функції системи) – вводяться позначення всіх визначених функцій, специфікуються всі вхідні дані та результати виконання кожної визначеної функції з їх типами, обмеженнями та співвідношеннями, а також визначається семантика кожної функції, що є найбільш важкою задачею;

3) опис небажаних (виключних) ситуацій, які можуть виникнути при виконання програм системи, а також реакцій на ці ситуації, які повинні забезпечити відповідні програми – перераховуються всі істотні з точки зору зовнішнього користувача випадки, коли ПЗ не зможе нормально виконувати ту чи іншу свою функцію, а також визначає реакцію ПЗ на кожен такий випадок.

Функційні специфікації можуть створюватись з різними цілями. Одна з основних цілей – привести групу розробників до єдиної думки про те, як в результаті повинна виглядати програма, перш ніж вдаватись до дій, які потребують значних часових витрат (написання сирцевого коду, тестування, відлагодження програм). Як правило, консенсус досягається після одного або декількох оцінювань економічно ефективних шляхів досягнення технічних вимог, які потрібні для виконання програмного продукту.

У промисловому впорядкованому процесі розроблення ПЗ функційна специфікація описує об’єкт, який повинен бути розроблений. Цей документ з системи специфікацій описує, як будуть реалізовані функції з використанням обраного програмного середовища. В непромислових, прототипних системах розроблення, функцій на специфікація пишеться після або як частина аналізу вимог. Функційна специфікація використовується на етапі тестування ПЗ як еталон, оскільки під час тестування здійснюється порівняння дій програми з очікуваними, визначеними у специфікації.

Специфікація неможлива без чіткого опису структур даних програми. Функційна специфікація розробляється під керівництвом та за безпосередньої участі архітектора (бізнес-аналітика) проекту.

Функційна специфікація може сильно змінюватись від проекта до проекта. В великих комплексних проектах специфікації мають декілька рівнів деталізації. Першоджерелом для розроблення функцій них специфікацій є технічне завдання (Customer Requirements Specification), в якому описуються всі вимоги до ПЗ.

На верхньому рівні специфікування ПЗ розробляється зовнішня специфікація для замовника (Software Specification Document) – документ, який в бізнес-термінах, бкз перевантаження технічними подробицями, пояснює, що повинна робити система.

На другому рівні може бути розроблений концептуальний документ, який описує архітектуру системи (Software Architecture Document) – документ, який відображає функціювання всієї системи в цілому, не деталізуючи побудови окремих модулів, представляє структуру об'єктів та їх залежності.

На заключному етапі розробляється технічна специфікація (Detail Design Document) – документ, який дозволяє повністю зосередитись на етапі кодування (реалізації) системи, описує низькорівневу організацію продукту, всі вимоги для кожного модуля (передавані параметри, глобальні структури та змінні, підпрограми, що викликаються і т.і.).

Задача розроблення специфікації вважається виконаною, якщо користувач чітко сформулював свої очікування до результуючого продукту, а програміст здатен однозначно реалізувати ці очікування в продукті без будь-яких інших знань про проект, керуючись лише специфікацією.

Фазу проектування ПЗ розглядають як діяльність, результат якої складається з двох складових частин:

1) архітектурне проектування - опис високорівневої структури та організації компонентів ПЗ (software architectural design, top-level design);

2) деталізація архітектури - деталізований опис кожного компонентами в потрібному для конструювання обсязі (software detailed design).

Отже, фаза проектування включає наступні 2 етапи:

1) розроблення системного проекту – на цьому етапі формується відповідь на запитання «Що повинна робити майбутня

програмна система?», а саме: визначаються архітектура системи, її функції, зовнішні умови функціювання, інтерфейси та розподіл функцій між користувачами та системою, вимоги до програмних та інформаційних компонент, склад виконавців та терміни розроблення. Основу системного проекту складають моделі проектованої програмної системи, які будуються на основі моделі “TO-BE”. Документальним результатом є технічне завдання;

2) розроблення технічного проекту – на цьому етапі на основі системного проекту здійснюється власне проектування системи, яке включає проектування архітектури системи та детальне проектування. Таким чином, формується відповідь на запитання «Як побудувати систему, щоб вона задовольняла висунуті до неї вимоги?». Моделі проектованої програмної системи при йому уточняються і деталізуються до необхідного рівня.

Архітектура ПЗ - це опис підсистем та компонентів ПЗ, а також зв'язків між ними. Архітектура намагається визначити внутрішню структуру розроблюваного ПЗ, задаючи спосіб, яким ПЗ організовується або конструюється.

Цілі етапу проектування:

1) підвищення продуктивності бізнес-процесів;

2) зменшення витрат;

3) покращення операційної бізнес-діяльності;

4) підвищення ефективності керування;

5) зменшення ризиків;

6) підвищення ефективності IT-організації;

7) підвищення продуктивності роботи користувачів;

8) підвищення інтероперабельності (можливості та прозорості взаємодії);

9) зменшення вартості підтримки життєвого циклу ПЗ;

10) покращення характеристик безпеки;

11) підвищення керованості.

Принципи проектування:

1) абстракція (Abstraction) - результатом є модель, яка спрощує поставлену проблему до рамок, значущих у даному контексті;

2) зв'язність та зчеплення (Coupling and Cohesion): зв'язність визначає силу зв'язку та взаємовпливу між модулями, тобто може розглядатись як ступінь самодостатності модуля; зчеплення визначає силу зв'язків всередині модуля (внутрішніх зв'язків), тобто функціональну залежність;

3) декомпозиція та розбиття на модулі (Decomposition and Modularization) - проводиться з метою одержання більш дрібних та відносно незалежних програмних компонентів, кожен з яких має різну функціональність;

4) інкапсуляція/приховування інформації (Encapsulation/information hiding) - групування та упаковування елементів та внутрішніх деталей абстракції по відношенню до реалізації з метою недоступності цих деталей користувачам компонентів;

5) розділення інтерфейсу та реалізації (Separation of interface and implementation) - визначення компоненту через вивільнення інтерфейсу від безпосередніх деталей реалізації;

6) достатність, повнота та простота (Sufficiency, completeness and primitiviness) моделі або проекту ПЗ.

Фундаментальними та ключовими проблемами проектування є:

1) паралелізм (Concurrency) - питання, піходи та методи організації процесів, задач і потоків для забезпечення ефективності, атомарності, синхронізації та розроділу в часі оброблення інформації;

2) контроль та оброблення подій (Control and Handling of Events) - методи оброблення подій, часто реалізованих як функції зворотнього виклику;

3) розподіл компонентів (Distribution of Components) - розподіл (і виконання) оброблення в термінах апаратного забезпечення, мережевої інфраструктури і т.і., використання проміжного зв'язуючого ПЗ (middleware);

4) оброблення помилок та виключних ситуацій, забезпечення відмовостійкості (Errors and Exception Handling and Fault Tolerance) - як попередити збої, або, якщо збій все ж відбувся, як забезпечити подальше функціонування ПЗ;

5) взаємодія та представлення (Interaction and Presentation) - представлення інформації користувачам та взаємодії користувачів з ПЗ з точки зору реакції ПЗ на дії користувачів;

6) збережуваність даних (Data Persistence) - вирішується питання, як повинні оброблятись "довгоживучі" дані.

Термінологічний розподіл різних видів проектування ПЗ:

1) D-дизайн (D-design, decomposition design) – декомпозиці структури ПЗ у вигляді набору фрагментів або компонентів;

2) FP-дизайн (FP-design, family pattern design) – сімейство архітектурних представлень, котрі базуються на шаблонах;

3) I-дизайн (I-design, invention) – створення високорівневої концепції представлення того, що собою являє програмна система.

Існує ряд атрибутів, які допомагають оцінити та досягти якісного виконання етапу проектування. Ці атрибути описують характеристики програмного забезпечення та елементів архітектури - тестованість, можливість переносу, модифікованість, продуктивність, безпека і т.і. Дані атрибути стосуються лише результатів етапу проектування (архітектури), але не самого процесу проектування. Взагалі, обговорювані атрибути можна розбити на декілька груп:

1) застосовувані до часу виконання ПЗ (середній час відгуку ПЗ, який дозволяє оцінити якість результатів проектування з точки зору продуктивності);

2) орієнтовані на час етапу проетування, тобто ті, які дозволяють оцінювати якість одержуваного проекту ще на етапі проектування (середнє навантаження класів бізнес-методами);

3) атрибути якості архітектури (конецптуальна цілісність проекту, його непротиріччя, повнота, завершеність).

У програмній індустрії поширені методики, які допомагають досягти якісного проекту: 1) огляд проекту (software design review); 2) статичний аналіз (static analysis); 3) імітаційне моделювання та прототипування (simulation and prototyping).

Для швидкого одержання працюючих прототипів додатків використовуються відповідні інструментальні засоби (CASE–засоби).

На фазі проектування частина користувачів приймають участь в технічному проектуванні системи під керівництвом фахівців-розробників. Користувачі при взаємодії із розробниками уточнюють і доповнюють вимоги до системи, які не були вяивлені на попередній фазі: більш детально розглядаються процеси системи; для елементарних процесів складається частковий прототип, який усуває неясності та неоднозначності; встановлюються вимоги розмежування доступу до даних; визначається склад необхідної документації.

Результатом фази проектування повинні бути:

1) загальна інформаційна модель системи;

2) функціональні моделі системи в цілому та підсистем, реалізованих окремими командами розробників;

3) точно визначені інтерфейси між автономно розроблюваними додатками;

4) побудовані прототипи екранних форм, звітів, діалогів.

Фаза проектування базується на методології, яка основними завданнями цього етапу розглядає:

1) відображення вимог середовища функціонування і розроблення ПЗ;

2) визначення всіх конструкцій як композицій компонентів;

3) прив'язку проекту до технічних особливостей платформи реалізації, СУБД, організації комунікацій, наявності фактора реального часу;

4) закладення низки вимог по якості, вимог до структури ПЗ, вимог до навігації по ПЗ, вимог до дизайну інтерфейсів користувача, вимог до мультимедіа-компонентів ПЗ, вимог по зручності (usability) та інших технічних вимог.

Але існуючі методології виконання етапу проектування не враховують можливості раннього оцінювання або прогнозування таких найважливіших характеристик проекту і розроблюваного ПЗ, як надійність, складність та якість, хоча питання забезпечення якості взагалі є значущими для всіх етапів життєвого циклу.

Фаза реалізації ПЗ - детальне створення робочого ПЗ комбінуванням кодування, верифікації, модульного тестування, інтеграційного тестування та налагодження. Процес реалізації ПЗ

пов'язаний із важливими аспектами діяльності по проектуванню і тестуванню - відштовхується від результатів проектування, а з його результатами працює тестування. У процесі реалізації створюється більша частина активів програмного проекту - конфігураційних елементів.

Фундаментальні основи реалізації ПЗ включають:

1) мінімізацію складності (Minimizing Complexity) - потреба у зменшенні складності впливає на всі аспекти реалізації і особливо критична для процесів верифікації та тестування результатів конструювання. Зменшення складності під час реалізації ПЗ досягається шляхом приділення особливої уваги створенню простого, читабельного коду. Це не означає, що слід відмовитись від застосування певних розвинутих можливостей мови програмування, а означає лише надання більшої значущості читабельності коду, простоті тестування, прийнятному рівню продуктивності, задоволенню заданих критеріїв без огляду на терміни, функціональність та інші обмеження проекту;

2) очікування змін (Anticipating Changes) - одна з рушійних сил реалізаціїя;

3) реалізація з можливістю перевірки (Constructing for Verification) - передбачається, що реалізація ПЗ повинна проводитись таким чином, щоб саме ПЗ допомагало вести пошук причин збоїв, було прозорим для застосування різних методів перевірки та внесення необхідних змін. Для досягнення такого результату використовуються наступні методики: огляд коду, модульне тестування, структурування коду для можливості застосування автоматизованих засобів тестування, обмежене застосування складних або важких для розуміння мовних конструкцій;

4) стандарти реалізації (Standards in Constructing) - зовнішні та внутрішні стандарти, пов'язані із мовами програмування, використовуваним інструментальним забезпеченням, технічними інтерфейсами, взаємним впливом, питаннями очікування та оброблення змін та ризиків, мінімізацією складності, питаннями конструювання для перевірки.

Більша частина результатів та процес реалізації можуть бути виміряні, в тому числі кількісно. Ці вимірювання, названі результатами аудиту коду або метриками коду, корисні як для оцінювання ризиків та якості, так і для вибору операцій по зниженню ризиків і підвищенню якості, а також для керування конструюванням та програмними проектами в цілому.

Окрему увагу слід приділити повторному використанню у ПЗ (Reuse). Реалізація повторного використання ПЗ передбачає і тягне за собою дещо більше, ніж просто створення та використання бібліотек активів. Воно вимагає формалізації практики повторного використання на основі інтеграції процесів та діяльності з повторного використання у життєвий цикл ПЗ. Повторне використання досить важливе безпосередньо при реалізації ПЗ. Задачі, пов'язані з повторним використанням в процесі реалізації: вибір модулів та тестів, які підлягають повторному використанню; оцінювання потенціалу повторного використання коду та тестів; відстежування інформації та створення звітності з повторного використання.

Існує ряд методик, призначених для забезпечення якості коду, виконуваних в ході його реалізації. Вибір та використання конкретних методик часто диктується стандартами, а також залежать від досвіду фахівців, які займаються реалізацією коду. Основі методики забезпечення якості в процесі реалізації:

1) модульне та інтеграційне тестування;

2) розроблення з первинністю тестів (test-first development - тести розробляються до конструювання);

3) покрокове кодування;

4) використання процедур затверджень (assertion);

5) налагодження (debugging);

6) технічні огляди та оцінки (review);

7) статичний аналіз.

Діяльність із забезпечення якості на етапі реалізації відірзняється від операцій із забезпечення якості на інших етапах ЖЦ ПЗ. Основна відмінність полягає у концентруванні уваги на програмному коді та інших артефактах, тісно пов'язаних з кодом, зокрема, на детальних моделях.

2. Стандарти в галузі розроблення ПЗ

Основні стандарти в галузі розроблення ПЗ:

ISO 5806:1984 Обработка информации. Спецификация таблиц и решений с единственно выраженным ответом

ISO/IEC 8631:1989 Информационные технологии. Программные структуры и условные обозначения для их представления

ISO/IEC TR 9126-2:2003 Программирование. Качество продукта. Часть 2. Внешние показатели

ISO/IEC TR 9126-3:2003 Программирование. Качество продукта. Часть 3. Внутренние показатели

ISO/IEC TR 9126-4:2004 Программирование. Качество продукта. Часть 4. Качество при использовании показателей

ISO 9127:1988 Системы обработки информации. Документация пользователя и информация на упаковке для потребительских программных пакетов

ISO/IEC 10746:1998 Информационные технологии. Открытая распределенная обработка. Эталонная модель. Часть 1. Обзор Часть 2. Основы Часть 3. Архитектура Часть 4. Архитектурная семантика

ISO/IEC 11411:1995 Информационные технологии. Представление перехода состояний программного обеспечения для конечного пользователя

ISO/IEC TR 12182:1998 Информационные технологии. Классификация программного обеспечения

ISO/IEC 12207:2008 Информационные технологии. Процессы жизненного цикла программного обеспечения

ISO/IEC 13235-1:1998 Информационные технологии. Открытая распределенная обработка. Часть 1. Спецификация

ISO/IEC 14102:2008 Информационные технологии. Руководство по оцениванию и выбору инструментальных CASE-средств

ISO/IEC 14143:2007 Информационные технологии. Оценка программного обеспечения. Измерение функционального размера.

ISO/IEC TR 14471:2007 Информационные технологии. Программирование. Руководящие положения по принятию средств автоматизированной разработки программного обеспечения

ISO/IEC 14598:1998 Информационные технологии. Оценка программного продукта.

ISO/IEC 14756:1999 Информационные технологии. Измерение и оценка эксплуатационных характеристик автоматизированных систем программного обеспечения

ISO/IEC TR 14759:1999 Разработка программного обеспечения. Макет и прототип. Категоризация моделей макета и прототипа программного обеспечения и их применение ISO/IEC 14764:2006 Разработка программного обеспечения. Процессы жизненного цикла программного обеспечения. Сопровождение

ISO/IEC TR 15026:2010 Проектирование систем и разработка программного обеспечения. Гарантирование систем и программного обеспечения.

ISO/IEC/IEEE 15289:2011 Системная и программная инженерия. Содержание информационных продуктов процесса жизненного цикла систем и программного обеспечения (документация)

ISO/IEC 15504:2004 Информационные технологии. Оценка процессов.

ISO/IEC 15939:2007 Технология программного обеспечения. Процесс измерения

ISO/IEC 15940:2013 Разработка систем и программ. Службы средств поддержки программных разработок

ISO/IEC 16085:2006 Системы и разработка программного обеспечения. Процессы жизненного цикла. Управление рисками

ISO/IEC/IEEE 16326:2009 Разработка систем и программного обеспечения. Процессы жизненного цикла. Управление проектом

ISO/IEC TR 18018:2010 Информационная технология. Разработка и управление системами и программным обеспечением. Руководство по возможностям управления конфигурацией

ISO/IEC 19501:2005 Информационные технологии. Открытая распределительная обработка. Унифицированный язык моделирования (UML).

ISO/IEC TR 19759:2005 Совокупность заний о разработке программного обеспечения. Руководство.

ISO/IEC 19761:2011 Разработка программного обеспечения. COSMIC: Метод измерения функционального размера

ISO/IEC 19770:2012 Информационные технологии. Менеджмент программного обеспечения.

ISO/IEC 19793:2008 Информационные технологии. Открытая распределенная обработка. Использование Унифицированного Языка Моделирования для спецификаций систем распределенной обработки

ISO/IEC 20000:2011 Информационные технологии. Менеджмент услуг.

ISO/IEC 20926:2009 Разработка программного обеспечения и систем. Измерения в программном обеспечении. Метод измерения функционального размера IFPUG 2009

ISO/IEC 20968:2002 Разработка программного обеспечения. Анализ функциональных точек Mk II. Руководство по практике подсчета

ISO/IEC 23026:2006 Разработка программного обеспечения. Рекомендуемая практика для Интернета. Разработка веб-сайтов, администрирование веб-сайтов и жизненный цикл веб-сайтов

ISO/IEC TR 24748:2010 Разработка систем программного обеспечения. Менеджмент жизненного цикла.

ISO/IEC TR 24766:2009 Информационные технологии. Разработка систем и программного обеспечения. Руководство по требованиям к возможностям инженерного иструмента

ISO/IEC TR 24774:2010 Разработка систем и программного обеспечения. Менеджмент жизненного цикла. Руководящие указания по описанию процесса

ISO/IEC 25000:2005 Технология программного обеспечения. Требования и оценка качества программного продукта. Руководство

ISO/IEC 25001:2007 Программирование. Требования к качеству программного продукта и его оценка. Планирование и менеджмент

ISO/IEC 25010:2011 Проектирование систем и разработка программного обеспечения. Требования к качеству систем и программного обеспечения и их оценка (SQuaRE). Модели качества систем и программного обеспечения

ISO/IEC 25012:2008 Программотехника. Требования к качеству и оценка (SQuaRE) программного продукта. Модель качества данных

ISO/IEC 25020:2007 Разработка программного обеспечения. Требования к качеству и оценка качества программного продукта. Измерительная эталонная модель и руководство ISO/IEC 25021:2012 Разработка систем и программ. Требования к качеству систем и программ и их оценка. Элементы показателя качества

ISO/IEC 25030:2007 Разработка программного обеспечения. Требования к качеству и оценка качества программного продукта. Требования к качеству

ISO/IEC 25040:2011 Проектирование систем и разработка программного обеспечения. Требования к качеству систем и программного обеспечения и их оценка (SQuaRE). Процесс оценки

ISO/IEC 25041:2012 Разработка систем и программ. Требования и оценивание качества систем и программ. Руководство по оцениванию для разработчиков, покупателей и независимых оценщиков

ISO/IEC 25045:2010 Разработка систем и программного обеспечения. Требования к качеству и оценка качества систем и программного обеспечения. Модуль оценки восстанавливаемости

ISO/IEC 25051:2006 Технология программного обеспечения. Качество программного продукта. Требования и оценка. Требования к качеству коммерческого программного продукта и инструкции по испытанию

ISO/IEC 25062:2006 Техника программного обеспечения. Оценка и требования качества программного изделия. Общий промышленный формат годных к отчету об испытании

ISO/IEC/IEEE 26511:2011 Разработка систем и программ. Требования к управляющим документацией для пользователя

ISO/IEC 26514:2008 Разработка систем и программ. Требования к дизайнерам и разработчикам пользовательской документации

ISO/IEC/IEEE 29119:2013 Software and systems engineering -- Software testing

ISO/IEC/IEEE 29148:2011 Системная и программная инженерия. Процессы жизненного цикла. Разработка требований

ISO/IEC TR 29154:2013 Software engineering -- Guide for the application of ISO/IEC 24773:2008 (Certification of software engineering professionals - Comparison framework)

ISO/IEC/IEEE 31320:2012 Информационные технологии. Языки моделирования.

ISO/IEC/IEEE 42010:2011 Системная и программная инженерия. Описание архитектуры

ISO/IEC 90003:2004 Техника программного обеспечения. Рекомендации по применению ISO 9001:2000 к компьютерному программному обеспечению

ISO/IEC TR 90005:2008 Разработка систем. Руководящие указания по применению ISO 9001 к процессам жизненного цикла систем

Інші стандарти в галузі можна переглянути за посиланням: http://www.iso.org/iso/ru/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45086&published=on

Взаємозв'язок найбільш визнаних і застосовуваних у світі стандартів розробки, менеджменту якості та управління проектами ПЗ представлений на рис.5.1.

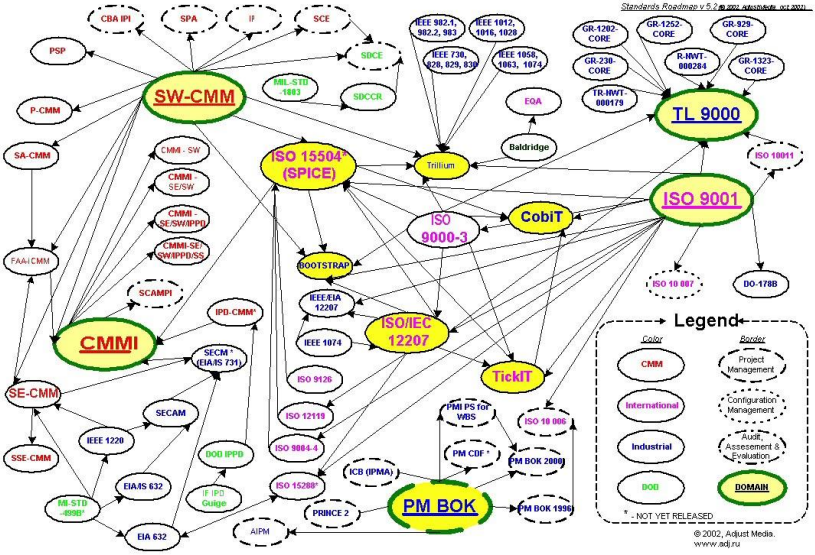


Рис.5.1 - Взаємозв'язок стандартів в галузі розроблення ПЗ

3. Програмні засоби підтримки життєвого циклу

Використання мов моделювання при проектуванні системи необхідно, але якщо робити це вручну, то виникне ряд проблем:

- неадекватна специфікація вимог;
- нездатність виявити помилки у проектних рішеннях;
- низька якість документації;
- тривалий термін розроблення;
- невисока якість тестування.

Таким чином, необхідні програмно-технологічні засоби спеціального класу - *CASE-засоби*, які реалізують CASE-технологію створення та супроводу програмної системи. Термін CASE (Computer Aided Software Engineering) використовується наразі у широкому значенні. Перше значення терміну CASE, обмежене питаннями автоматизації розроблення лише програмного забезпечення, наразі набуло нового змісту, який охоплює процес розроблення складних інформаційних систем в цілому. Тепер під терміном CASE-засоби розуміють програмні засоби, які підтримують процеси створення та супроводу ПЗ, включаючи

аналіз та формулювання вимог, проектування прикладного ПЗ та баз даних, генерацію коду, тестування, документування, забезпечення якості, конфігураційне керування та керування проектом, а також інші процеси. CASE-засоби разом із системним ПЗ та технічними засобами утворюють повне середовище розроблення ПЗ.

CASE-технологія – це методологія проектування ПЗ, а також набір інструментальних засобів, який дозволяє наочно моделювати предметну галузь, аналізувати цю модель на всіх етапах розроблення та супроводу ПЗ та розробляти додатки відповідно до інформаційних потреб користувачів.

Більшість існуючих CASE-засобів засновані на методологіях структурного (в основному) або об'єктно-орієнтованого аналізу та проектування, які використовують специфікації у вигляді діаграм або текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поведінки системи та архітектури програмних засобів. Згідно огляду передових технологій (Survey of Advanced Technology), складеному фірмою Systems Development Inc. На основі анкетування більше 1000 американських фірм, CASE-технологія наразі потрапила до найбільш стабільних інформаційних технологій (її використовували 50% всіх опитаних користувачів більш ніж у третині своїх проектів, з них 85% завершились успішно). Однак, незважаючи на всі потенційні можливості CASE-засобів, існує чимало прикладів їх невдалого впровадження, в результаті яких CASE-засоби стають «полочним» ПЗ (shelfware). В зв'язку з цим необхідно відзначити наступні особливості використання CASE-засобів:

- CASE-засоби не обов'язково дають негайний ефект; він може бути одержаний лише через якийсь час;

- реальні витрати на впровадження CASE-засобів набагато перевищують витрати на їх придбання;

- CASE-засоби забезпечують можливості для одержання істотної вигоди лише після успішного завершення процесу їх впровадження.

Враховуючи різноманітну природу CASE-засобів, було б помилково робити якісь беззаперечні твердження відносно реального задовлення тих чи інших очікувань від їх впровадження. Можна перерахувати наступні фактори, які ускладнюють визначення можливого ефекту від використання CASE-засобів:

- широке розмаїття якості та можливостей CASE-засобів;
- відносно невеликий час використання CASE-засобів у різних організаціях та брак досвіду їх застосування;
- широке розмаїття у практиці впровадження різних організацій;
- відсутність детальних метрик та даних для вже виконаних та поточних проектів;
- широкий діапазон предметних галузей проектів;
- різний ступінь інтеграції CASE-засобів у різних проектах.

Внаслідок цих труднощів доступна інформація про реальні впровадження надто обмежена та суперечлива. Вона залежить від типу засобів, характеристик проектів, рівня супроводу та досвіду користувачів. Деякі аналітики вважають, що реальна вигода від використання деяких типів CASE-засобів може бути одержана лише після одно- або дворічного досвіду. Інші вважають, що вплив може реально проявитись на етапі експлуатації, коли технологічні покращення можуть призвести до зниження експлуатаційних витрат.

Для успішного впровадження CASE-засобів організація повинна мати наступні якості:

- Технологія. Розуміння існуючих можливостей та задатність прийняти нову технологію;
- Культура. Готовність до впровадження нових процесів та взаємозв'язків між розробниками та користувачами;
- Керування. Чітке керівництво та організованість стосовно найбільш важливих етапів та процесів впровадження.

Якщо організація не має хоча б однієї з перерахованих якостей, то впровадження CASE-засобів може завершитись невдачею незалежно від ступеня ретельності дотримання різних рекомендацій з впровадження. Щоб прийняти зважене рішення відносно інвестицій в CASE-технологію, користувачі змушені

проводити оцінку окремих CASE-засобів, спираючись на неповні та суперечливі дані. Ця проблема часто ускладнюється недостатнім знанням всіх можливих «підводних каменів» використання CASE-засобів.

Серед найбільш важливих проблем виділяються наступні:

- достовірна оцінка віддачі від інвестицій у CASE-засоби ускладнена відсутністю прийнятних метрик та даних по проектам та процесам розроблення ПЗ;

- впровадження CASE-засобів може представляти собою достатньо тривалий процес і може не принести негайної віддачі. Можливо навіть короткотривале зниження вартості в результаті зусиль, які витрачаються на впровадження. Внаслідок цбого керівництво організації-користувача може втратити інтерес до CASE-засобів та припинити підтримку їх впровадження;

- відсутність повної відповідності між тими процесами та методами, які підтримуються CASE-засобами, і тими, які використовуються в даній організації, може призвести до додаткових труднощів;

- CASE-засоби часто важко використовувати в комплексі з іншими подібними засобами. Це пояснюється як різними парадигмами, які підтримуються різними засобами, так і проблемами передачі даних та керування від одного засобу до іншого;

- деякі CASE-засоби вимагають надто багато зусиль для того, щоб виправдати їх використання в невеликому проекті, при цьому, тим не менш, можна мати вигоду з тієї дисципліни, до якої зобов’язує їх застосування;

- негативне ставлення персоналу до впровадження нової CASE-технології може бути головною причиною провалу проекта. Користувачі CASE-засобів повинні бути готові до необхідності тривалих витрат на експлуатацію, частої появи нових версій та можливого швидкого морального застарівання засобів, а також постійних витрат на навчання та підвищення кваліфікації персоналу.

Незважаючи на всі вищенаведені перестороги та деякий песимізм, грамотний та розумний підхід до використання CASE-

засобів може подолати всі перераховані труднощі. *Успішне впровадження CASE-засобів повинне забезпечити такі переваги*:

- високий рівень технологічної підтримки процесів розроблення та супроводу ПЗ;

- позитивний вплив на деякі або всі з переахованих факторів: продуктивність, якість продукції, дотримання стандартів, документування;

- прийнятний рівень віддачі від інвестицій у CASE-засоби.

CASE-засоби звичайно служать для підтримки конкретної методології та реалізуючої її технології проектування ПЗ. Можна сказати, що методології та реалізуючі їх технології постачаються в електронному вигляді разом з CASE-засобами і включають бібліотеки процесів, шаблонів, методів, моделей та інших компонентів, призначених для побудови ПЗ того класу систем, на який орієнтовано методологію. Електронні методології включають також засоби, які повинні забезпечувати їх адаптацію для конкретних користувачів та розвиток методології за результатами виконання конкретних проектів. Процес адаптації полягає у видаленні непотрібних процесів, дій ЖЦ та інших компонентів методології, в зміні неприйнятних або у додаванні власних процесів та дій, а також методів, моделей, стандартів та керівництв. Налагодження методології може здійснюватись також і за наступними аспектами: етапи та операції ЖЦ, учасники проекту, використовувані моделі ЖЦ, підтримувані концепції. Електронні методології та технології (а також підтримуючі їх CASE-засоби) складають ядро комплексу узгоджених інструментальних засобів середовища розроблення програмних систем.

Огляд популярних CASE–засобів:

1) CASE–засіб JAM (методологія RAD) – засіб розроблення додатків JAM (JYACC's Application Manager). Основна риса – його відповідність методології RAD, оскільки він дозволяє досить швидко реалізувати цикл розроблення додатку, який полягає у формуванні чергової версії прототипу додатка з врахуванням вимог, виявлених на попередньому кроці, і демонстрації його користувачу. JAM має модульну структуру і складається з наступних компонент: ядро системи (редактор екранів, редактор

меню, набір допоміжних утиліт, засоби виготовлення промислової версії додатку); JAM/DBi – спеціалізовані модулі інтерфейсу до СУБД; JAM/RW – модуль генератора звітів; JAM/CASEi – спеціалізовані модулі інтерфейсу до CASE-засобів; JAM/TPi – спеціалізовані модулі інтерфейсу до менеджерів транзакцій; Jterm – спеціалізований емулятор X-терміналу. JAM реалізує частину функцій підтримки групового розроблення;

2) CASE-засіб Silverrun (методологія DATARUN) - CASE-засіб для підтримки однієї з найбільш поширених в світі електронних методологій DATARUN, згідно з якою ЖЦ ПЗ розбивається на стадії, які зв'язуються з результатами виконання основних процесів, які визначаються стандартом ISO 12207. Кожну стадію крім її результатів повинен завершувати план робіт на наступну стадію. Стадія формулювання вимог та планування включає в себе дії з визначення початкових оцінок обсягу та вартості проекту. Повинні бути сформульовані вимоги та економічне обґрунтування для розроблення ПЗ, функціональні моделі та початкова концептуальна модель даних, які дають підстави для оцінювання технічної реалізованості проекту. Основні результати – моделі діяльності організації, вимоги до системи, початковий бізнес-план. Стадія концептуального проектування починається з детального аналізу первинних даних і уточнення концептуальної моделі даних, після чого проектується архітектура системи. Архітектура включає в себе розподіл концептуальної моделі на очевидні підмоделі. Оцінюється можливість використання існуючого ПЗ та обирається відповідний метод їх перетворення. Після побудови проекту уточнюється початковий бізнес-план. Результуючими компонентами цієї стадії є концептуальна модель даних, модель архітектури системи та уточнений бізнес-план. На стадії специфікації додатків триває процес створення та деталізації проекту. Концептуальна модель даних перетворюється в реляційну модель даних. Визначається структура додатку, необхідні інтерфейси додатку у вигляді екранів, звітів та пакетних процесів разом із логікою їх виклику. В кінці цієї стадії приймається кінцеве рішення про спосіб реалізації додатків. За результатами стадії повинен бути побудований проект ПЗ, який

включає моделі архітектури ПЗ, даних, функцій, інтерфейсів, вимог до розроблюваних додатків, вимог до доопрацювань існуючого ПЗ, вимог до інтеграції додатків, а також сформований кінцевий план створення ПЗ. На стадії розроблення, інтеграції та тестування повинна бути створена тестова база даних, часткові та комплексні тести. Проводиться розроблення, прототипування та тестування баз даних та додатків відповідно до проекту. Відлагоджуються інтерфейси з існуючими системами. Описується конфігурація поточної версії ПЗ. На основі результатів тестування проводиться оптимізація бази даних та додатків. Додатки інтегруються в систему, проводиться тестування додатків у складі системи та випробування системи. Основними результатами стадії є готові додатки, перевірені у складі системи на комплексних тестах, поточний опис конфігурації ПЗ, скоригована за результатами випробувань версія системи та експлуатаційна документація на систему. Стадія впровадження містить дії з встановлення та впровадження баз даних та додатків. Основними результатами стадії повинні бути готова до експлуатації та перенесена на програмно-апаратну платформу замовника версія системи, документація супроводу та акт приймальних випробувань за результатами дослідної експлуатації. Стадії супроводу та розвитку містять процеси та операції, пов’язані з реєстрацією, діагностуванням та локалізацією помилок, внесенням змін та тестуванням, проведенням доопрацювань, тиражуванням та поширенням нових версій ПЗ у місця його експлуатації, переносом додатків на нову платформу та масштабуванням системи. Стадія розвитку фактично є повторною ітерацією стадії розроблення.

Методологія DATARUN спирається на дві моделі: модель організації, модель програмної системи. Ця методологія базується на системному підході до опису діяльності організації. Побудова моделей починається з опису процесів, з яких потім видобуваються первинні дані (стабільна підмножина даних, які організація повинна використовувати для своєї діяльності). Первинні дані описують продукти або послуги організації, виконувані операції та споживані ресурси. До первинних належать дані, які описують зовнішні та внутрішні сутності, а також дані, отримані в результаті

прийняття рішень. Основний принцип DATARUN полягає у тому, що первинні дані, якщо вони певним чином організовані в модель даних, стають основою для проектування архітектури ПЗ. На основі стуркутр первинних даних в модулі Silverrun ERX створюється концептуальна модель даних (ER-модель). На основі моделі бізнес-процесів та концептуальної моделі даних проектується архітектура ПЗ. CASE-засіб Silverrun використовується для аналізу та проектування ПЗ бізнес-класу та орієнтований в більшому ступені на спіральну модель ЖЦ. Він застосовний для підтримки будь-якої методології, заснованої на роздільній побудові функційної та інформаційної моделей (діаграм потоків даних та діаграм «сутність-зв'язок»);

3) об'єктно-орієнтовані CASE-засоби Rational Rose (методологія RUP) – призначені для автоматизації етапів аналізу та проектування ПЗ, а також для генерації кодів різними мовами та випуску проектної документації. Rational Rose використовує синтез-методологію об'єктно-орієнтованого аналізу та проектування, засновану на підходах трьох провідних фахівців в даній галузі (Буча, Рамбо, Джекобсона). Розроблена ними універсальна нотація для моделювання об'єктів (UML - Unified Modeling Language) претендує на роль стандарта в галузі об'єктно-орієнтованого аналізу та проектування. В основі роботи Rational Rose лежить побудова картини системи, яка містить всі діаграми UML, що визначають логічну та фізичну структуру моделі, її статичні та динамічні аспекти. Більш детально цей засіб розглядався у попередній лекції;

4) локальні CASE-засоби (ERwin, BPwin, S-Designor, CASE.Аналитик):

- ERwin – засіб концептуального моделювання БД, який використовує методологію IDEF1X; реалізує проектування схеми БД, генерацію її опису мовою цільової СУБД та реінжиніринг існуючої БД;

- BPwin – засіб функційного моделювання, який реалізує методологію IDEF0;

- S-Designor – засіб для проектування реляційних баз даних; за функційними можливостями та вартістю близький до ERwin;

реалізує стандартну методологію моделювання даних та генерує опис БД для таких СУБД, як ORACLE, Informix, Ingres, Sybase, DB/2, Microsoft SQL Server; виконує реінжиніринг БД для існуючих систем;

- CASE.Аналитик – є практично єдиним наразі конкурентоздатним російським CASE-засобом функційного моделювання та реалізує побудову діаграм потоків даних; дозволяє виконувати побудову та редагування DFD, аналіз діаграм та проектних специфікацій на повноту та суперечливість, одержувати різноманітні звіти за проектом, генерувати макети документів відповідно до вимог ГОСТ 19.XXX та ГОСТ 34.XXX

Висновки

Розроблення ПЗ (англ. software engineering, software development) – це рід діяльності (професія) та процес, спрямований на створення та підтримку робото здатності, якості та надійності ПЗ з використанням технологій, методології та практики з керування проектами, математики, інженерії та інших галузей знань. Глобально процес розроблення ПЗ може бути розділений на декілька процесів - формулювання вимог, формулювання цілей проекту, аналіз прикладної галузі, створення функційної специфікації, проектування, реалізація. Проектування ПЗ наразі ведеться засобами автоматизованого розроблення ПЗ (CASE-засобами). Всі процеси розроблення ПЗ повинні виконуватись з відпрацюванням ряду стандартів (правил, угод), яких повинні дотримуватись всі учасники проекту.

Лекції №6-7
ЕТАП ВИЗНАЧЕННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

План:

1. Методи визначення вимог
2. Планування етапу визначення вимог
3. Формалізація вимог: виділення вимог за допомогою прецедентів
4. Формалізація вимог: псевдокод, кінцеві автомати, графічні дерева рішень
5. Формалізація вимог: візуальне подання вимог за допомогою діаграм UML
6. Завдання та результати етапу аналізу вимог

Використані джерела:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.
2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.
4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002
5. Карл И. Вигерс. Разработка требований к программному обеспечению — Русская редакция, 2004.
6. Кобёрн А. Современные методы описания функциональных требований к системам — М.: Лори, 2002.
7. Леффингуелл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению — М.: Вильямс, 2002.
8. Ю. Ю. Якунин. Технологии разработки программного обеспечения - Красноярск: ИПК СФУ, 2008

9. IEEE 830-1998. Recommended Practice for Software Requirements Specifications - New York: IEEE, 1998

10. IEEE 1233-1998. Guide for Developing System Requirements Specifications - New York: IEEE, 1998

Вступ

Керування вимогами – це процес систематичного виявлення, організації та документування вимог до програмного забезпечення, а також процес, в ході якого виробляється і забезпечується угода між замовником та виконуючою групою з приводу змінюваних вимог до системи.

Після одержання загального уявлення про діяльність та цілі організацій, в яких працюватиме майбутня програмна система, про її предметну галузь та про наявні проблеми, можна визначити більш чітко, які саме задачі система вирішуватиме. Крім того, важливо розуміти, які з задач найбільш гострі і обов'язково повинні бути підтримані вже в першій версії, а які можуть бути відкладені до наступних версій або взагалі винесені за межі галузі відповідальності програмної системи. Ця інформація виявляється при аналізі потреб можливих користувачів та замовників. Потреби визначають на основі найбільш актуальних проблем та задач, які користувачі та замовники бачать перед собою. При цьому необхідним є акуратне виявлення значущих проблем, визначення того, наскільки добре вони вирішуються при поточному становищі справ, а також виділення пріоритетів при розгляді недостатньо добре вирішуваних проблем.

Формулювання *потреб* може розбиватись на наступні етапи: 1) виділення однієї-двох-трьох основних проблем; 2) визначення причини виникнення проблем, оцінювання ступеня їх впливу, виділення найбільш істотних проблем, які спричиняють появу наступних; 3) визначення обмежень на можливі рішення. Формулювання потреб не повинне накладати зайвих обмежень на можливі рішення, які їм задовільнюють. Потрібно спробувати сформулювати, що саме є проблемою, а не пропонувати одразу можливі рішення. При виявленні потреб користувачів аналізуються

моделі діяльності користувачів та організацій, в яких вони працюють, для виявлення проблемних місць.

Після виділення основних потреб потрібно вирішити питання про розмежування області відповідальності майбутньої системи, тобто визначити, які з потреб потрібно намагатись задовольнити в її межах, а які – ні. При цьому всі зацікавлені особи розділяються на користувачів, які будуть безпосередньо використовувати створювану систему для вирішення своїх задач, і другорядних зацікавлених осіб, які не вирішують своїх задач за її допомогою, але чиї інтереси так чи інакше торкаються нею. Потреби користувачів задовольняються першочергово (на це слід виділити більше зусиль), а інтереси другорядних зацікавлених осіб повинні бути тільки адекватно враховані у результуючій системі.

На основі виділених потреб користувачів, віднесених до області відповідальності системи, формулюються можливі *функції* майбутньої системи, які представляють собою послуги, надавані системою та задовольняючі потреби однієї або декількох груп користувачів. Формулювання функцій повинне бути досить коротким, ясним для користувачів, без зайвих деталей. Наприклад, «всі дані про клієнтів зберігатимуться у БД», «статус виконання замовлення клієнт зможе взнати через Інтернет», «система підтримуватиме до 10000 одночасно працюючих користувачів», «розклад проведення ремонтних робіт будуватиметься автоматично» і т.і. Пропонуючи ті чи інші функції, потрібно вміти акуратно оцінювати їх вплив на структуру та діяльність організацій, в межах якої використовуватиметься ПЗ (це можна зробити на основі моделей діяльності, одержаних при аналізі предметної галузі).

Маючи набір функцій, які досить добре підтримують розв’язок найбільш істотних задач, з якими доведеться працювати розроблюваній системі, можна складати *вимоги* до неї, які представляють собою деталізацію роботи цих функцій.

Співвідношення між проблемами, потребами, функціями та вимогами показане на рис.6.1.

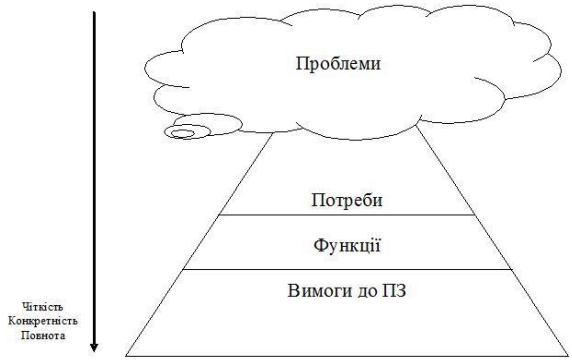


Рис.6.1 - Співвідношення між проблемами, потребами, функціями та вимогами

При цьому часто необхідно враховувати, що ПЗ є частиною програмно-апаратної системи, вимоги до якої потрібно перетворювати у вимоги до програмної та апаратної її складовим. Останнім часом, в зв'язку із значним падінням цін на потужне апаратне забезпечення загального призначення, фокус уваги змістився, в основному, на програмне забезпечення. В багатьох проектах апаратна платформа визначається із загальних міркувань, а підтримку більшості потрібних функцій здійснює ПЗ.

Кожна вимога розкриває деталі поведінки системи при виконанні нею деякої функції за деяких обставин. При цьому частина вимог випливає з потреб та побажань зацікавлених осіб та рішень, які задовольняють ці потреби та побажання, а частина – із зовнішніх обмежень, які накладаються на систему, наприклад, основними законами предметної галузі, державним законодавством, корпоративною політикою і т.і.

Ще до переходу від функцій до вимог корисно розставити пріоритети та оцінити трудомісткість їх реалізації та ризикованість. Це дозволить відмовитись від реалізації найменш важливих та найбільш трудомістких функцій, які не відповідають бюджету проекта, ще до їх детального опрацювання, а також виявити можливі проблемні місця проекту – найбільш трудомісткі та незрозумілі функції, які ввійшли до нього.

Правила роботи з вимогами до ПЗ та найбільш загальними системними вимогами визначаються наступними двома *стандартами IEEE*:

1) IEEE 830-1998 Recommended Practice for Software Requirements Specifications (Рекомендована практика для специфікації вимог до ПЗ), який описує структуру документів для фіксації вимог до ПЗ, а також визначає характеристики, які повинен мати вірно складений набір вимог (коректність або адекватність, однозначність, повнота, несуперечливість, впорядкованість, можливість перевірки, модифікованість, відстежуваність в ході розроблення);

2) IEEE 1233-1998, 2002 Guide for Developing System Requirements Specifications (Керівництво з розроблення специфікацій вимог до систем), який описує правила побудови вимог для програмно-апаратних систем в цілому, а також виділяє необхідні властивості набору вимог (одноразове згадування окремих вимог, відсутність перетинів між вимогами, явне вказання зв'язків між вимогами, повнота, несуперечливість, визначення обмежень, визначення області дії та контексту для кожної вимоги, модифікованість, конфігурованість, зручність підтримки, прийнятний для системи рівень абстракції). Крім того, стандарт визначає необхідні властивості кожної вимоги (абстрактність, однозначність, відстежуваність, можливість перевірки), а також приписує визначати атрибути для кожної вимоги (унікальний ідентифікатор, пріоритет, важливість реалізації з точки зору користувачів, критичність для побудови та успішності системи з точки зору аналітиків, здійснюваність з точки зору готовності користувачів до новацій та вартості реалізації, ризики високої вартості, ризики наслідків використання для навколишнього середовища, ризики конфліктів зі стандартами та законодавством, джерело, тип вимоги – вимоги на вхідні дані, вимоги на результуючі дані, вимоги надійності, вимоги робот здатності, вимоги щодо зручності супроводу, вимоги продуктивності, вимоги доступності, вимоги з точки зору обмежень навколишнього середовища, вимоги ергономічності, вимоги безпеки, вимоги захищеності, вимоги до обладнання, вимоги транспортованості,

вимоги щодо зручності навчання, вимоги щодо документованості, вимоги до зовнішніх інтерфейсів, вимоги до тестованості, вимоги якості, вимоги слідування корпоративним та законодавчим нормам, вимоги сумісності з відомими системаи, вимоги слідування стандартам та технологічним нормам, вимоги конвертації даних, вимоги щодо можливості збільшення системи, вимоги щодо зручності розгортання). Крім того, стандарт IEEE 1233 виділяє помилки, яких необхідно уникати при визначенні вимог (опис можливих рішень замість вимог, надто детальні специфікації, надто сильні обмеження, нечіткі вимоги, суб'єктивні вимоги, невизначені вимоги, не сформульовані припущення щодо режимів роботи та властивостей оточення).

1. Методи визначення вимог

Одним з найбільш важливих та зрозумілих методів одержання вимог є *інтерв'ю з клієнтом*; цей метод можна використовувати практично в будь-якій ситуації. Одна з основних задач інтерв'ювання – зробити все можливе, щоб упередження та переваги інтерв'юованих не впливали на вільний обмін інформацією. Це складна проблема. Соціологія показує, що неможливо сприймати оточуючий світ, не фільтруючи його відповідно до свого походження та накопичення досвіду.

Процес проведення інтерв'ю складається з етапів:

1) розроблення питань (зразок питань представлено у таблиці 6.1);

2) вибір опитуваних користувачів – існує декілька груп користувачів: персонал початкового рівня, організатори проекту середнього рівня, менеджери та інші замовники особливого роду – генеральні директори та віце-президенти, наукові співробітники, звичайні користувачі системи;

3) планування контактів;

4) проведення інтерв'ю – інтерв'ю можуть проводитись у телефонному режимі, персонально або за допомогою Інтернету (відео конференція, чат, електронна пошта), однак кращим способом здійснення інтерв'ю є безпосереднє спілкування ;

5) завершення зустрічі та визначення наступних дій.

Таблиця 6.1 – Узагальнене контекстно-вільне інтерв'ю

Частина 1. Визначення профілю замовника та користувача
Ім'я
Компанія
Галузь
Посада
(Вищенаведена інформація може бути внесена наперед)
Які ваші основні обов'язки?
Що ви, в основному, виробляєте, виготовляєте?
Для кого?
Як вимірюється успішність вашої діяльності?
Які проблеми впливають на успішність вашої діяльності?
Які тенденції, якщо такі є, роблять вашу роботу простіше чи складніше?
Частина 2. Оцінювання проблеми
Для яких проблем (прикладного типу) ви маєте нестачу гарних, якісних рішень, ідей?
Назвіть їх. (Не забувайте запитувати «А ще?»)
Для кожної проблеми виясняти наступне: Чому існує ця проблема?, Як вона вирішується наразі?, Як замовник хотів би її вирішувати?
Частина 3. Розуміння середовища користувачів
Хто такі користувачі?
Яка у них освіта?
Які їх навички в галузі комп'ютерної техніки та програмних засобів?
Чи мають користувачі досвід роботи з даним типом додатків?
Яка платформа використовується?
Які ваші плани відносно майбутніх платформ?
Чи використовуються додаткові додатки, які стосуються даного додатку?
Якщо так, то розповіти про такі додатки.
Які очікування замовника відносно практичності продукту?
Скільки часу необхідно для навчання?
В якому вигляді повинна надаватись довідкова інформація для користувача (в інтерактивному чи в друкованому)?

Частина 4. Резюме (перераховуються основі пункти, щоб перевірити, чи все вірно зрозумів розробник)

Отже, ви сказали мені (перераховуєте описані проблеми своїми словами).

Чи адекватно цей список представляє проблеми, які є при існуючих рішеннях?

Які ще проблеми (якщо такі є) ви маєте?

Частина 5. Припущення, аналітика відносно проблеми замовника (перевірені або неперевірені)

Для проблем, які не згадувались: Які проблеми, якщо вони є, пов'язані з (переахувати всі потреби або додаткові проблеми, які, по-вашомсу, може мати замовник або користувач)?

Для кожної названої проблеми слід вияснити:

- Чи є вона реальною?
- Які її причини?
- Як вона вирішується наразі?
- Як би замовник (користувач) хотів її вирішувати?
- Наскільки важливо для замовника (користувача) вирішення цієї проблеми в порівнянні з іншими, названими ним?

Частина 6. Оцінювання пропонованого вами рішення (якщо це можливо).

(Охарактеризуйте основні можливості пропонованого вами рішення і задайте питання:)

Що, якщо ви зможете вирішити дану проблему?

Як ви розцінюєте важливість цього?

Частина 7. Оцінювання можливості.

Хто в організації потребує даного додатку?

Скільки вказаних користувачів використовуватимуть його?

Наскільки значущим є для вас успішне рішення?

Частина 8. Оцінювання необхідного рівня надійності та продуктивності, а також потреби у супроводі.

Які ваші очікування відносно надійності?

Якою, по-вашому, повинна бути продуктивність?

Чи будете ви займатись підтримкою продукту чи цим займатимуться інші?

Чи маєте ви потребу у підтримці? Що ви думаєте про доступ для супроводу та обслуговування? Які вимоги стосовно безпеки? Які вимоги стосовно встановлення та конфігурації? Чи існують спеціальні вимоги по ліцензуванню? Як буде розподілене програмне забезпечення? Чи є вимоги на маркування та упаковку?
Частина 9. Інші вимоги. Чи існують законодавчі вимоги, вимоги інформаційного середовища або інші стандарти, яких необхідно дотримуватись? Чи немає інших вимог, про які ми мали б знати?
Частина 10. Завершення. Чи існують інші питання, які я мав би вам задати? Якщо мені ще знадобиться задати вам декілька питань, чи можу я вам зателефонувати? Чи будете ви приймати участь в обговоренні вимог?
Частина 11. Заключення аналітика. Після інтерв'ю, поки його дані ще свіжі в вашій пам'яті, зафіксуйте три потреби або проблеми з найвищими пріоритетами, виявлені вами в бесіді з даним замовником (користувачем).

Наступним методом визначення вимог є *«мозковий штурм»*. Це метод проведення зібрання, при якому група людей намагається знайти рішення специфічної проблеми за допомогою накопичення всіх спонтанних ідей. Даний процес має ряд очевидних переваг: 1) підтримує участь всіх присутніх; 2) дозволяє учасникам розвивати ідеї один одного; 3) секретар веде запис всього ходу обговорення; 4) метод є застосовним за різних обставин; 5) як правило, в результаті використання цього методу можна одержати множину можливих рішень для будь-якої поставленої проблеми; 6) метод сприяє вільному мисленню, необмеженому звичайними рамками.

«Мозковий штурм» складається з двох фаз: генерація ідей та їх відбір. Основна мета на етапі генерації полягає в описі якомога більшої кількості ідей, не обов'язково глибоких. На етапі відбору головною задачею є аналіз всіх ідей. Відбір ідей включає

відсікання, організацію, впорядкування, розвиток, групування, уточнення і т.і.

«Мозковий штурм» можна проводити різними способами. Наприклад, розглянемо один з простих процесів. Всі основні учасники збираються в одній кімнаті, їм роздаються матеріали для приміток. Це може бути просто стос паперу та чорний маркер. Аркуші паперу повинні бути не менші 7х12 см і не більше 12х17 см. Кожному учаснику слід видати не менше 25 аркушів на кожен сеанс «мозкового штурму». Кожен учасник сеансу виконує одну з трьох ролей: лідер, секретар або член команди. Лідер відповідає за спрямування процесу в правильне русло, за порядок та допомагає секретарю робити записи. Секретар записує всі ідеї таким чином, щоб кожна людина, яка знаходиться в кімнаті, могла бачити ці записи. Учасники генерують ідеї.

Правила проведення «мозкового штурму»:

1) Не припускається критика або дебати;

2) Слід надати свободу фантазії;

3) Слід генерувати якомога більше ідей;

4) Варто переробляти та комбінувати ідеї.

Після завершення фази генерації ідей починається процес відбору, який складається з декількох етапів:

1) відсікання – відсікання тих ідей, які не варті уваги; якщо є дві однакові ідеї, то вони об’єднуються;

2) групування ідей – групи одержують назви залежно від принципу групування. Наприклад, нові функції, питання продуктивності, пропозиції з вдосконалення існуючих функцій, інтерфейс користувача і т.і. Для будь-якої з груп можна відновити генерацію ідей, якщо виявиться, що процес групування стимулював виникнення нових ідей або деяка область важливих функцій них можливостей залишилась неохопленою;

3) визначення функцій – перераховуються всі ідеї, які залишились, і автори дають їх опис, який складається з одного речення;

4) розстановка пріоритетів.

Наступним методом є *сумісне розроблення додатків (JAD – Joint Application Design)*. Це зареєстрований товарний знак, який

належить до компанії IBM. Він представляє собою груповий підхід до визначення вимог, при реалізації якого особлива увага приділяється вдосконаленною групового процесу та вірному підбору людей, залучених до роботи над проектом. Сеанси JAD аналогічні сеансам «мозкового штурму», однак не у всьому. Сеанси «мозкового штурму» тривають близько двох годин, а сеанси JAD – до трьох днів. На сеансах «мозкового штурму» відбувається швидке генерування ідей, а на сеансах JAD розробляються високо рівневі специфічні програмні моделі функцій, даних та ліній поведінки. Сеанс JAD має певну структуру, на ньому дотримуються певної дисципліни, він відбувається під керівництвом арбітра. В його основі лежить обмін інформацією з використанням документації, фіксованих вимог та правил роботи. З моменту появи методики JAD на сеансах використовуються CASE-інструменти та інші програмні засоби, призначені для побудови діаграм потоку даних DFD, діаграм взаємозв'язків між сутностями ERD, діаграм зміни станів та інших об'єктно-орієнтованих діаграм.

Ролі в сеансах JAD:

- розробники – в задачу розробника входить надання допомоги організаторам у формулюванні їхніх потреб, які звичайно є рішенням існуючих проблем. Таким чином, визначення вимог до ПЗ відбувається сумісно із організаторами проекту;

- учасники – для успішного проведення сеансу ключовою вимогою є високий рівень кваліфікації запрошених. Вірний підбір людей дозволить швидко приймати рішення та розробляти вірні моделі, навіть якщо вони не будуть завершені;

- арбітр/консультант –повинен зводити до мінімуму прояв непродуктивних людських емоцій (агресію, самозахист). Арбітр не є власником ані процесу, ані продукту. Він присутній лише для того, щоб допомагати організаторам проекту розробляти програмний продукт;

- секретар – документує ідеї та допомагає стежити за часом.

Сеанс JAD – це своєрідна майстерня, працююча у максимально напруженому режимі,де рішення приймаються сумісно з усіма учасниками. При цьому учасники є крупними фахівцями в розглядуваному питанні.

Процес дослідження проекту розбивається на наступні етапи: пошук фактів та збирання інформації, підготовка сеансу JAD, проведення самого сеансу та перевірка зібраної інформації.

Результатами проведення сеансу можуть бути:

- діаграма контексту даних;
- діаграма потоку даних першого рівня;
- глобальна модель даних – діаграма взаємозв'язків між сутностями;
- перелік первинних об'єктів;
- об'єктна модель високого рівня;
- обов'язки кандидатів та співробітників для об'єкта;
- перелік первинних процесів / сценарії вибору;
- інші діаграми потоку даних, діаграми стану, дерева альтернатив, таблиці рішень;
- вимоги, які висуваються до даних для кожного процесу;
- перелік припущень;
- документація по аналізу відкритих питань.

Результати сеансу JAD використовуються в процесі визначення вимог для організації наступного етапу – створення специфікації вимог до ПЗ (SRS).

Отже, під час сеансу JAD учасники передають свої ідеї арбітру або секретарю. Для запобігання невірної інтерпретації зібраних даних, необхідно вжити деяких заходів запобігання. Використання під час сеансу автоматизованих інструментальних засобів та перевірка результатів всіма учасниками зменшить ризик. На сеансах JAD розглядаються переважно інформаційні системи, в яких особливу увагу приділено елементам даних та проекту інтерфейса. Є мало інформації про використання методу JAD для визначення вимог, які висуваються до систем реального часу. На проведення триденного сеансу JAD из представниками всіх груп організаторів проекту, кожна з яких складається з кваліфікованих фахівців, потрібно чимало коштів. Для аналізу складних вкладених систем реального часу та систем, від яких залежить людське життя, часто потрібно більше часу, ніж 3 дні. Якщо сеанс триває до моменту настання втоми учасників, то втома може наступити в той момент, коли будуть лише визначені сценарії вибору.

Наступний метод визначення вимог – *розкадрування*. Метою розкадрування є одержання ранньої реакції користувачів на запропоновані концепції додатку. За її допомогою можна на найбільш ранніх етапах ЖЦ спостерігати реакцію користувачів, до того як концепції будуть перетворені в код, а в багатьох випадках навіть до розроблення детальної специфікації. Розкадрування має наступні переваги: невисока вартість, дружність користувачу, інтерактивність, неформальність, забезпечення раннього аналізу інтерфейс ів користувача, легкість у створенні та модифікованість. Розкадрування можна використовувати для прискорення концептуального розроблення різних сторін додатку. Його можна застосовувати для розуміння візуалізації даних, визначення та розуміння бізнес-правил, які будуть реалізовані і новому бізнес-додатку, для визначення алгоритмів та інших математичних конструкцій, які будуть виконуватись всередині вбудованих систем, або для демонстрації звітів та інших результатів на ранніх етапах. Розкадрування можна (і потрібно!) використовувати практично для всіх додатків, в яких раннє одержання реакції користувачів є ключовим фактором успіху.

Розкадрування поділяються на три типи в залежності від режиму взаємодії з користувачем: пасивні, активні та інтерактивні. *Пасивні розкадрування* являють собою історію, яка розповідається користувачу. Вони можуть складатись зі схем, картинок, миттєвих копій екрану, презентацій PowerPoint або зразків результуючої інформації системи. В пасивному розкадруванні аналітик відіграє роль системи і просто проводить користувача розкадруванням, пояснюючи можливі кроки та відгуки на них. *Активне розкадрування* забезпечує автоматизваоний опис поведінки системи при типовому використанні або в операційному сценарії. Воно створюється за допомогою анімації або автоматизації, можливо, за допомогою автоматичного послідовного показу слайдів, анімаційних засобів або навіть фільму. *Інтерактивне розкадрування* дає користувачу досвід спілкування з системою майже такий реальний, як на практиці. Для свого виконання воно вимагає участі користувачів. Інтерактивне розкадрування може бути імітаційним, у вигляді макету або навіть може представляти

відкидуваний пізніше код. Складне інтерактивне розкадрування, засноване на відкидуваному коді, може бути вельми схожим на відкидуваний прототип. Ці три типи розкадрування пропонують широкий спектр можливостей – від зразків результуючої інформації до «живих» демонстраційних версій. Різниця між складним розкадруванням та ранніми прототипами продукту вельми умовна.

Вибір типу розкадрування залежить від складності системи і того, наскільки великий ризик, що колектив невірно розуміє її призначення. Для безпрецедентної нової системи, яка має розпливчасте визначення, може знадобитись декілька розкадрувань, від пасивної до інтерактивної, по мірі вдосконалення колективом розуміння системи.

Ще одним методом визначення вимог є *метод обігрування ролей*, який дозволяє колективу розробників відчути весь світ користувача після перебування в його ролі. Концпеція, яка лежить в основі даного методу, досить проста: хоча, спостерігаючи і задаючи питання, ми підвищуємо рівень свого розуміння, наївно вважати, що за допомогою самого спостереження розробник/аналітик може одержати істинно глибоке розуміння вирішуваної проблеми або вимог до системи, яка покликана дану проблему вирішити.

Аналітик або будь-який член колективу посідає місце користувача і виконує його звичайні дії. Наприклад, розглянемо проблему введення замовлень на купівлю. Розробник/аналітик може «відчути» проблеми та неточності, присутні існуючій системі введення замовлень на купівлю, просто посідаючи місце оператора і намагаючись ввести декілька замовлень. Одержаний досвід назавжди зімнить розуміння командної суті проблеми.

Існує два різновиди методу обігрування ролей: сценарний перегляд та CRC-картки. *Сценарний перегляд* – це виконання ролі на папері. При сценарному перегляді кожен учасник слідує сценарію, який задає конкретну роль у «п’єсі». Перегляд буде демонструвати будь-які неточності в розумінні ролей, брак інформації, доступної актору або підсистемі, або брак конкретного опису поведінки, необхідного акторам для успішного виконання їх

ролі. Однією з переваг сценарного перегляду є те, що сценарій можна модифікувати знову стільки разів, скільки необхідно, доки актори не вважатимуть його вірним. Сценарій можна також повторно використовувати та програвати знову, коли необхідно змінити поведінку системи. В певному змісті даний сценарій стає «живим» розкадруванням для проекту. *CRC-картки* (Class-Responsibility-Collaboration: клас-обов'язок-взаємодія) як метод обігрування ролей часто застосовується в об'єктно-орієнтованому аналізі. В даному випадку кожному учаснику видається набір індексних карток, який описує клас (об'єкт), обов'язки (поведінку), а також взаємодії (з якими з модельованих сутностей взаємодіє об'єкт). Ці взаємодії можуть просто представляти сутності проблемної галузі або об'єкти, існуючі в галузі рішення. Коли актор-ініціатор ініціює певну поведінку, всі учасники слідують поведінці, заданій карткою. Якщо процес переривається через брак інформації або якщо однієї сутності необхідно поспілкуватись з іншою, а взаємодія не визначена, то картки модифікуються, і ролі розігруються знову.

Наступним методом визначення вимог є *швидке прототипування*. Швидке прототипування – це часткова реалізація системи ПЗ, створена з метою допомогти розробникам, користувачам та клієнтам краще зрозуміти вимоги до системи. Чим детальніше прототип, тим легше зрозуміти вимоги замовника. З іншого боку, прототипи самі по собі є програмами, тому, чим детальнішим є прототип, тим він дорожчий.

Для випадків, коли однозначно не можна визначити, розробляти прототип чи ні, потрібно оцінити витрати на розроблення прототипу та можливий прибуток від його реалізації. На основі цієї оцінки визначаються оптимальні витрати на прототип та приймається рішення про його розроблення та розмір фінансування у випадку позитивного рішення. По мірі того, як зростають витрати на прототип, зростає і його придатність, але також зростають і витрати з виділеного бюджету. В результаті, ймовірно, існує момент, в який витрати оптимальні (точка максимуму на кривій), і деяка точка, за якою гроші вже витрачені даремно (де крива перетинає горизонтальну вісь).

Існує багато способів розбиття прототипів на категорії. Виділяються наступні прототипи: відкидувані, еволюціонуючі, операційні; вертикальні та горизонтальні; інтерфейси користувача та алгоритмічні. Яким повинен бути прототип в кожному конкретному випадку, залежить від того, яку проблему ви намагаєтесь вирішити шляхом побудови прототипу.

2. Планування етапу визначення вимог

Процес керування вимогами представлено на рис.6.2.

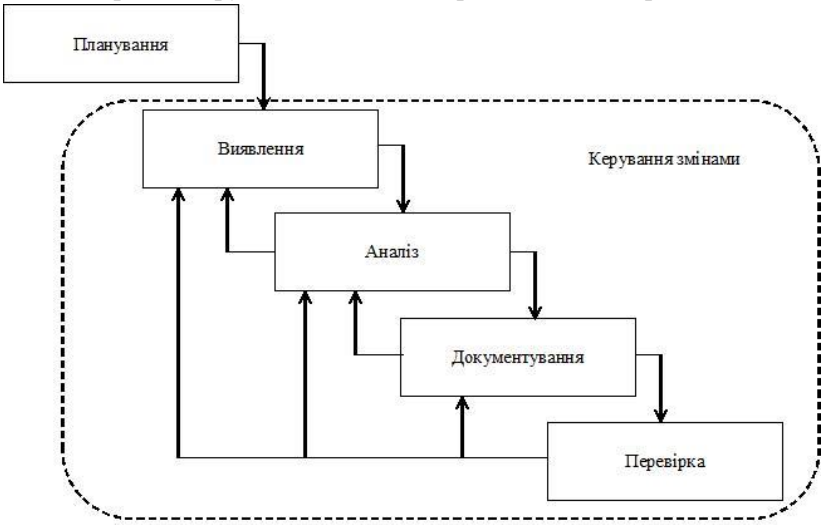


Рис.6.2 - Процес керування вимогами

Процес керування вимогами починається з планування. *Планування вимог* – це набір дій, які ініціюються на початку проекта або на початковій фазі робіт, пов'язаних з вимогами і спрямованих на визначення основних моментів, які необхідно враховувати в подальшій роботі при виявленні, аналізі, документуванні, перевірці та керуванні вимогами.

На етапі планування системний аналітик створює план керування вимогами та шаблони необхідної документації. Планування – перший крок при роботі з вимогами, він починається

на етапі перед проектного обстеження. Під час планування визначають, які дії (щодо аналітики) повинні виконуватись на проекті, як і ким вони будуть виконуватись в процесі роботи з вимогами. Таким чином, планування вимог включає в себе ідентифікацію ролей учасників проекту, вибір методології та процесу розроблення, підготовку шаблонів документів та регламентів робіт.

План керування вимогами є одним з найбільш важливих документів в процесі керування вимогами. В даному документі визначаються типи вимог та атрибути кожного типу, відношення між вимогами, використовувані у даному процесі документи. Також системний аналітик визначає і вносить до плану рішення про використання спеціального інструментального засобу для керування вимогами. Розширений варіант плану керування вимогами може містити опис ролей, які приймають участь у процесі, задач, виконуваних кожною роллю, та іншу інформацію.

На етапі планування створюються шаблони необхідних документів: глосарію, технічного завдання, документу-концепції. При роботі з державним замовником для опису вимог найчастіше використовується технічне завадання, розроблене відповідно до ГОСТ 34.602 або ГОСТ 19.201. Якщо немає жорстких вимог з боку замовника на відповідність державним стандартам, можна використовувати специфікацію вимог (SRS) на основі стандарту IEEE 830-1998. Трапляються випадки, коли замовник висуває жорсткі обмеження до оформлення технічної документації (за певним стандартом), в той час як колектив розробників вже декілька років працює, використовуючи Agile-методологію або якусь іншу. В такому випадку можна використовувати два різни документи: один для замовника – специфікація вимог (описує вимоги до системи), інший для проектного колективу – детальна специфікація, часткове технічне завдання (описує більш детальні вимоги, описи сутностей, детальні алгоритми роботи системи) і т.і. (рис.6.3).

Рис.6.3 - Вибір методології на етапі планування вимог

Сучасні інструментальні засоби дозволяють створювати автоматичні звіти з необхідною інформацією. Якщо прийнято рішення про те, що документація буде створюватись автоматично з використанням звітів, на етапі планування необхідно створити шаблони таких звітів. Шаблони звітів, як і шаблони документів, повинні бути розроблені з врахуванням вимог стандартів.

Чим ретельніше буде сплановано роботу з вимогами, тим вища ймовірність одержанні найкращого результату.

Основні етапи планування вимог:

1) побудова аналітичного процесу, вибір методології та підходу;

2) визначення проектних ролей (експерт предметної галузі, бізнес-аналітик, системний аналітик, координатор команди аналітиків);

3) розподіл обов’язків, персонал та його навички;

4) регламент узгодження вимог;

5) регламент керування змінами;

6) регламент роботи з аналітичними ризиками.

Основні результати планування:

1) методологія та процес розроблення ПЗ;

2) ролі учасників проекту та виконувані ними задачі;

3) список учасників проекту та їх навички;

4) різні регламенти робіт в частині виявлення, документування, узгодження вимог;

5) план керування вимогами;

6) шаблони документів.

План керування вимогами включає:

1) типи вимог (рис.6.4);

2) трасування вимог (рис.6.4);

3) атрибути вимог;

4) використовувані інструменти;

5) список аналітичних документів;

6) список ролей;

7) список аналітиків, інформація для зв'язку з ними;

8) список зацікавлених осіб;

9) засоби забезпечення взаємодії із замовником;

10) звіти.

Рис.6.4 - Типи вимог та трасування

Детальніше про планування процесу керування вимогами можна прочитати у другому випуску журналу Analyze IT - http://www.uml2.ru/downloads/journal/AnalyzeIT_March2009.pdf

3. Формалізація вимог: виділення вимог за допомогою прецедентів

Іноді присутня природній мові неоднозначність просто неприйнятна, особливо коли вимоги стосуються життєво важливих

питань або коли невірна поведінка системи може призвести до надзвичайних економічних або юридичних наслідків. Якщо визначення вимоги складно сформулювати природньою мовою і неможливо запобігти невірному розумінню специфікації, слід спробувати написати цю частину вимог за допомогою теоретично обґрунтованих формальних методів.

Метод варіантів використання (прецедентів) є частиною методології об'єктно-орієнтованого проектування. Цей метод аналізу та проектування складних систем, який представляє собою спосіб опису поведінки системи з точки зору того, як різні користувачі взаємодіють з нею для досягнення своїх цілей. Такий орієнтований на користувача підхід надає можливість дослідити різні варіанти поведінки системи при ранньому залученні користувача.

Варіанти використання можна успішно застосовувати протягом всього ЖЦ ПЗ: при аналізі, проектуванні і в процесі тестування. В цьому випадку, процес розроблення ПЗ називають «заснованим на варіантах використання».

Варіант використання – це функційний зв'язний блок, виражений в вигляді транзакції між актантом (активним, значущим учасником ситуації) та системою. Варіант використання описує послідовність дій, виконуваних системою з метою надати корисний результат конкретному актанту.

Варіантом використання (use case) називають деякий сценарій дій системи, який забезпечує відчутний та значущий для її користувачів результат. На практиці у вигляді одного варіанту використання оформляється сценарій дій системи, який буде, скоріше за все, неодноразово виникати під час її роботи та має досить чітко визначені умови початку виконання та завершення.

Модель варіантів використання системи складається з усіх актантів системи та різних варіантів використання, за допомогою яких актанти взаємодіють з системою, тим самим описуючи багатоваріантність її функціональної поведінки. Вона також відображає зв'язки між варіантами використання, що поглиблює наше розуміння системи.

В мові UML варіант використання зображається у вигляді овалу, позначеного іменем варіанту, який представляється. Варіанти використання можуть бути зв'язані з діючими особами (акторами), які приймають в них участь, зображаються у вигляді чоловічка та представляють різні ролі користувачів системи або зовнішні системи, які з нею взаємодіють.

Варіанти використання можуть бути зв'язані один з одним трьома видами зв'язків: узагальненням (generalization), розширенням (extend relationship) та включенням (include relationship). Діючі особи також можуть бути пов'язані один з одним за допомогою зв'язків узагальнення.

Перший крок моделювання варіантів використання полягає у створенні системної діаграми, яка описує границі системи та визначає її актанти. Це дозволяє паралельно здійснити етапи 3 і 4, в яких потрібно виявити зацікавлених осіб та визначити межі ПЗ.

Другий крок описує системну поведінку, тобто те, як користувачі взаємодіють із системою, здійснюючи деякі послідовності дій для досягнення певних цілей (рис.6.5).

Наступний крок полягає в уточненні деталей функціональної поведінки кожного варіанту використання. Специфікації варіантів використання складаються з текстових та графічних описів кожного варіанту використання, написаних з позицій користувача.

Рис.6.5 - Діаграма варіантів використання

Формування специфікації варіантів використання:

1) визначення потоку подій –текстового опису операцій актанту та різних відповідей системи. Потік подій описує, що, як передбачено, робитиме система в залежності від поведінки актанта. Не обов'язково описувати потік у текстовій формі, для цього можна використовувати діаграми взаємодії UML, а також інші формальні методи. Головне – забезпечити розуміння, оскільки єдиного підходу на всі випадки не існує. Однак, як правило, цілком підходить опис природньою мовою. Потік подій описує досягнення цілі варіанту використання і призначений для розгляду наступними зацікавленими особами: клієнтами, користувачами, розробниками варіантів використання, рецензентами, проектувальниками, тестерами, менеджером проекту, укладачем технічної документації, особами з відділу маркетингу та продажів. Варіант використання може мати різні потоки в залежності від умов, які виникають. Іноді ці потоки зв'язані з виявленими в процесі обробки помилковими умовами, в інших випадках вони можуть описувати додаткові способи оброблення конкретних умов;

2) виявлення перед- та постумов – використовувати передумови та посту мови потрібно лише тоді, коли необхідно прояснити поведінку, виражену варіантом використання. Важливо проводити різницю між подіями, які запускають потоки варіанту використання, і передумовами, які повинні бути виконані до того, як можна буде ініціювати варіант використання. Постумови дозволяють точно вказувати стан, який повинен бути істинним по завершенню варіанту використання, навіть якщо використовувались альтернативні шляхи.

Застосування варіантів використання має ряд переваг порівняно з традиційним підходом, коли визначаються окремі декларативні програмні вимоги:

- варіанти використання легко писати;
- варіанти використання пишуться мовою користувача;
- варіанти використання пропонують зв'язні сценарії поведінки, які зрозумілі як користувачу, так і розробнику.

Завдяки опису «сценаріїв поведінки», спеціалізованим елементам та нотаціям моделювання, які містяться в мові UML,

варіанти використання забезпечують додаткові можливості зв'язувати діяльність з розроблення вимог з проектуванням та реалізацією.

Графічне представлення варіантів використання в UML та їх підтримка різними інструментальними засобами моделювання забезпечують візуалізацію зв'язків між варіантами використання, що може сприяти розумінню складної програмної системи.

Сценарій, описаний за допомогою варіанту використання, може практично без змін застосовуватись як сценарій тестування під час перевірки правильності.

Якісно описаний варіант використання має наступні атрибути:

1) ім'я, яке ясно вказує на призначення варіанту використання (ВВ);

2) опис – декілька речень, які описують цей ВВ;

3) частота – наскільки часто даний ВВ виникає;

4) передумови – всі умови запуску варіанта використання;

5) постумови – всі умови, які повинні бути виконаня після успішного виконання варіанту використання;

6) основний сценарій роботи, який використовується в більшості випадків;

7) альтернативні сценарії, які виникають іноді; для кожного альтернативного сценарію вказуються його передумови;

8) задіяні діючі особи (необов'язково);

9) розширювані варіанти використання (необов'язково);

10) варіанти використання, які включаються (необов'язково);

11) статус «в розробленні», «готовий до перевірки», «в процесі перевірки», «підтверджено», «відхилено» (необов'язково);

12) припущення про оточення та хід роботи системи, використані при розробленні даного варіанту (необов'язково).

Крім того, варіанти використання можуть доповнюватись діаграмами інших видів – перш за все, сценарними діаграмами та діаграмами активностей, які описують послідовності дій компонентів, діаграмами стантів та переходів компонентів, діаграмами класів цих компонентів.

4. Формалізація вимог: псевдокод, кінцеві автомати, графічні дерева рішень

Псевдокод – це квазімова програмування; спроба поєднати неформальну природню мову зі строгими синтаксичними та керуючими структурами мови програмування. В чистому вигляді псевдокод складається з комбінації наступних елементів:

1) імперативні речення з одним дієсловом та одним об'єктом;

2) обмежена множина (не більше 40-50) «орієнтованих на дії» дієслів, з яких повинні конструюватись речення;

3) рішення, представлені формальною структурою IF-ELSE-ENDIF;

4) ітеративні дії, представлені структурами DO-WHILE та FOR-NEXT.

На рис.6.6 представлено приклад специфікації за допомогою псевдокоду алгоритма обчислення відкладеного доходу від послуг протягом даного місяця в бізнес-додатку.

```
Set Sum(X)=0
for каждого клиента x
   if клиент оплатил услуги вперед
      and ((Текущий месяц)>=(2 мес. после даты приобретения)) and ((Текущий
      месяц)<=(14 мес. после даты приобретения))
   then Sum(X) = Sum(x) + (сумма, заплаченная клиентом)/12
```

Рис.6.6 - Приклад специфікації з використанням псевдокоду

Фрагменти тексту на псевдокоді розташовані з виступами; такий формат використовується для того, щоб виділити логічні блоки. Сполучення синтаксичних обмежень, формату та розбиття істотно зменшує неоднозначність вимоги, яка в інакшому випадку була би дуже складною та незрозумілою. В той же час таке представлення вимоги зрозуміліше для людини, яка не є програмістом (непотрібні знання C++ або Java).

Представлене у вигляді *кінцевого автомату* описує можливі життєві цикли об'єкту і складається зі станів, з'єднаних переходами. Кожен стан – це такий період життєвого циклу об'єкта, коли він задовольняє певні умови. Деяка подія може призвести до переходу, в результаті якого об'єкт перейде у новий

стан. При переході може виконуватись дія, призначена даному переходу. Представлення у вигляді кінцевого автомату зображається на діаграмах станів та переходів. На рис.7.1 наведено приклад діаграми станів та переходів для об'єкту «Замовлення».

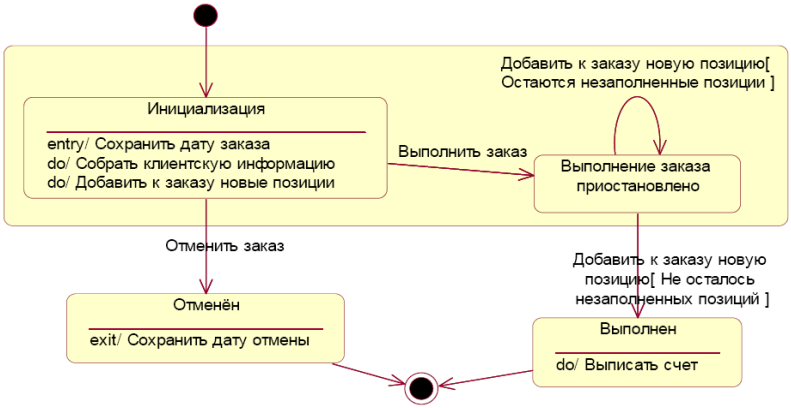


Рис.7.1 - Діаграма станів та переходів об'єкту «Замовлення»

Дерево рішень застосовується для відображення інформації у вигляді графу. На рис.7.2 відображене дерево рішень, використовуване для опису послідовності дій.

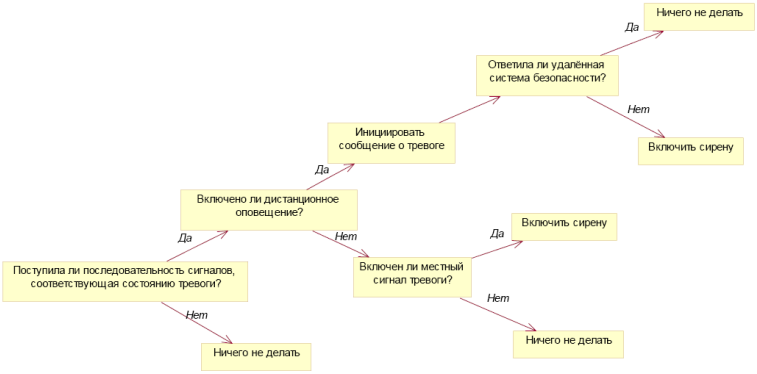


Рис.7.2 - Графічне дерево рішень

5. Формалізація вимог: візуальне подання вимог за допомогою діаграм UML

Стандарт UML містить наступний набір діаграм:

1) структурні (structural) моделі:

- діаграми класів (class diagrams);
- діаграми компонентів (component diagrams);
- діаграми розташування (deployment diagrams);

2) поведінкові (behavioral) моделі:

- діаграми варіантів використання (use case diagrams);
- діаграми взаємодії (interaction diagrams): діаграми послідовності (sequence diagrams) та діаграми кооперації (collaboration diagrams);
- діаграми станів (statechart diagrams);
- діаграми діяльності (activity diagrams).

Сукупність цих діаграм є самодостатньою у тому змісті, що в них утримується вся інформація, що необхідна для реалізації проекту складних систем.

Діаграми класів. Діаграма класів показує статичну структуру частини системи. Таким чином, дані діаграми описують класи, об'єкти й відносини між ними. Нотація класів й об'єктів проста й інтуїтивно зрозуміла всім, хто коли-небудь мав досвід роботи з різного роду CASE-інструментами. Клас представлений прямокутником із трьома розділами, у яких відповідно містяться ім'я класу, атрибути й операції (рис. 7.3). Схожа нотація застосовується й для об'єктів - екземплярів класу, з тим розходженням, що до імені класу додається ім'я об'єкта й весь напис підкреслюється. Нотація UML надає широкі можливості для відображення додаткової інформації (абстрактні операції й класи, стереотипи, загальні й часткові методи, інтерфейси й т.д.). Асоціації, тобто статичні зв'язки між класами, зображуються у вигляді сполучної лінії, на якій може вказуватися потужність асоціації, її напрямок, назва й можливе обмеження, що реалізує механізм розширення UML.

Існує можливість відбити специфічні властивості асоціації, наприклад: відношення агрегації, коли складовими частинами класу можуть виступати інші класи. Таке відношення зображується

у вигляді ромба, розташованого поруч із класом, що агрегує. Відношення узагальнення також має власну графічну нотацію у вигляді трикутника й сполучної лінії, дозволяючи представити ієрархію спадкування: від суперкласу до підкласів.

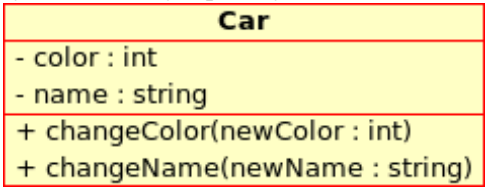


Рис.7.3 - Представлення класу на діаграмах класів UML

Діаграми розташування. Діаграми розгортання (розташування) використовуються для подання конфігурації компонентів, процесів й об'єктів, що є присутнім в системі на етапі виконання. Крім того, діаграми розміщення показують фізичну залежність апаратних пристроїв, задіяних у реалізації системи, а також з'єднань між ними - маршрутів передачі інформації.

Для побудови діаграми на UML використовують 4 види графічних конструкцій:

1) значки (або піктограми) - графічна фігура фіксованого розміру й форми, вони не можуть збільшувати свої розміри, щоб розміщати усередині себе додаткові символи. Значки можуть розміщатися як усередині, так і поза графічною конструкцією;

2) графічні символи на площині - геометричні фігури з змінюваними розмірами для розміщення інших символів;

3) шляхи - послідовності з відрізків ліній, що з'єднують окремі графічні символи, при цьому кінцеві крапки шляху обов'язково стикаються з геометричними фігурами;

4) рядки тексту - служать для подання різної інформації у деякій граматичній формі, що допускає розбір.

При графічному зображенні діаграм слід керуватись наступними правилами:

1) діаграма повинна бути закінченим поданням розглянутого фрагмента предметної області;

2) всі сутності на діаграмі одного концептуального рівня;

3) вся інформація реальна на діаграмі;

4) мінімальність пояснювального тексту;

5) немає суперечливої інформації;

6) діаграма повинна бути самодостатньою;

7) кількість типів діаграм для конкретної моделі не є строго фіксованим.

Діаграми варіантів використання. Техніка варіантів використання була вперше запропонована Айваром Якобсоном в 1992 і швидко завоювала загальне визнання за рахунок простоти й легкості сприйняття й застосування. Суть її полягає в наступному: проектована система представляється у вигляді наборів акторів, взаємодіючих із системою за допомогою так званих варіантів використання. Актором є будь-яка сутність, взаємодіюча із системою ззовні. Ним може бути людина, устаткування, інша система, тобто ми визначаємо, що взаємодіє з системою. У свою чергу варіант використання описує, що система надає акторові, тобто визначає деякий набір транзакцій, чинений актором при діалозі із системою, при цьому нічого не говоритися про те, яким чином буде реалізована взаємодія. Діаграма варіантів використання несе в собі високий рівень абстракції, що дозволяє ще на ранніх етапах проекту визначити й зафіксувати функціональні вимоги до системи й забезпечити гнучкий й ефективний механізм взаємодії між розроблювачем й замовником проекту.

Діаграми взаємодії. Діаграми взаємодії описують взаємодію об'єктів системи, виконуване ними для одержання деякого результату. Під одержанням результату мається на увазі виконання закінченого дії, наприклад, опис у термінах взаємодіючих об'єктів змодельованого раніше варіанта використання системи або деякого сервісу системи, оголошеного як операція класу на відповідній діаграмі. Діаграми взаємодії представляються у двох формах: діаграма послідовності й діаграма кооперації. І та, й інша описують потоки повідомлень (виклики методів або сигнали) між об'єктами, що беруть участь у взаємодії.

Діаграма послідовності. Діаграма послідовності наголошує на тимчасову послідовність переданих повідомлень, порядок, вид й ім'я повідомлення, на діаграмі зображуються винятково ті об'єкти,

які безпосередньо беруть участь у взаємодії й не показуються можливі статичні асоціації з іншими об'єктами.

Таким чином, для діаграми послідовності ключовим моментом є динаміка взаємодії. Діаграма послідовності має два виміри (рис.7.4). Одне – ліворуч і праворуч у вигляді вертикальних ліній, що зображують об'єкти, які приймають участь у взаємодії. Верхня частина ліній доповнюється прямокутником, що містить ім'я класу об'єкта або ім'я екземпляра об'єкта. Другий вимір - вертикальна тимчасова вісь. Повідомлення, що посилається одним об'єктом іншому, зображуються у вигляді стрілок з ім'ям повідомлення й упорядковані за часом виникнення.

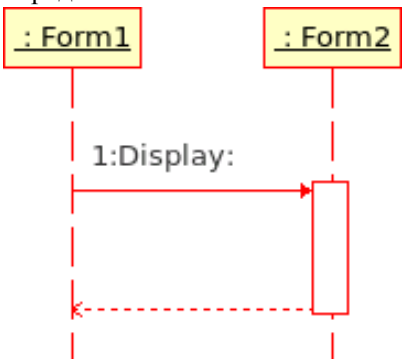


Рис.7.4 - Приклад діаграми послідовності

Діаграми кооперації. Для діаграми кооперації головною є можливість відобразити не стільки послідовність взаємодії, скільки всі оточення об'єктів, що беруть участь у ньому. Тобто показані не тільки ті, що посилають і приймають повідомлення, але й непрямі зв'язки між асоційованими об'єктами. Діаграми кооперації описують повний контекст взаємодії і являють собою своєрідний часовий "зріз" конфігурації мережі об'єктів, взаємодіючих для виконання певної бізнес - мети програмної системи.

Діаграма кооперації зображує об'єкти, що беруть участь у взаємодії у вигляді прямокутників, що містять ім'я об'єкта, його клас й значення атрибутів. Асоціації між об'єктами, як і на діаграмах класів, зображуються у вигляді сполучних ліній.

Можлива вказівка імені асоціації й ролей, які грають об'єкти в даній асоціації.

Динамічні зв'язки - потоки повідомлень представляються також у вигляді сполучних ліній між об'єктами, зверху яких розташовується стрілка із вказівкою напрямку й імені повідомлення.

Діаграми станів. Діаграми станів (рис.7.5) описують поводження об'єкта в часі, тобто моделює всі можливі зміни в стані об'єкта, викликані зовнішніми впливами з боку інших об'єктів або ззовні. Діаграми станів застосовуються для опису поводження об'єктів і для опису операцій класів. На відміну від діаграм взаємодії даний тип діаграм описує зміну стану тільки одного класу або об'єкта. Кожен стан об'єкта представляється на діаграмі станів у вигляді прямокутника з закругленими кутами, що містить ім'я стану і значення атрибутів об'єкта в цей момент часу. Перехід здійснюється при настанні деякої події: одержанні об'єктом повідомлення або прийомом сигналу й зображується у вигляді стрілки, з'єднуючої два сусідніх стани. Ім'я події вказується на переході. Крім того, на переході можуть указуватися дії, вироблені об'єктом у відповідь на зовнішні події (при переході з одного стану в інший або при знаходженні в певному стані). Діаграма стану описує, в основному, реакцію об'єкта на асинхронні зовнішні події, для опису реакції на внутрішні події призначені діаграми активності. Спрацьовування переходу може залежати не тільки від настання деякої події, але й від виконання певної умови, що називається пусковою умовою. Об'єкт перейде з одного стану в інший тільки в тому разі, якщо відбулася зазначена подія й пускова умова прийняла значення "істина".

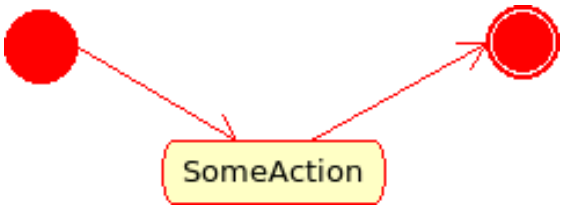


Рис.7.5 - Приклад діаграми станів

Діаграми діяльності. Діяльність – це потік робіт або виконання операції. Представлення діяльності відбиває як послідовні, так і паралельні види діяльності. Зображаються такі моделі на діаграмах діяльності. На рис.7.6 зображено діаграму, яка описує дії (діяльності) покупця в Інтернет-магазині. Тут представлені дві точки розгалуження – для вибору способу пошуку товара та для прийняття рішення про купівлю. Присутні три лінійки синхронізації: верхня – відображає розділення на два паралельних процеси, середня – відображає і розбиття, і злиття процесів, а нижня – лише злиття процесів. Додатково на цій діаграмі показано дві доріжки – доріжка покупця та доріжка магазину, які розділені вертикальною лінією. Кожна доріжка має ім'я та фіксує область діяльності конкретної особи, позначаючи зону її відповідальності.

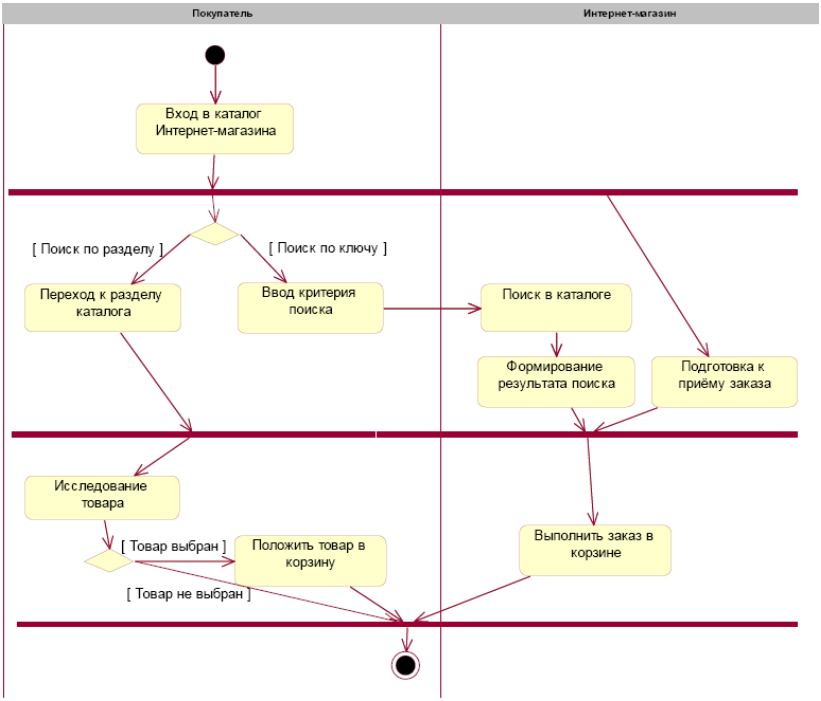


Рис.7.6 - Діаграма діяльності покупця в Інтернет-магазині

Діаграма діяльності відбиває реальні потоки робіт в людській організації. Таке бізнес-моделювання і є основним її призначенням. Також її можна використовувати при моделюванні робіт програмного додатку.

Діаграми діяльності (активності) - окремий випадок діаграм станів. Кожен стан - це виконання деякої операції, і перехід в наступний стан спрацьовує тільки при завершенні операції у вихідному стані. Таким чином, реалізується принцип процедурного, синхронного керування, обумовленого завершенням внутрішніх дій. Описуваний стан не має внутрішніх переходів й переходів по зовнішніх подіях. Графічна нотація практично не відрізняється від нотації діаграм станів з тією різницею, що на переходах відсутня сигнатура події й доданий символ "синхронізації" переходів для реалізації паралельних алгоритмів.

Основним напрямком використання цієї діаграми є опис операцій класів, коли необхідно представити алгоритм її реалізації, при цьому кожен крок є виконанням операції деякого класу.

6. Завдання та результати етапу аналізу вимог

На практиці часто застосовується підхід, використовуваний в різних методологіях розроблення ПЗ, який базується на визначенні груп вимог до продукту. Такий підхід звичайно включає групи (типи, категорії) вимог, наприклад: системні, програмні, функційні, нефункційні (рис.7.7).

Щоб побудувати *будь-що*, ми, перш за все, повинні зрозуміти, чим *це* має бути. Процес розуміння та документування *цього* і називається *аналізом вимог*. Звичайно вимоги виражають, *що* додаток повинен робити: часто тут не намагаються сформулювати, як досягнути виконання цих функцій. Наприклад, вираз Y (Система повинна надавати користувачу доступ до балансу його банківського рахунка) є вимогою для бухгалтерського додатку, а вираз N (Баланси рахунків клієнтів зберігатимуться в таблиці під назвою «баланси» в базі даних) не є вимогою для додатку, оскільки стосується того, *як* повинен бути побудований додаток, а не того, *що* цей додаток повинен робити. Вимога на одному рівні часто переходить в одну або декілька конкретних

вимог на наступному, більш детальному рівні, тоді твердження N може стати вимогою на наступних рівнях процесу розроблення. Зустрічаються виключення з правила, яке забороняє специфікувати у вимогах, як повинен працювати додаток. Наприклад, у замовника можуть бути особливі причини вимагати, щоб баланси рахунків зберігались у базі даних з конкретним ім'ям, тоді твердження N дійсно стає вимогою.

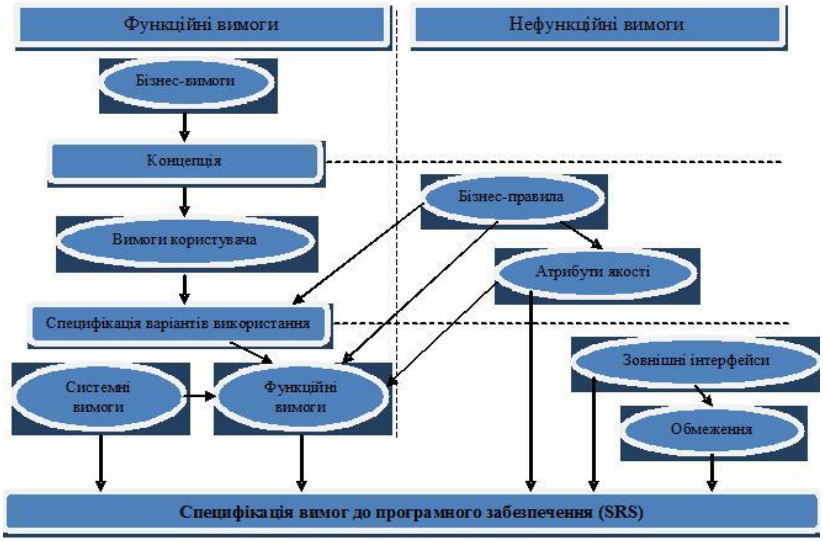


Рис.7.7 - Рівні вимог за Вігерсом

Протягом деякого часу відбувались дебати відносно того, кому «належать» вимоги: замовнику або розробникам. Для вирішення цього питання аналіз вимог можна розділити на два рівні. Перший рівень документує бажання та потреби замовника і пишеться мовою, зрозумілою замовнику. Результати іноді називають *вимогами замовника або С-вимогами*. Первинною аудиторією для С-вимог буде спільнота замовників, а вторинною – спільнота розробників. Другий рівень документує вимоги в спеціальній структурованій формі. Ці документи називають *вимогами розробника або D-вимогами*. Первинною аудиторією для

D-вимог буде спільнота розробників, а вторинною – спільнота замовників.

Хоча цільові аудиторії для C- і D-вимог різні, замовники та розробники тісно співпрацюють при створенні успішних продуктів. Один з способів, який дозволяє забезпечити гарну взаємодію – сумісна робота замовників та розробників. Це принцип екстремального програмування.

Навіть новачку може здатись очевидним, що потрібно письмово формулювати, як повинна поводитись завершена програма. Однак цей процес написання часто ігнорується або виконується з помилками. В таких випадках іноді вважають, що сирцевий код виражає всі вимоги: оскільки ми не можемо обійтись без сирцевого коду, то чому б не звести весь процес до цього єдиного документа? Але так не може бути. Теорія розроблення програм, досвідчені інженери, багаторічна практика наполягають на ретельному документуванні вимог. Без таких документів колектив практично не знає, яких цілей вона намагається досягти, не може коректно перевірити свою роботу, простежити свою продуктивність, одержати адекватні дані по своїй роботі, передбачити обсяг та зусилля у своїй наступній роботі та задовольнити своїх замовників. Отже, не може бути професійного розроблення без задокументованих вимог.

Аналіз вимог – це необхідність, а не розкіш. Якщо вартість виявлення та виправлення помилки під час формулювання вимог складає 100$, то вартість виявлення та виправлення того ж дефекту в кінці процесу розроблення складатиме 2000-5000$. Важливим виграшем від аналізу вимоги є досягнення розуміння та згоди відносно додатку, який створюється. Коректне формулювання вимог - достатньо складний процес. Конкретні, реалістичні та вимірювані вимоги є показником високого професійного рівня.

Результатом аналізу вимог є документ, який називають *специфікацією вимог до програмного забезпечення* (SRS). Вона включає множину функціональних вимог, які описують всі взаємодії користувача з програмним забезпеченням, а також нефункціональні (додаткові) вимоги, які накладають обмеження на

проект чи реалізацію (наприклад, вимоги продуктивності, стандарти якості, обмеження).

Згідно IEEE 830-1998, специфікація вимог до програмного забезпечення повинна складатись з наступних розділів: 1) вступ (огляд продукту, мета, межі, посилання, визначення та абревіатури); 2) загальний опис (перспективи, функції, характеристики користувачів загальні обмеження, припущення й залежності); 3) конкретні вимоги: вимоги функціонального характеру – вимоги до поведінки ПЗ (бізнес-вимоги, вимоги користувача, функціональні вимоги) та вимоги нефункціонального характеру – вимоги до характеру поведінки ПЗ (бізнес-правила, системні вимоги, атрибути якості, зовнішні системи та інтерфейси, обмеження); 4) додаткові матеріали.

Висновки

У лекції розглядались питання, пов’язані із встановленням вимог до ПЗ. Встановлення вимог передує їх специфікації. Встановлення вимог стосується їх виявлення та оформлення у вигляді документа, який має переважно описовий характер. В результаті специфікації вимог розробляються більш формалізовані моделі вимог. Виявлення вимог пов’язане з їх пошуком за двома напрямками – в знаннях про проблемну галузь та в галузі прецедентів. Ці два напрямки дослідження вимог додають одна одну і призводять до визначення моделі ділової дяльності для розроблюваної системи. Існують різні методи виявлення вимог. Одержані від замовників вимоги можуть перекриватись та суперечити одна одній. Усунення дублювання та зняття суперечливості – задача бізнес-аналітика, який вирішує її за допомогою узгодження та перевірки обґрунтованості вимог. Для належного вирішення цієї задачі бізнес-аналітик повинен приписати вимогам певні ризики та пріоритети.

Великі проекти пов’язані із керування великими масивами вимог. Для таких проектів істотним моментом є позначення та класифікація формулювань вимог. Потім можна визначити ієрархію вимог. Здійснення подібних кроків забезпечує необхідний

рівень простежуваності вимог на наступних стадіях проекту, а також належне опрацювання запитів на зміни вимог.

Незважаючи на те, що встановлення вимог не включає формального моделювання систем, вже на цьому етапі ЖЦ розроблення ПЗ можна побудувати бізнес-модель основних вимог. Бізнес-модель може бути представлена у вигляді трьох загальних діаграм – діваграми контексту, діаграми бізнес-прецедентів та діаграми бізнес-класів.

Документ, який створюється в результаті виконання етапу встановлення вимог – документ опису вимог – починається з загального опису зауважень до проекту (який створюється в інтересах представлення проекту керівництву). Основні частини документу описують системні сервіси та системні обмеження. Заключна частина документа пов’язана з іншими проектними питаннями, включаючи подробиці, які стосуються проектного план-графіка та бюджета.

Лекція №8
АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ

План:

1. Планування архітектури
2. Проектування архітектури
3. Документування архітектури
4. Аналіз архітектури

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.

4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

5. Ю. Ю. Якунин. Технологии разработки программного обеспечения - Красноярск: ИПК СФУ, 2008

Вступ

Після аналізу предметної галузі, визначення та формулювання вимог потрібно дослідити можливі способи вирішення тих задачі, які поставлено у вимогах. Іноді досвід розробників одразу підказує, як можна вирішувати поставлені задачі, але частіше потрібно виконати *аналіз області рішень*. Метою цієї діяльності є розуміння, чи можна взагалі вирішити задачі, які стоять перед системою, за яких умов та обмежень це можна зробити, як вони вирішуються, якщо рішення є, а якщо немає – чи модна придумати спосіб його відшукати або одержати хоча б наближене рішення і т.і. Як правило, задача добре досліджена в межах якої-небудь галузі людських знань, але іноді

доводиться витратити деякі зусилля на генерацію власних рішень. Крім того, рішень, як правило, декілька, і вони відрізняються за деякими характеристиками, здатними надалі відіграти важливу роль в процесі розвитку та експлуатації створеної на їх основі програми. Тому важливо зважити їх плюси та мінуси і визначити, які з них найбільш підходять в межах даного проекту, або вирішити, що всі вони повинні використовуватись для забезпечення більшої гнучкості ПЗ. Коли визначено принципові способи вирішення всіх поставлених задач (може бути, в деяких обмеженнях), основною проблемою стає спосіб організації програмної системи, який дозволив би реалізувати всі ці рішення та при цьому задовольнити вимоги, які стосуються нефункційних аспектів розроблюваної програми. Шуканий спосіб організації ПЗ як системи взаємодіючих компонентів називають *архітектурою*, а процес її створення – *проектуванням архітектури ПЗ*.

Архітектура та інженерія як види людської діяльності існували задовго до появи комп'ютерних технологій. Перш за все, ці види діяльності зв'язував процес створення проекту – прототипу, прообразу передбачуваного або можливого об'єкту. Інакше кажучи, проектування ПЗ містить в своєму складі поняття «архітектура» та «інженерія». Архітектурне проектування ПЗ пов'язане із забезпеченням максимальної функціональності проектованих об'єктів.

Архітектура ПЗ – це представлення програмної системи, яке дає інформацію про компоненти системи, про взаємозв'язки між цими компонентами та правила, якіф регламентують ці взаємозв'язки. Таке представлення призначене для ефективного розроблення проекту такої системи. *Проектування ПЗ* передбачає генерацію властивостей системи на основі аналізу постановки задачі (моделей предметної галузі та вимог до ПЗ), а також досвіду проектувальника. *Проектування архітектури ПЗ* – це високорівневе проектування, метою якого є створення гнучкої структури, яка задовольняє всі основні вимоги та передбачає деяку ступінь свободи реалізації. Елементи архітектури ПЗ та моделі їх з'єднання призначені для задоволення вимог до проектованих програмних систем. В проекті архітектури ПЗ повинні бути

враховані функціональні та нефункціональні вимоги до ефективності, витривалості, розширюваності, відмовостійкості, продуктивності, можливості повторного використання, адаптації розроблюваного ПЗ. Архітектурний проект ПЗ дозволяє оперативно визначити, наскільки даний програмний продукт відповідає висунутим до нього вимогам.

Метою архітектурного проектування предметної галузі є наступні артефакти: розроблення архітектури множини систем, які входять до даної предметної галузі; складання плану реалізації моделі предметної галузі; реалізація моделі предметної галузі.

1. Планування архітектури

Сучасні методи розроблення ПЗ передбачають зворотній зв'язок між всіма діючими особами - від проектувальника до аналітика. Всі ці особи є учасниками процесу створення архітектури програмної системи. Під архітектурою розуміємо структуру компонентів програмної системи, взаємозв'язки, а також принципи та норми їх проектування і розвитку в часі. Перш ніж розпочати вивчення процесу планування архітектури, необхідно ознайомитись із плануванням програмних проектів в цілому, а також з основними діаграмами етапу планування.

Для графічного відображення етапів програмного проекту використовують мережеву діаграму, на якій зображається взаємодія етапів, їх послідовність і час виконання.

Мережева діаграма (мережа граф мережі, PERT-діаграма) – це графічне відображення робіт проекту та їхніх взаємозв'язків. Мережа - це повний комплекс робіт проекту із встановленими між ними зв'язками.

При побудові мережевого графіка робіт створюється граф (приклад - на рис.8.1), в якому вказуються:

- початкова точка – подія або набір подій, які відбулися до початку виконання даної фази процесу;
- тривалість – інтервал часу, за який процес повинен успішно завершити своє виконання;
- кінцева точка процесу – контрольна точка, в якій замовник перевіряє якість отриманих результатів процесу.

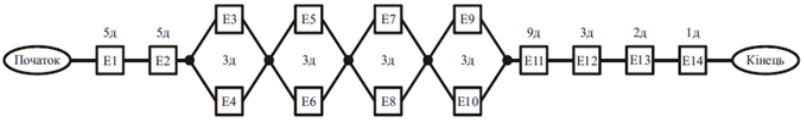


Рис.8.1 - Мережева діаграма етапів проекту №1

На рисунку 8.1: Е1 – аналіз предметної області, Е2 – формування вимог до програми, Е3 – побудова діаграми варіантів використання, Е4 – побудова діаграми класів, Е5 – побудова діаграми станів, Е6 – побудова діаграми діяльності, Е7 – побудова діаграми послідовності, Е8 – побудова діаграми кооперацій, Е9 – побудова діаграми компонентів, Е10 – побудова діаграми розгортання, Е11 – реалізація програми в кодах, Е12 – від лагодження, Е13 – тестування, Е14 – впровадженні і експлуатація.

Правила розроблення мережевої діаграми:

1) подія не може відбутись, якщо не завершені всі роботи, які до неї ведуть;

2) робота не може розпочатись, якщо не відбулась подія, яка лежить на її початку;

3) жодні дві роботи не можуть мати однакових початкових та кінцевих подій;

4) стрілки в мережевій діаграмі позначають відношення передування та слідування; стрілки можуть перетинатись;

5) кожна операція повинна мати свій власний номер;

6) номер наступної операції повинен бути більше номера будь-якої передуючої операції;

7) утворення петель неприпустиме;

8) умовні переходи від однієї операції до іншої неприпустимі;

9) один вузол повинен визначати початок всього комплексу робіт і один вузол – завершення.

Наприклад, для проекту №2 маємо наступну організацію виробництва інноваційної продукції – таблиця 8.1.

Таблиця 8.1 – Опис етапів проекту №2

№ операції	Операція	Попередні	Тривалість (в тижнях)
1	Підписання контракту	Немає	1
2	Реєстрація юридичної особи	1	4
3	Пошук приміщення	1	2
4	Наймання персоналу	1	1
5	Оренда приміщення	2, 3	1
6	Закупівля та постачання обладнання	2 (1)	8
7	Навчання персоналу	4	1
8	Ремонт приміщення	5	4
9	Монтаж обладнання	6, 8	1
10	Запуск обладнання	7, 9	1

Мережева діаграма такого проекту представлена на рис.8.2.

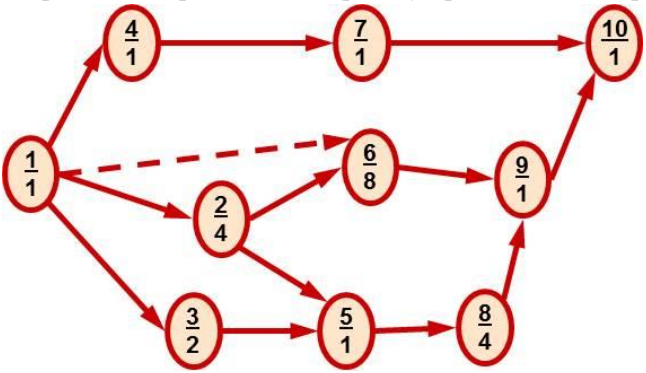


Рис.8.2 – Мережева діаграма проекту №2

Для проекту 3 маємо наступну таблицю етапів (таблиця 8.2) і мережеву діаграму (рис.8.3).

Таблиця 8.2 – Таблиця етапів проекту №3

Етап	Тривалість (в днях)	Залежність
T1	8	
T2	15	

T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

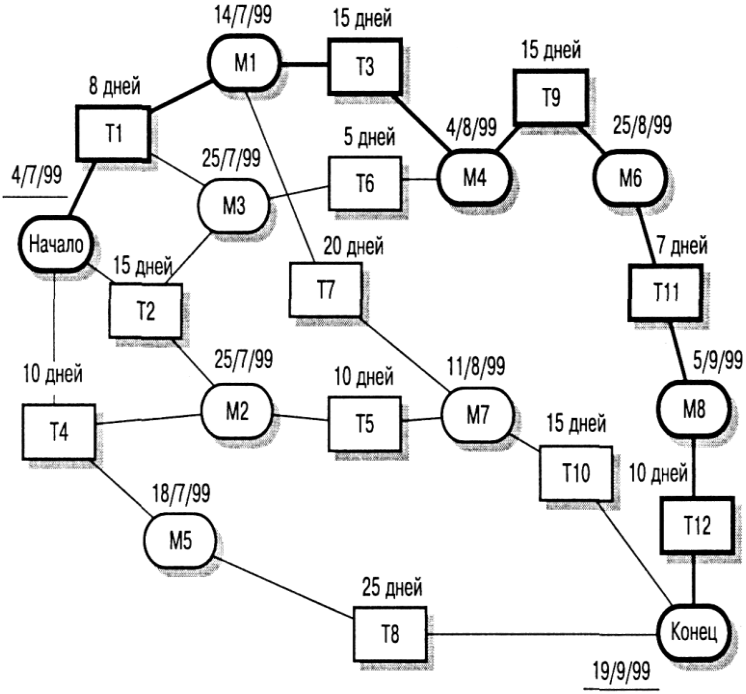


Рис.8.3 - Мережева діаграма проекту 3

Критичний шлях - найбільш довгий повний шлях в мережі; роботи, які лежать на цьому шляху, також називаються критичними. Саме тривалість критичного шляху визначає найменшу загальну тривалість робіт з проекту в цілому. Час

виконання всього проекту в цілому може бути скорочений за рахунок скорочення часу виконання завдань, які лежать на критичному шляху. Відповідно будь-яка затримка виконання завдань критичного шляху приводить до збільшення часу виконання проекту. Критичний шлях в графі вказує максимальну тривалість робіт на графі (від початкової роботи до останньої). При виконанні проекту вибираються і виконуються роботи, які не впливають на час виконання інших (незалежних) робіт проекту або на їх тривалість. Роботи на критичному шляху можуть скорочуватися за рахунок зміни часу виконання.

Наприклад, для підрахунку критичного шляху визначимо максимальний час реалізації проекту 1 (рис.8.1): Критичний шлях = 3*5днів+9*3днів+9днів+1день = 52 днів. Проте, мінімальний час реалізації проекту 1 (рис.2) становить: Мін.час = 5днів+5днів+4*3дні+9днів+3дні+5днів +1день=40 днів.

Тривалість критичного шляху проекту 2 (рис.8.2) складає 15 тижнів.

Найчастіше визначення ролей виконавців проекту відповідає етапам розроблення. Склад і кількість співробітників, що входять до групи проекту, залежить від масштабу робіт і досвіду співробітників. Співробітники повинні бути настільки кваліфікованими, щоб могли виявити помилки і неточності в проекті на самих ранніх етапах розроблення.

Для оцінювання часу зайнятості виконавців на етапах будується діаграма Ганта (приклад для проекту 1 - на рис.8.4). Діаграма Ганта - горизонтальна лінійна діаграма, на якій завдання проекту представляються термінами у вигляді відрізків часу і мають термін виконання можливо із затримками і іншими тимчасовими параметрами. На діаграмі Ганта розташовуємо роботи проекту за часом з врахуванням їх послідовності. Знаходимо можливості більш пізнього початку робіт. Роботи, які неможливо почати пізніше, лежать на критичному шляху.

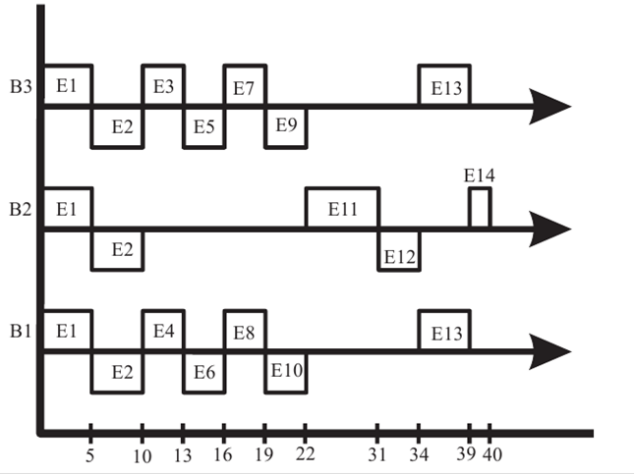


Рис.8.4 - Діаграма Ганта для проекту №1

На діаграмі шкала часу позначається у різних одиницях часу, наприклад, для проекту №1 – у днях.

Представимо роботи по проекту №4 таблицею 8.3.

Таблиця 8.3 – Роботи по проекту №4

Роботи	**Тривалість**	**Попередні**	**Наступні**
A	3	Початок	C, D
B	2	Початок	E
C	3	A	F
D	6	A	G
E	5	B	G
F	6	C	Кінець
G	4	D, E	Кінець

Тоді діаграма Ганта такого проекту має вигляд (рис.8.5).

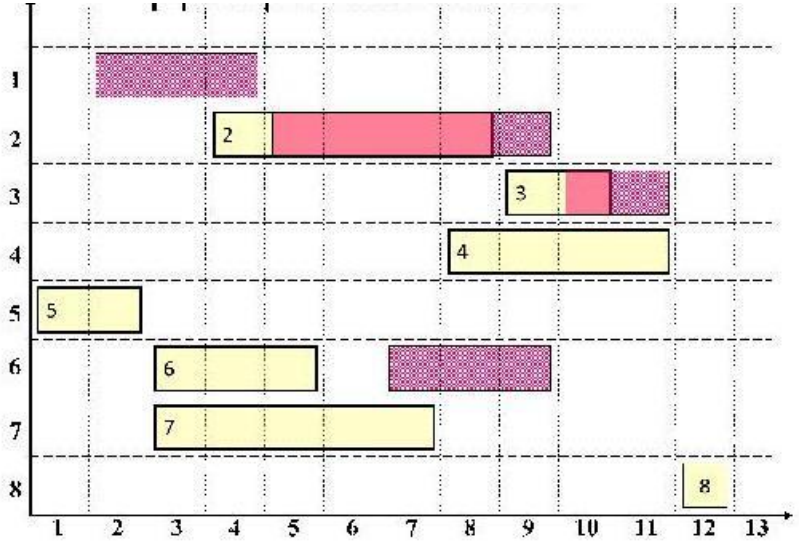


Рис.8.5 - Діаграма Ганта для проекту №4

Графік робіт представляється у вигляді діаграм і графіків. Часові діаграми відображають час початку і закінчення етапу на відносній (приклад для проекту №1 - на рис.8.6) та абсолютній шкалі (приклад для проекту №1 - на рис.8.7).

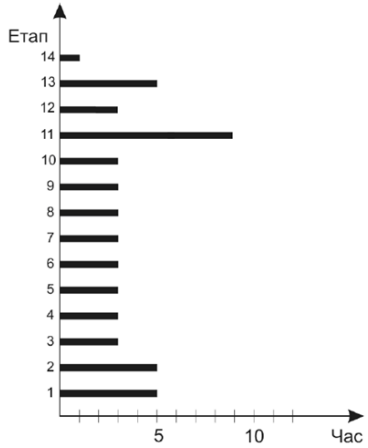


Рис.8.6 – Часова діаграма (відносна)

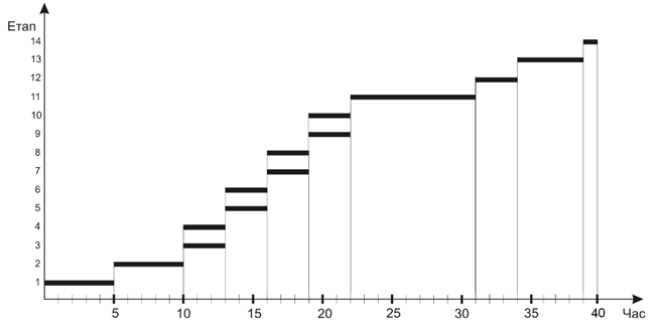


Рис.8.7 – Часова діаграма (абсолютна)

Знайомство із плануванням архітектури почнемо із поняття *архітектурно-економічного циклу* (АЕЦ). Взаємозв'язки між виробничими задачами, вимоги до продукту, досвід архітектора, архітектури та створені системи утворюють цикл з ланцюгами зворотнього зв'язку. Частково зворотній зв'язок надходить від самої архітектури, частково – від побудованої на її основі системи. Архітектура є основним визначальним фактором при заданні властивостей розроблюваної системи або систем.

Програмним процесом називаються дії з організації, нормування та керування розробленням ПЗ. Перелік операцій, спрямованих на створення програмної архітектури, її застосування для реалізації проектного рішення, а згодом – на реалізацію або керування розвитком цільової системи або додатку:

1) створення економічної моделі системи – із залученням архітектора надаються відповіді на запитання: «Скільки повинен коштувати продукт?», «Який цільовий сегмент ринку?», «Наскільки швидко продукт повинен вийти на ринок?», «Чи повинен він взаємодіяти із іншими системами?», «Чи є певні обмеження, в межах яких він повинен існувати?»;

2) виявлення вимог – сценарії, моделі кінцевих автоматів (приклад – на рис.8.8), мови формальних специфікацій, моделювання;

3) створення нової або вибір існуючої архітектури – основною умовою стабільного проектування системи є дотримання

концептуальної цілісності, а вона може проявитись лише у вузькому колі людей, які сумісно працюють над проектуванням її архітектури;

4) документування та поширення відомостей про архітектуру – щоб архітектура дійсно стала основою проекту, її суть необхідно чітко та однозначно донести до всіх зацікавлених осіб: розробники повинні розуміти, що від них вимагають, тестувальники повинні усвідомлювати структуру своїх задач, менеджмент повинен знати графік і т.і.;

5) аналіз або оцінка архітектури – в процесі проектування завжди розглядається множина варіантів проекту: деякі з них відбраковуються одразу, з числа інших відбирається найбільш прийнятний. Одна з глобальних задач, які стоять перед будь-яким архітектором, полягає якраз у тому, щоб зробити цей вибір раціонально. Оцінити архітектуру на предмет атрибутів якості, які вона забезпечує, абсолютно необхідно – без цього не можна бути впевненим, що кінцева система зможе задовольнити всі потреби зацікавлених осіб. Все більше поширення одержують методики аналізу, орієнтовані на оцінку атрибутів якості, які повідомляються архітектурою, наприклад, за методом аналізу компромісних архітектурних рішень – ATAM, за методом аналізу вартості та ефективності – CBAM;

6) реалізація системи на основі архітектури;

7) перевірка відповідності реалізації архітектурі.

Приклад автоматної моделі програми, що надає можливість адміністратору формувати декілька списків розсилок за електронними адресами і відправляти інформаційні повідомлення окремо в кожен список, зображено на рис.8.8.

Програмна архітектура - це структура структур програми, тобто викладення її програмних елементів, їхніх зовнішніх властивостей та встановлених між ними відношень. Зовнішніми (externally visible) властивостями називаються ті припущення, які сторонні елементи можуть висувати щодо даного елементу - зокрема, вони стосуються надаваних елементом послуг, робочих характеристик, усунення несправностей, сумісного використання ресурсів і т.і.

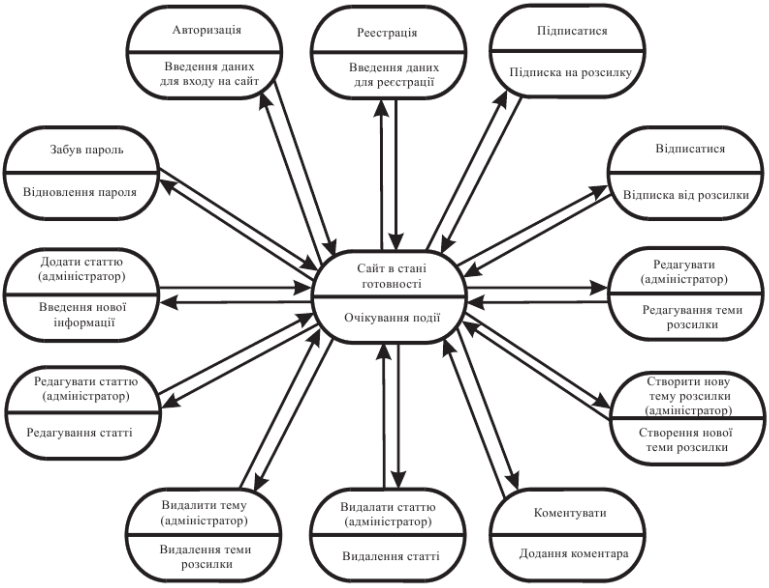


Рис.8.8 – Автоматна модель

Отже, архітектура визначає програмні елементи та взаємовідношення елементів, тобто архітектура - це, в першу чергу, абстракція системи, в якій відсутня інформація про елементи, яка не стосується того, як вони використовують, використовуються, співвідносяться та взаємодіють з іншими елементами. У склад будь-якої системи може входити і входить цілий ряд структур, а, отже, жодної окремо взятої структури, яку можна було б впевнено назвати архітектурою, не існує. Згідно визначення архітектури, програмна архітектура є в будь-якої обчислювальної системи з ПЗ. Е пов'язано із тим, що бущдь-яку систему можна представити як сукупність її елементів та встановлених між ними відношень. Якщо поведінку окремо взятого елемента можна зафіксувати або виявити з точки зору інших елементів, то ця поведінка входить до складу архітектури. Саме вона дозволяє елементам взаємодіяти між собою, а взаємодія, як відомо, відображається в архітектурі обов'язково.

Розглянемо питання оцінки архітектури та архітектурного проектування. Найпростішими варіантами архітектур є схеми з

прямокутників та ліній. Комплексні архітектури укомплектовані всією необхідною інформацією про систему. У проміжку між найпростішими та комплексними архітектурами інсують численні перехідні етапи, кожний з яких є результатом прийняття ряду архітектурних рішень, сукупністю архітектурних альтернатив. Розглянемо три таких проміжних етапи:

1) *архітектурний зразок* - це опис типів елементів і відношень та викладення ряду обмежень на їх використання; сукупність обмежень, накладених на архітектуру, точніше на типи елементів та зразки їх взаємодії (наприклад, архітектурний зразок "клієнт-сервер");

2) *еталонна модель* - це розподіл між окремими блоками функційних можливостей та потоків даних; це стандартна декомпозиція відомої проблеми на частини, які, взаємодіючи, задтні її вирішити; наявність еталонних моделей характерна лише для сформованих предметних галузей;

3) *еталонна архітектура* - це еталонна модель, відображена на програмні елементи, які спільно реалізують функційність, визначену в еталонній моделі, та потоки даних між ними; еталонна архітектура відображає функції на декомпозицію системи.

Сучасні програмні системи настільки складні, що розбирати їх у комплексі надто складно. Доводиться концентрувати увагу на одній чи декількох структурах програмної системи. Щоб міркувати про архітектуру, слід визначитись із тим, яка структура чи які структури в даний момент є предметом обговорення, про яке представлення (view) архітектури йде мова. *Представлення* - це відображення ряду зв'язаних архітектурних елементів у тому вигляді, в якому ними оперують зацікавлені в системі особи. В ньому фіксуються відображення сукупності елементів та встановлених між ними зв'язків. *Структура* - це власне ряд елементів, існуючих в межах ПЗ або АЗ. Зокрема, модульна структура представляє собою набір модулів системи з вказанням їх організації. *Модульне представлення* ж - це відображення цієї структури, документоване та застосовуване тими чи іншими зацікавленими особами.

Архітектурні структури поділяються на три загальні групи:

1) модульні структури;

2) структури "компонент і з'єднувач";

3) структури розподілу.

2.Проектування архітектури

Атрибутний метод проектування (Attribute-Driven Design, *ADD*) - це методика визначення програмної архітектури, в якій процес декомпозиції базується на передбачуваних атрибутах якості продукту. Це рекурсивний процес декомпозиції, на кожному з етапів якого відбувається відбір тактик та архітектурних зразків, які задовольняють тим чи іншим сценаріям якості, а також розподіл функційності, спрямований на конкретизацію типів модулів даного зразка. В контексті ЖЦ ADD розташований одразу після аналізу вимог, а починається він в той момент, коли про архітектурні мотиви можна говорити з достатньою впевненістю.

Результатом застосування ADD є перші декілька рівнів представлення декомпозиції модулів архітектури, а також всі інші зв'язані з ними представлення. Не варто думати, що після ADD стають відомими всі деталі представлень - система описується як набір контейнерів функційності та існуючих між ними взаємозв'язків. Це перший випадок з'єднання архітектури, результати якого вельми наближені. Але це важливий момент, тому що саме тут закладаються основи досягнення функційності. Різниця між архітектурою, яка є результатом виконання ADD, та архітектурою, готовою до реалізації, полягає в площині прийняття детальних проектних рішень.

Етапи ADD:

1) вибір модулю для декомпозиції - як правило, спочатку розглядається система в цілому; але повинні бути відомі всі необхідні вхідні дані (обмеження, вимоги);

2) уточнення модулю в декілька етапів:

а) вибір архітектурних мотивів з набору конкретних сценаріїв реалізації якості та функційних вимог;

б) вибір архітектурного зразка, який відповідає архітектурним мотивам; створення або вибір зразка, тактики якого

дозволяють реалізувати ці мотиви; виявлення дочірніх модулів, необхідних для реалізації цих тактик;

в) конкретизація модулів, розподіл функційності з елементів варіанту використання, складання декількох представлень;

г) визначення інтерфейсів дочірніх модулів;

д) перевірка та уточнення елементів варіанту використання у сценаріях якості та накладання, виходячи з них, обмежень на дочірні вузли;

3) вищеперераховані етапи виконати по відношенню до всіх модулів, які потребують подальшої декомпозиції.

Після проектування архітектури та формування робочих груп можна братись до конструювання макету системи. Мета цього етапу в тому, щоб забезпечити можливість реалізації функційності системи в переважному (з точки зору задач проекту) порядку.

В першу чергу, слід реалізувати ті програми, які пов'язані з виконанням та взаємодією між архітектурними компонентами. В системі реального часу це може бути планувальник, в системі на основі правил - процесор правил, в клієнт-серверній системі - механізм координації дій клієнта та сервера. Крім базової інфраструктури передачі даних або взаємодії є зміст ввести в дію ряд найпростіших функцій, які ініціюють механічну поведінку. На цьому етапі з'являється виконувана система - як основа, на яку можна починати нарощувати додаткову функційність.

Далі під впливом декількох факторів (зниження ризику, можливості та спеціалізація персоналу, якомога коротший термін роботи над проектом) обирають, які функційні елементи слід ввести до даної системи. Після рішення про введення до системи чергової порції функційності, слід звернутись до структури використання, яка повідомить про те, які ще програмні засоби системи повинні коректно виконуватись (розвинутись із "заглушки" в повноцінні елементи), щоб реалізація обраних функцій стала можливою. Таким чином у систему можна додавати все нові й нові елементи, поки вони не закінчаться. На будь-якому з таких етапів дії з інтеграції та тестування не складні - легко

виявити джерело несправностей які виникли. Чим менший приріст, тим більш передбачувані бюджет та графік.

3. Документування архітектури

Характер архітектури будь-якої системи обумовлюється висуваними до неї вимогами; це твердження справедливе і по відношенню до документації архітектури, іншими словами, вміст документації залежить від передбачуваних варіантів її застосування. Документація за жодних обставин не може бути універсальною. З одного боку, вона повинна бути абстрактною і доступною для розуміння новими співробітниками, з іншого боку - вельми детальною - настільки, щоб її можна було використовувати як план проведення аналізу. Між архітектурною документацією, призначеною, скажімо, для провдення аналізу безпеки, та архітектурною документацією для вивчення конструкторами повинні існувати істотні відмінності. З іншого боку, у цих варіантів буде мало спільного із вмістом керівництва для ознайомлення, призначеного новому співробітнику.

Архітектурна документація одночасно інструктивна та описова. Обмежуючи діапазон можливих рішень з одного боку і перераховуючи прийняті раніше проектні рішення по системі з іншого боку, вона зобов'язує дотримуватись певних принципів.

З цього слідує, що зацікавлені у складанні документів особи мають вельми різні цілі - від інформації вимагають різної спрямованості, різних рівнів деталізації, різних трактувань. Тому слід прагнути до того, щоб будь-яка зацікавлена особа змогла швидко знайти необхідну інформацію, якомога менше в процесі її пошуку наштовхуючись на незначні (з її точки зору) відомості. В деяких випадках простіше створити декілька документів, призначених для різних зацікавлених осіб. Але, як правило, задача обмежується створенням єдиного комплекту документації з додатковими інструкціями, які повинні полегшити пошук відомостей. Згідно фундаментального правила технічної документації, укладач повинен приймати позицію читача.

Одним з найбільш важливих понять документування програмної архітектури є *представлення (view)* - спосіб фіксації

структури. З поняттям представлення пов'язаний основний принцип документування програмної архітектури. *Документування архітектури* передбачає документування всіх значущих представлень з наступною фіксацією відомостей, які належать одночасно до декількох представлень. Цей принцип корисний тим, що він допомагає розбити проблему документування архітектури на ряд менших елементів:

1) *вибір значущих представлень* - для вибору значущих представлень слід знати зацікавлених осіб та переважні для них способи користування документацією; не менш серйозний вплив на вибір представлень для подальшого документування здійснюють найцінніші для більшості зацікавлених у розробленні системи осіб атрибути якості - наприклад, багаторівневе представлення (layered view) повідомляє відомості про можливість переносу системи, з представлення розташування (deployment view) можна робити висновки про продуктивність та надійність системи; процедура вибору значущих представлень для конкретного проекту складається з таких *трьох кроків*: 1. складання списку можливих представлень у вигляді таблиці, в якій по рядках відображено зацікавлених осіб, а по стовпцях - застосовні до системи представлення, відповідно у комірках відображується ступінь деталізації (довільна, огляд, середня або висока) відомостей про кожне представлення, необхідна тим чи іншим зацікавленим особам; 2. комбінація представлення - для скорочення списку представлень обрати представлення із оглядовою деталізацією або представлення, необхідні обмеженій кількості зацікавлених осіб, та перевірити, чи не можна задовольнити їх іншим, білош універсальним представленням; аналогічно провести пошук представлень, які можна комбінувати - виразити в одному зведеному представленні інформацію з декількох початкових представлень; 3. визначення пріоритетів - представлення, які залишились, повинні відповідати потребам спільноти зацікавлених осіб, тому необхідно визначити, які з цих представлень мають найбільше значення, виходячи з деталей проекту та пріоритету інтересів зацікавлених осіб;

2) *документування представлення* - стандартного шаблону не існує, тому варто користуватись методикою, яка довела свою життєздатність у практичній діяльності - *стандартна семичастинна структура*, яка за допомогою розподілу інформації за окремими розділами допомагає укладачу документації впевнено взятись до вирішення задачі та встановити момент її виконання, що ж стосується читача, то йому стає простіше швидко відшукати потрібну інформацію і пропустити все непотрібне. *Розділи*: 1. первинне відображення (primary presentation) - перелік елементів представлення та встановлені між ними відношення; 2. каталог елементів (element catalog) призначений для більш детального опису елементів та відношень; 3. відображення на контекстній діаграмі відношень зображеної системи до свого оточення із словника представлення; 4. керівництво із зімнюваності (variability guide) - демонструє способи застосування змінюваних параметрів, які входять до складу показаної у даному представленні архітектури; 5. передумови архітектурного рішення (architecture background) - це розділ, в якому міститься обгрунтування відбитого в даному представленні проектного рішення; 6. глосарій термінів (glossary of terms); 7. інша інформація; при документуванні представлення особливу увагу слід приділити документуванню поведінки (поведінкові UML-моделі) та докмуентуванню інтерфейсу (розкадрування, прототипування, моделювання);

3) документування відомостей, які належать до декількох представлень.

4. Аналіз архітектури

Метод аналізу компромісних архітектурних рішень (Architecture Tradeoff Analysis Method, *ATAM*) - це комплексна універсальна методика оцінювання програмної архітектури. Відповідно до назви цей метод виявляє ступінь реалізації в архітектурі тих чи інших задач з якості, а також (виходячи з припущення про те, що будь-яке архітектурне рішення впливає одразу на декілька задач якості) механізм їх взаємодії - іншими словами, їх взаємозамінюваність. Основне призначення ATAM полягає в тому, щоб виявити комерційні задачі, поставлені в

контексті розроблення системи та проектування архітектурри. Разом із участю зацікавлених осіб це допомагає фахівцям з оцінювання сфокусуватись на тих елементах архітектури, які відіграють найважливішу роль для реалізації згаданих задач.

Операції оцінювання за методом ATAM розподіляються на чотири етапи:

0) встановлення партнерських стосунків та підготовка - керівники групи оцінки проводять неофіційні наради із основними відповідальними за проект особами та пропрацьовують подробиці очікуваної роботи; представники проекту вводять фахівців з оцінки в суть проекту, підвищуючи кваліфікацію деяких з них; ці дві групи приймають узгоджені логістичні рішення та узгоджують перелік зацікавлених осіб із ролями, встановлюють терміни та одержувачів зведеного звіту, організовують забезпечення фахівців з оцінки архітектурною документацією, визначають, яку інформацію слід надати на першому етапі;

1) оцінка на основі аналітичних операцій - до початку цього етапу учасники групи оцінки повинні ознайомитись із документацією з архітектури, одержати достатнє уявлення про систему, знати задіяні архітектурні методики та орієнтуватись в найважливіших атрибутах якості; учасники групи оцінки зустрічаються з особами, відповідальними за проект (протягом одного дня);

2) оцінка на основі аналітичних операцій (продовження) - до фахівців приєднуються зацікавлені в архітектурі особи, і протягом приблизно двох днів вони проводять аналітичні заходи сумісно;

3) допрацювання - група оцінки складає в письмовому вигляді та надає одержувачам зведений звіт. По суті, учасники займаються самоперевіркою та вносять у результати своєї роботи різні корективи. На заключній нараді обговорюються успіхи та труднощі. Учасники вивчають звіти, видані їм на першому та другому етапах, слухають витсуп спостерігачів, отже, займаються пошуком шляхів вдосконалення виконання своїх ролей, щоб проводити наступні оцінки з меншими зусиллями та більшою ефективністю.

Чотири етапи ATAM, їх учасники та наближений графік представлені у таблиці 8.4.

Таблиця 8.4 - Етапи ATAM та їх характеристики

Етап	Операції	Учасники	Середня тривалість
0	Встановлення партнерських стосунків та підготовка	Керівництво групи оцінки та основні відповідальні за проект особи	Проходить в неформальній обстановці, згідно конкретної ситуації; може тривати декілька тижнів
1	Оцінка	Група оцінки та відповідальні за проект особи	1 день з наступною перервою тривалістю від 2 до 3 тижнів
2	Оцінка (продовження)	Група оцінки, відповідальні за проект особи, зацікавлені особи	2 дні
3	Допрацювання	Група оцінки та замовник оцінки	1 тиждень

За результатами оцінки програмної системи методом ATAM є ряд документованих артефактів: 1) опис комерційних задач, які визначають успішність системи; 2) набір архітектурних представлень, які документують існуючу або запропоновану архітектуру; 3) дерево корисності, яке виражає декомпозицію задач, які зацікавлені особи ставлять перед архітектурою, - від узагальнених формулювань атрибутів якості до конкретних сценаріїв; 4) ряд виявлених ризиків; 5) ряд точок чутливості (архітектурнрх рішень, які впливають на окремий показник атрибуту якості); 6) ряд точок компромісі (архітектурних рішень, які діють одразу на декілька показників атрибуту якості, причому на один позитивно, а на інші - негативно).

Метод аналізу вартості та ефективності (Cost Benefit Analysis Method, *CBAM*) базується на методі ATAM і забезпечує моделювання витрат та вигод, пов'язаних із прийняттям архітектурно-проектних рішень, та сприяє їх оптимізації. Методом CBAM оцінюються технологічні та економічні фактори, а також власне архітектурні рішення.

Всі програмні архітектори та відповідальні особи прагнуть довести до максимуму різницю між вигодами, одержаними від системи, та вартістю реалізації її проектного рішення. Метод CBAM є продовженням методу ATAM і базується на його артефактах.

Стратегії, застосовувані програмними архітекторами та проектувальниками, повинні бути поставлені в залежність від комерційних задач програмної системи. Прямий економічний фактор - це вартість реалізації системи. Технічними факторами є характеристики системи - іншими словами, атрибути якості. У атрибутів якості також є економічний аспект - вигоди, одержувані від їх реалізації. ATAM допомагає виявити ряд основних архітектурних рішень, значущих в контексті сформульованих зацікавленими особами сценаріїв атрибутів якості, але кожне архітектурне рішення пов'язане із певними побічними ефектами (вартістю). Ці архітектурні рішення призводять до різних вимірюваних рівнів готовності, які мають певну цінність для компанії-розробника системи. ATAM виявляє архітектурні рішення, прийняті відносно розглядуваної системи, та встановлює їх зв'язок із комерційними задачами та кількісною мірою реакції атрибутів якості. Приймаючи ці дані, CBAM допомагає виявити пов'язані з такими рішеннями витрати та вигоди. Базуючись на такій інформації, зацікавлені особи можуть прийняти остаточні рішення щодо забезпечення необхідних атрибутів якості. CBAM не замінює рішень, які приймаються зацікавленими особами. Він лише допомагає їм встановити та документувати витрати та вигоди архітектурних інвестицій, усвідомити невизначеність цього "портфелю". На цій основі запцікавлені особи можуть приймати раціональні рішення, які задовольняють їх потреби та мінімізують ризики.

Метод CBAM виходить з припущення про те, що архітектурні стратегії (як сукупність архітектурних тактик) впливають на атрибути якості системи, а ті, в свою чергу, надають зацкавленим особам деякі вигоди. Ці вигоди називають корисністю (utility). Будь-яка архітектурна стратегія відрізняється тією чи іншою корисністю для зацікавлених осіб. З іншого боку, є побічні ефекти (вартість та час, якй необхідно витратити на реалізацію цієї стратегії). Метод CBAM допомагає зацікавленим особам в процесі вибору архітектурних стратегій, які характеризуються максимальним прибутком на інвестований капітал (return on investment, ROI), тобто найбільш вигдіних з точки зору співвідношення вигод та витрат.

Висновки

Під архітектурою ПЗ найчастіше розуміють набір внутрішніх структур ПЗ, які складаються з компонентів, зв'язків та можливих взаємодій між ними, а також видимих ззовні властивостей цих компонентів. Під компонентом в цьому визначенні мають на увазі досить довільний структурний елемент ПЗ, який можна виділити шляхом визначення інтерфейсу взаємодії між цим компонентом та всім, що його оточує. Тремін "компонент" в розробленні ПЗ найчастіше має дещо інший, більш вузький, зміст - це одиниця збирання системи, її розгортання та конфігураційного керування, те, що не може бути розділене на дрбініші елементи при розгортанні та постачанні системи. Архітектура ПЗ представляє собою набір структур аюбо представлень з різними рівнями абстракції та з різними аспектами, які об'єднані прив'язкою всіх представлених даних до структурних елементів ПЗ. Архітектура важлива, тому що саме вона визначає більшість характеристик якості ПЗ в цілому. Архітектура служить основним засобом спілкування між розробниками, а також і між всіма особами, зацікавленими в даному ПЗ. Вибір архітектури визначає спосіб реалізації вимог на високому рівні абстракції. Саме архітектура майже повністю визначає такі характеристики ПЗ як надійність, можливість переному, зручність супроводу. Архітектура значно впливає і на зручність використання та ефективність ПЗ, які

визначаються також і реалізацією окремих компонентів. Значно менший вплив архітектура має на функційність - як правило, для реалізації заданої функційності можна використовувати різні архітектури. Тому вибір між тією чи іншою архітектурою визначається перш за все саме нефункційними вимогами та необхідними властивостями ПЗ з позицій зручності супроводу та можливості переносу. При цьому для побудови якісної архітектури потрібно враховувати можливі протиріччя між вимогами до різних характеристик та вміти обирати компромісні рішення, які надають прийнятні значення за всіма показниками. Так, для підвищення ефективності вигідніше використовувати монолітні архітектури, в яких виділено невелику кількість компонентів (економія пам'яті завдяки мінімальній кількості компонентів з власними даними, скорочення часу роботи). Для підвищення зручності супроводу навпаки краще розбивати систему на велику кількість окремих компонентів, щоб кожен з них вирішував невелику, але чітко визначену частину загальної задачі. Для підвищення надійності краще використовувати дублювання функцій, тобто зробити декілька компонентів відповідальними за вирішення однієї підзадачі, причому, оскільки помилки ПЗ невипадкові, краще використовувати досить різні рішення однієї й тієї ж задачі в різних компонентах. Список стандартів, які регламентують опис архітектури та проектну документацію:

- IEEE 1016-1998 Recommended Practice for Software Design Descriptions;
- IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.

Лекція №9
УПРАВЛІННЯ РИЗИКАМИ ПРИ РОЗРОБЛЕННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Поняття ризику
2. Управління ризиками. Метод PERT-аналізу

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.

4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

5. Ю. Ю. Якунин. Технологии разработки программного обеспечения - Красноярск: ИПК СФУ, 2008

6. Ф.А.Новиков, Э.А.Опалева, Е.О.Степанов. Учебно-методическое пособие по дисциплине «Управление проектами и разработкой ПО» // [Електронний ресурс] - Режим доступу: http://window.edu.ru/resource/366/60366/files/itmo305.pdf

7. В.В.Липаев. Анализ и сокращение рисков проектов сложных программных средств – М.:Синтег, 2005. – 224 с.

Вступ

Наразі ефективність та результативність процесу розроблення ПЗ в цілому залишає бажати кращого. Незважаючи на помітні успіхи технології програмування, залишається значна частина програмних проектів, які не можна вважати цілком успішними. Поряд з ефективними досягненнями є й численні невдачі. На жаль, і досі надто часто доводиться робити висновок,

що програмування – це ризикований бізнес, програми - ненадійні, а програмісти – некеровані.

Так, дослідження «Хаос», проведене Standish Group, отримало наступні результати програмних проектів:

- середнє перевищення часу розроблення – 222%;
- середнє перевищення витрат на розроблення – 189%;
- відставання для задоволення вимог користувачів – на 27 місяців.

Дослідження «Хаос» також виявило, що лише 25% програмних проектів успішно впроваджені, 28% проектів призупинені і в 46% проектів виникли значні проблеми з постачанням.

Очевидно, що розроблення ПЗ не завжди завершується успішно і часто пов'язане із затримками або перевищенням бюджету. Тому недивно, що менеджери проектів прагнуть покращити управління ризиками при розробленні ПЗ.

В життєвому циклі ПЗ не завжди вдається досягнути необхідного позитивного ефекту, і може проявлятись деякий збиток – ризик у створюваних програмних системах. Ризики характеризують можливі негативні наслідки або збитки при функціонуванні ПЗ. Задача розробників зводиться до скорочення та ліквідації ризиків. Зниження ризиків проекту сприяє підвищенню його якості.

1. Поняття ризику

Ризиками називають негативні події ймовірнісного характеру, які негативно впливають на результат проекту; негативні події та їх величини, які відбивають втрати або збитки від процесів або продуктів, викликані дефектами при проектуванні вимог, недоліками обґрунтування проектів ПЗ, а також при наступних етапах розроблення, реалізації та всього ЖЦ ПЗ. Ризики проявляються як негативні наслідки функціонування або порушення безпеки використання ПЗ в результаті відхилення характеристик об'єктів або процесів від заданих вимог замовника, здатні завдати збитків системі, зовнішньому середовищу або користувачу (наприклад, втрата системи, втрати налсдіків

діяльності особи або колективу, персональний збиток або поява юридичної відповідальності за негативні результати проекту). Важливо розуміти, що *ризик – це ймовірна подія, яка може відбутись, а може й не відбутись*. Якщо негативна подія обов'язково відбудеться, то це вже не ризик, а постійно діючий фактор. Наприклад, «хвороба або звільнення менеджера проекту» - це ризик, а «відсутність досвіду розроблення у персоналу» - це фактор.

Проблема дослідження та скорочення ризиків функціонування ПЗ та систем в процесі розроблення та ЖЦ виникла і розвивається внаслідок зростання розмаїття, складності та відповідальності задач їх використання. *Причинами виникнення та прояву ризиків* можуть бути: зловмисні, активні впливи зацікавлених осіб або випадкові негативні прояви дефектів зовнішнього середовища, системи, дій розробників або користувачів. В першому випадку ризики можуть бути обумовлені спотвореннями програм та інформаційних ресурсів і їх вразливістю від навмисних, зовнішніх впливів (атак) з метою незаконного використання або змін інформації та програм. Для вирішення цієї проблеми створені та активно розвиваються методи, засоби та стандарти забезпечення інформаційної безпеки та захисту програм і даних від навмисних негативних зовнішніх впливів, які є предметом вивчення інших навчальних дисциплін і в цій лекції не розглядаються. Ризики при випадкових негативних впливах дефектів за відсутності зловмисного впливу на системи істотно відрізняються від попередніх задач. Ці ризики об'єктів та систем залежать від ситуацій відмов, які відбиваються на робото здатності та безпеці реалізації їх основних функцій, причинами яких можуть бути дефекти та аномалії в апаратурі, програмах, даних або обчислювальних процесах. При цьому істотно спотворюється процес функціонування систем, що може завдавати значних збитків при застосуванні систем. Основними джерелами ситуацій відмов є некоректні початкові вимоги при проектуванні, збої та відмови апаратури, дефекти або помилки в програмах і даних: за даними тестування відкритого ПЗ компанією Coverity, ПЗ містить в середньому 434 дефекти на один мільйон рядків коду, а для дуже

поширених, якісно протестованих та відлагоджених продуктів ця цифра складає 290 дефектів на один мільйон рядків коду. Наразі не існує методів, які дозволяють гарантувати відсутність дефектів ані у специфікаціях, ані безпосередньо в програмах, ані в експлуатаційній документації. З точки зору кінцевого користувача, прояви дефектів ПЗ можуть представляти собою від тимчасових незручностей до техногенних катастроф. В реальних складних системах можливі катастрофічні наслідки та відмови функціонування з великими збитками, які можуть за результатами перевищувати наслідки зловмисних впливів, тому такі ризики вимагають самостійного вивчення та адекватних методів і засобів скорочення. Розглянуті ризики можуть бути обумовлені порушеннями технологій або обмежень при використанні ресурсів, виділених на розроблення ПЗ.

Взагалі можна виділити наступні *типові важливі причини, які призводять до виникнення рИзикових ситуацій* другого типу у програмних проектах:

- нереальна оцінка потрібного часу реалізації проекту та виділеного бюджету;
- нереальна оцінка можливостей колективу розробників;
- недостатня кількість та кваліфікація колективу розробників;
- недостатнє володіння інструментарієм розробниками;
- помилки визначення вимог до розроблюваного ПЗ (в т.ч. недостатня деталізація вимог);
- порушення основних правил процесів розроблення (наприклад, порушення в управління версіями, які призводять до втрати версій);
- неперервна зміна вимог до розроблюваного ПЗ в ході проекту;
- істотна зміна ринкової ситуації, яка робить беззмістовним слідування початковим планам (наприклад, поява на ринку доступного ПЗ, яке перевершує за можливостями розроблюване);

- неперервна зміна «правил гри» в колективі розробників або групі проекту (правил комунікації, розподілу відповідальності, розподілу обов'язків);
- помилки проектування архітектури ПЗ;
- помилки розроблення;
- помилки інтеграції;
- недоліки зовнішнього обслуговування;
- технічні та програмні збої.

В ЖЦ ПЗ можна виділити *три класи ризиків*:

1) недоліки і дефекти функційної придатності - спотворення або неповна реалізація потрібного призначення, функцій або взаємодії ПЗ з компонентами системи або зовнішнього середовища;

2) недостатні і невідповідні вимогам реалізації конструктивних характеристик якості ПЗ при його функціонуванні та використанні за прямим призначенням;

3) порушення обмежень на використання економічних, часових або технічних ресурсів при створенні та використанні ПЗ.

Проблеми аналізу та оцінювання ризиків ПЗ можуть розглядатись з промислової та практичної точок зору. *Промислова позиція* передбачає, що управління ризиками ПЗ є інженерною дисципліною, комплекси програм завжди містять ризики, які апріорі неможливо достовірно передбачити і контролювати, але вони іноді катастрофічно відбиваються на якості функціонування систем або зовнішнього середовища. *Практична точка зору* передбачає, що існує недостатнє розуміння замовниками, розробниками і користувачами значення та необхідності аналізу і скорочення ризиків в ЖЦ складних програмних систем, недостатньо використовуються технологічні дисципліни та інструментальні засоби для управління і зменшення ризиків при створенні та використанні ПЗ.

Першою характеристикою ризику є ймовірність того, що ризикова подія відбудеться. Ймовірності прийнято задавати числом від 0 до 1 або у відсотках.

Другою важливою характеристикою ризику є збиток внаслідок реалізації ризику. Ця величина визначає, наскільки сильно постраждає проект, якщо ризикова подія відбудеться.

Наприклад, ризик «хвороба або звільнення менеджера проекту» без сумніву завдасть шкоди проекту, але якщо процес розроблення організовано вірно, то реалізація цього ризику не є катастрофою для проекту. В той же час ризик «втрата всіх даних проекту» може стати катастрофою, якщо ця подія відбудеться напередодні випуску першої версії продукту.

Існують різні способи задання величини збитку. Найбільш простий і зрозумілий – вказати долю загального бюджету проекта, яка буде незворотньо втрачена, якщо ризикова подія відбудеться. В такому випадку збиток також можна задавати числом від 0 до1 або у відсотках. Дві введені величини, ймовірність та збиток, дозволяють порівнювати ризики між собою. Як правило, в якості *величини ризику* обирається математичне сподівання збитку.

Наприклад, нехай ймовірність ризикової події «хвороба або звільнення менеджера проекту» складає 0,1 (10%) і збиток також складає 0.1 (10%). Тоді величина ризику складає 0.01 (1%). Нехай ймовірність ризикової події «втрата всіх даних проекту» - 0.01 (1%), а збиток – 1 (100%), тоді величина ризику також складає 0.01 (1%). Ці ризики рівні за величиною. Якщо ймовірність ризикової події «недостатні навички програмування» рівна 0.1 (10%), а збиток – 0.5 (50%), то величина такого ризику – 0.05 (5%), він істотно більший за попередні.

Ризики поділяються на:

- *усувні* – такі, яких за допомогою спеціальних заходів можна уникнути або яких можна запобігти;
- *неусувні* – такі, повпливати на які менеджер не може.

Наприклад, ризик «втрата всіх даних проекту» можна практично подолати за допомогою проведення резервного копіювання всіх даних щодня і зберігання резервних копій у сейфах трьох різних банків. Ризику ж «припинення фінансування проекту в результаті банкрутства інвестора» менеджер проекту, як правило, запобігти не може.

Різниця між подоланням та запобіганням (попередженням, уникненням) ризику полягає в наступному:

- *подолання ризику* – це проведення заходів, в результаті яких ймовірність ризикової події зводиться до нуля;

- *запобігання ризику* – це проведення заходів, в результаті яких збиток ризику зводиться до нуля.

В обох випадках величина ризику зводиться до нуля, тому подолання і запобігання ризику разом називають *стратегіями усунення ризику*. Наприклад, ризик «втрата всіх даних проекту» можна подолати за допомогою резервного копіювання, а ризику «хвороба або звільнення менеджера проекту» можна запобігти за допомогою завчасної підготовки і введення в курс справ резервного менджера.

Для усувних ризиків дуже важливою характеристикою є вартість заходів з подолання або запобагння ризиків, яку також можна задавати числом від 0 до 1 або у відсотках. Наприклад, вартість подолання ризику «втрата всіх даних проекту» можна оцінити в 1%, а вартість запобігання ризику «хвороба або звільнення менеджера проекту» більш реалістично оцінити в 5%.

Можливі ризики програмних проектів представлені у таблиці 9.1.

Таблиця 9.1 – Можливі ризики програмних проектів

Ризик	**Типи ризику**	**Опис ризику**
Плинність розробників	Ризик для проекту	Досвідчені розробники полишають проект до його завершення
Зміна в керуванні організацією	Ризик для проекту	Організація змінює свої пріоритети в керуванні проектом
Неготовність апаратних засобів	Ризик для проекту	Апаратні засоби, необхідні для проекту, не надійшли вчасно або не готові до експлуатації
Зміна вимог	Ризик для проекту та для розроблюваного продукту	Поява великої кількості непередбачуваних змін у вимогах, які висуваються до розроблюваного ПЗ

Затримка у розробленні специфікації	Ризик для проекту та для розроблюваного продукту	Специфікації основних інтерфейсів підсистем не надійшли до розробників відповідно до графіку робіт
Недооцінка розміру розроблюваної системи	Ризик для проекту та для розроблюваного продукту	Розмір системи значно перевищив початкову оцінку
Недостатня ефективність CASE-засобів	Ризик для розроблюваного продукту	CASE-засоби, призначені для підтримки проекту, виявились менш ефективними, ніж очікувалось
Зміни в технології розроблення ПЗ	Бізнес-ризик	Основні технології побудови програмної системи замінюються новими
Поява конкуруючого програмного продукту	Бізнес-ризик	На ринку програмних продуктів до завершення проекту з'явилась конкуруюча програмна система

Для успішної реалізації проектів однією з основ управління проектом повинне бути управління ризиками. Воно представлене як одна з 9 основних галузей знань в галузі управління проектами, описаних PMI (Американським інститутом управління проектами).

2. Управління ризиками. Метод PERT-аналізу

Управління ризиками починається з дослідження понять, які сприяють схваленню програмного проекта. Кваліфікований менеджер проекту повинен добре справлятись з оцінкою та управлінням ризиками, які виникають при розробленні ПЗ. Управління ризиками охоплює весь життєвий цикл розроблення. Аналіз ризиків, які виникають, та неперервне планування ведеться постійно на всіх етапах ЖЦ ПЗ. Ризики аналізуються, визначаються їх пріоритети з врахуванням тижневого графіку робіт, причому поточний список ризиків з «першої десятки» оголошується на щотижневих зборах, де розглядається поточний стан проекту.

Єдина можливість пом'якшення ступеня впливу ризиків полягає у їх обробці учасниками колективу проекта.

Управління ризиками – це процес прийняття та виконання управлінських рішень, спрямованих на зниження ймовірності виникнення несприятливого результату та мінімізацію можливих втрат, викликаних його реалізацією; це систематичні процеси, пов'язані з ідентифікацією, аналізом та прийняттям рішень, які забезпечують мінімізацію негативних наслідків настання ризикових подій, а також максимізацію ймовірності та наслідків настання позитивних подій. Управління ризиками включає повне розуміння внутрішніх та зовнішніх причин, які впливають на проект і можуть призвести до його зриву. Аналіз ризиків виконується після формування плану проекта. Головною метою управління ризиками є ідентифікація та контроль за факторами, що рідко зустрічаються і призводять до варіацій проекту.

Процес управління ризиками включає:

- планування управління ризиками – планування діяльності по управлінню ризиками проекту, включаючи набір методів, засобів та організації управління ризиками;

- ідентифікацію факторів ризику – визначення ризиків, здатних впливати на проект, і документування їхніх характеристик;

- оцінку ризиків – якісний та кількісний аналіз ризиків з метою визначення їх впливу на проект;

- планування реагування на ризики – розроблення мір, які забезпечують мінімізацію ймовірності та послаблення негативних наслідків ризикових подій при загальному підвищенні ймовірності успішного завершення проекту;

- моніторинг та контроль ризиків – моніторинг настання ризикових подій, визначення нових ризиків, виконання плану управління ризиками проекту та оцінка ефективності дій з мінімізації ризиків.

Процес управління ризиками представлено на рис.9.1.

Загальні моделі аналізу ризиків в складних програмних системах регламентовані стандартами, зокрема, стандартом ГОСТ Р 51901 «Управління надійністю. Аналіз ризику технологічних систем». Основною задачею стандарту є обґрунтування рішень, які

стосуються аналізу ризику реалізації проектів та технологій складних систем. Хоча стандарт і не спрямований виключно на проекти розроблення ПЗ, але викладені в ньому рекомендації можуть бути застосовані і в таких проектах. Управління ризиками в ЖЦ програмних проектів спеціально регламентоване міжнародними стандартами ISO 12207 «Процеси життєвого циклу програмних засобів» та ISO 15504 «Оцінка та атестація зрілості процесів створення і супроводу програмних засобів та інформаційних систем», які доцільно використовувати при розробленні комплексів програм. Стандарт ISO 15504 містить окремий розділ «Процес управління ризиками», призначенням якого є регламентування та планування процесів виявлення і усунення ризиків протягом всього ЖЦ ПЗ.

Рис.9.1 – Управління ризиками

Software Engineering Institute (SEI) розробив *модель управління ризиками при розробленні ПЗ*, яка включає в себе як вимоги згаданих стандартів, так і відомі «кращі практики» (best practices) управління ризиками. Підготовку управління ризиками проекту ця модель рекомендує проводити за наступною послідовністю:

- постановка цілей управління ризиками відповідно до цілей проекту;

- планування та координація робіт з оцінки ризиків проекту;

- виділення групи експертів для оцінки ризиків, підготовка та планування інтерв'ю з виконавцями з метою виявлення ризиків в результатих їх діяльності;

- ідентифікація та оцінка ризиків (виявлення можливих ризикових ситуацій та оцінювання впливу ризиків на проект);

- підготовка до усунення ризиків (розроблення заходів із скорочення та усунення ризиків, а також підготовка звіту з рекомендаціями для керівництва проекту з аналізу і управління ризиками);

- розроблення *плану управління ризиками*, який включає: списки ризиків за етапами або роботами проекту; методи, процеси та інструменти, застосовувані для скорочення або усунення ризиків; організацію та розподіл відповідальності за управління ризиками, а також за забезпечення припустимого рівня ризиків проекту.

Виконується план управління ризиками, як правило, ітераційно за основними крупними етапами ЖЦ проекту. Звичайний *порядок дій згідно моделі SEI* при цьому наступний:

1) ідентифікація ризиків: визначення можливих джерел ризиків; ідентифікація потенційних ризикових подій; визначення ситуацій та умов їх виникнення; визначення можливих збитків;

2) аналіз ризиків – визначення якісних, а краще – кількісних характеристик ризику, ранжування ризиків за пріоритетами;

3) планування відповідей на ризики – дій із зменшення або усунення ризику;

4) відстежування ризиків – контроль поточних ризиків, коригування планів із зниження ризику, перегляд результатів виконаних заходів плану управління ризиками;

5) підготовка та реалізація планів подальшого зниження та ліквідації ризиків проекту.

Схему процесу управління ризиками можна представити також в наступному вигляді (рис.9.2).

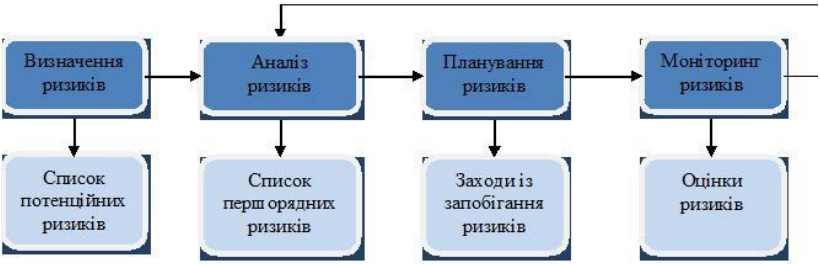


Рис.9.2 - Схема процесу управління ризиками

Існує ще одна *модель аналізу ризиків ПЗ*, яка базується на великих елементах, факторах та властивостях - метриках ризиків. До елементів ризиків включені взаємопов'язані технічні, вартісні та планові ризики, а також ризики процесів та процедур управління.

Технічні ризики визначаються вимогами та особливосятми об'єкту (ПЗ) і включають:

- функційні характеристики;
- характеристики якості;
- надійність;
- застосовність;
- часову продуктивність;
- супроводжуваність;
- повторне використання компонентів.

Вартісні ризики складають:

- обмеження сумарного бюджету;
- недоступна, фіксована або варіативна вартість проекту;
- ступінь реалізму пр оцінюванні витрат на проект.

Планові ризики складають:

- властивості та можливості гнучкості зміни планів;
- можливості порушення встановлених термінів етапів ЖЦ;
- реалізм планів та етапів ЖЦ.

До *ризиків процесів та процедур управління проектом* варто віднести:

- ризики ідентифікації;
- ризики стратегії та планування проекту;

- ризики оцінок;

- припустимі результуючі ризики про скороченні або усуненні загроз;

- ризики документування;

- ризики прогнозування, розвитку та вдосконалення проекту.

Перш ніж із ризиком можна щось зробити, він повинен бути ідентифікований. *Ідентифікація ризику* полягає в фіксації всіх факторів занепокоєння та стурбованості, пов'язаних із проектом, а потім в постійному обмірковуванні всім колективом інших можливих побоювань. Для виявлення ризиків необхідний скептичний склад розуму. Ідентифікація ризику подібна до проведення інспектування – глобального пошуку дефектів у плані розроблення. Для ідентифікації ризиків корисно мати контрольний список (checklist) ризиків схожих проектів. Фахівцями з управління ризиками в результаті досліджень були запропоновані наступні *глобальні категорії ризиків*, характерні для найпростіших проектів розроблення ПЗ:

- недостатнє залучення в проект керівництва;

- неможливість залучення реальних та адекватних користувачів;

- нерозуміння вимог розробниками;

- зміна області застосування або цілей проекту;

- нестача знань або навичок у персоналу.

Варто зауважити, що в цьому списку *лише 20% ризиків належать до технічних, а решта 80% - до організаційних*. Таке співвідношення відображає реальний стан справ. Ризики, які виникають через погану організацію проекту, набагато перевищують ризики, які виникають через вибір неадекватного інструменту, недостатнього досвіду програмування, використання нової інформаційної технології і т.п. Таким чином, основне навантаження з управління ризиками лягає на менеджера проекту. Кожний ідентифікований ризик повинен радісно сприйматись колективом проекту, оскільки в цьому випадку з ним можна почати щось робити. *Справжньою проблемою є ризики, які не вдалось ідентифікувати!* Оскільки відсоток проектів, які вже ніколи не

завершаться, досить великий, то лише постійна увага до ризиків робить більш ймовірним факт, що проект потрапить до 20% вдалих проектів або буде припинений до того, як будуть витрачені величезні кошти.

Менеджер проекту має справу із ризиками, які можна класифікувати наступним чином:

- *відоме у відомому* – ризики відомі розробникам проекту, визначені категорії ризику, а також реалістичні оцінки ймовірності та збитку конкретних ризиків для даного проекту. Наприклад, якщо відсутнім є досить кваліфікований виконавець крупного розділу проекта, то ризик відноситься до відомого типу, а відносно його наявності в даному проекті також можна зробити певні висновки;

- *відоме у невідомому* – ризики відомі колективу розробників проектів, знайома категорія ризику, але невідомі реальна ймовірність прояву або можлива величина збитку для даного проекту. Наприклад, відсутність достатньої взаємодії з кінцевими користувачами призводить до ризику, пов'язаному із некоректними формулюванням та ідентифікацією вимог. Однак може бути невідомо, чи існує взагалі цей ризик і чи може він завдати збитку для даного проекту;

- *невідоме у невідомому* – ризики можуть бути відомі розробникам проекту, знайома категорія ризику, але невідомі ані ймовірність, ані збиток, а отже, невідома його реальна величина для даного програмного проекту. Наприклад, подібні ризики проявляються у тому випадку, коли проект використовує специфічне технологічне рішення, вказане у вимогах контракту, але незнайоме менеджеру проекта та колективу проекта. За відсутності досвіду роботи із інструментом, менеджер проекту не може знати всіх потенційних ризиків, які може викликати його використання.

Типи та ознаки ризиків наведено у таблиці 9.2. Приклади ідентифікованих ризиків програмного проекту наведено у таблиці 9.3.

Таблиця 9.2 – Типи та ознаки ризиків

Тип ризиків	**Ознаки ризиків**
Технологічні ризики	Затримки у постачанні обладнання або програмних засобів підтримки процесу створення ПЗ, чисельні документовані технологічні проблеми
Ризики, пов'язані з персоналом	Низький моральний стан персоналу, натягнуті стосунки між членами колективу розробників, низька якість виконаної роботи
Організаційні ризики	Розмови серед персоналу про пасивність та недостатню компетентність вищого керівництва організації
Інструментальні ризики	Небажання розробників використовувати програмні засоби підтримки, несхвальні відгуки про CASE-засоби, запити на більш потужні інструментальні засоби
Ризики, пов'язані з системними вимогами	Необхідність перегляду багатьох системних вимог, незадоволеність замовника ПЗ
Ризики оцінювання	Зміни графіку робіт, чисельні звіти про порушення графіку робіт

Таблиця 9.3 – Приклади ідентифікованих ризиків

Категорія ризиків	**Приклади ризиків**
Технологічні ризики	- База даних, яка використовується у програмній системі, не забезпечує оброблення очікуваного обсягу транзакцій - Програмні компоненти, які використовуються повторно, мають дефекти, які обмежують їх функційні можливості
Ризики, пов'язані з персоналом	- Неможливо підібрати робітників з потрібним професійним рівнем - Провідний розробник захворів в найкритичніший час - Неможливо організувати необхідне навчання персоналу

Організаційні ризики	- В організації, яка виконує розроблення ПЗ, відбулась реорганізація, в результаті чого змінились пріоритети в управлінні проектом - Фінансові ускладнення в організації призвели до зменшення бюджету проекта
Інструментальні ризики	- Програмний код, генерований CASE-засобами, не ефективний - CASE-засоби неможливо інтегрувати з іншими засобами підтримки проекту
Ризики, пов'язані із системними вимогами	- Зміни вимог призводять до значних повторних робіт з проектування системи - Початкове нечітке формулювання вимог користувача призвело до значних змін системних вимог, які проявились на пізніх стадіях розроблення проекту
Ризики оцінювання	- Недооцінки часу виконання проекту - Швидкість виявлення дефектів в системі нижче раніше запланованої - Розмір системи значно перевищує початково розрахований

Ідентифікований ризик, величина якого може бути досить надійно оцінена, природньо називати виявленим. *Виявлення ризику* – це оцінка його величини. Виявлення ризиків – доволі складна задача, оскільки вона нерозривно пов'язана із задачею аналізу – оцінки кількісного значення величин за умов невизначеності. Методи, які використовуються для оцінки величини ризику, як правило, є кількісними. Однак повний кількісний аналіз не завжди можливий через брак інформації про систему або діяльність, які підлягає аналізу. За таких обставин може виявитись ефективним порівняльне кількісне або якісне ранжування ризику фахівцями, інформованими у даній галузі. При якісному ранжуванні необхідно мати чітке пояснення всіх використовуваних термінів та зафіксоване обґрунтування всіх класифікацій ймовірностей та збитків. У випадку проведення кількісної оцінки величини ризику, необхідно враховувати, що розрахункові значення ризику є наближеними оцінками, тому слід подбати, щоб їх точність

відповідала точності використовуваних вхідних даних та аналітичних методів.

Елементи оцінки величини ризику є спільними для всіх видів ризиків. Перш за все, аналізуються можливі причини негативної події з метою визначення частоти виникнення таких подій, їх тривалості та характеру. В процесі аналізу може виникнути необхідність визначення оцінки ймовірності небезпеки. *Аналіз ймовірності використовується для оцінки ймовірності кожної негативної події, яка проявилась на стадії ідентифікації небезпек, загроз.* Для оцінки частот подій використовуються наступні методи: використання наявних статистичних даних (передісторій), одержання частот негативних подій на основі аналітичних або імітаційних методів; використання експертних оцінок. *Аналіз збитків* використовується для оцінки ймовірного впливу, викликаного небажаною подією. Аналіз збитків повинен описувати і оцінювати будь-які наслідки негативних подій; враховувати заходи, спрямовані на пом'якшення наслідків; розглядати та враховувати як негайні наслідки, так і ті, які можуть відбутись через певний час, а також ті, які є вторинними. У таблиці 9.4 приведено список ризиків після проведення їх аналізу.

Таблиця 9.4 - Список ризиків після аналізу

Ризик	**Ймовірність***	**Ступінь збитків**
Фінансові ускладнення в організації призвели до зменшення бюджету проекта	Низька	Катастрофічний
Неможливо підібрати робітників з потрібним професійним рівнем	Висока	Катастрофічний
Провідний розробник захворів в найкритичніший час	Середня	Серйозний
Програмні компоненти, використовувані повторно, мають дефекти, які обмежують їх функційні можливості	Середня	Серйозний

Зміни вимог призводять до значних повторних робіт з проектування системи	Середня	Серйозний
В організації, яка виконує розроблення ПЗ, відбулась реорганізація, в результаті чого змінились пріоритети в керуванні проектом	Висока	Серйозний
База даних, яка використовується в програмній системі, не забезпечує оброблення очікуваного обсягу транзакцій	Середня	Серйозний
Недооцінки часу виконання проекту	Висока	Серйозний
CASE-засоби неможливо інтегрувати з іншими засобами підтримки проекту	Висока	Припустимий
Початкове нечітке формулювання вимог користувача призвело до значних змін системних вимог, які проявились на пізніх стадіях розроблення проекту	Середня	Припустимий
Неможливо організувати необхідне навчання персоналу	Середня	Припустимий
Швидкість виявлення дефектів в системі нижче раніше запланованої	Середня	Припустимий
Розмір системи значно перевищує початково розрахований	Висока	Припустимий
Програмний код, генерований CASE-засобами, неефективний	Середня	Незначний

* Ймовірність ризику вважається дуже низькою, якщо вона має значення менше 10%, низькою – 10-25%, середньою – 25-50%, високою – 50-75%, дуже високою – більше 75%.

Існує багато невизначеностей, пов’язаних з оцінкою ризику. Розуміння невизначеностей та їхніх причин необхідне для ефективної інтерпретації значень величини ризику. Аналіз

невизначеностей повинен передбачати визначення змін та неточностей в результатх моделювання, які є наслідком відхилення параметрів та припущень, застосовуваних при побудові моделі. У випадку, коли обґрунтовану оцінку ймовірності ризиків одержати важко (що є швидше правилом, аніж виключенням) корисно виконувати вимог стандарту ISO 15504, згідно якого слід визначити метрики, які відбивають зміни у стані (ймовірності, збитках, часовому діапазоні) ризиків залежно від діяльності з їх усунення. Спектр *методів кількісного аналізу невизначеностей та ризиків* досить широкий: від PERT-аналізу та аналізу «що-якщо» (what-if) до складного імітаційного моделювання методом Монте-Карло. Для цих методів існує спеціальне ПЗ – наприклад, Open Plan Professional дозволяє охарактеризувати невизначеності, пов'язані із тривалістю проекту (гістограма розподілу дати можливого завершення проекту, побудована за допомогою Open Plan Professional, наведена на рис.9.3).

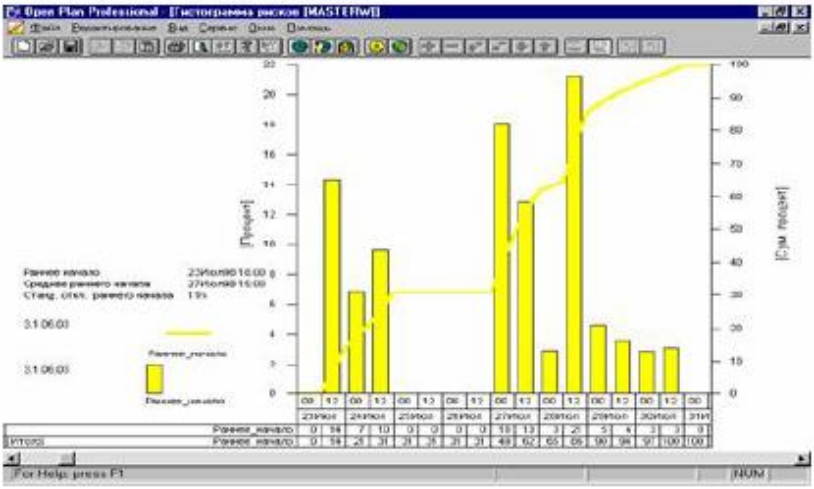


Рис.9.3 - Гістограма розподілу дати можливого завершення проекту (Open Plan Professional)

Серед кількісних методів оцінки невизначеностей та їх впливу на проекти популярним є *метод PERT-аналізу* (Project Evaluation and Review Technigue). Його суть полягає в тому, що для

кожної роботи проекту вказуються три оцінки її тривалості (трудомісткості, вартості) – оптимістична (*О*), найбільш імовірна (*І*) та песимістична (*П*). Одержані оцінки усереднюються із заданими ваговими коефіцієнтами, даючи можливість аналізувати побудовану таким чином мережеву діаграму, яка містить очікувані тривалості (трудомісткості, вартості). Стандартна формула усереднення: *(О+4*І+П)/6*, однак за необхідності можна змінити вагові коефіцієнти. Щодо оцінювання ризиків за таким методом, то виконується оптимістична, імовірна та песимістична оцінки тривалості (трудомісткості, вартості) проекту за наявності кожного ідентифікованого ризику, після чого виконується усереднення одержаних оцінок для кожного ризику за стандартною формулою усереднення. Цей аналіз дозволить визначити пріоритети для кожного з ризиків. Такий аналіз автоматизує MS Project Professional.

Одним із найпростіших методів підведення підсумків результатів аналізу є ранжування ризиків за пріоритетами, оскільки засобів та часу для усунення всіх ризиків, як правило, не вистачає. Якщо оцінки ймовірності ризику *p*, збитків *d*, вартості усунення *c*, то пріоритет *q* ризику можна визначати за формулою:

q=(1-p)(1-d)*c*.

Перевірка результатів аналізу повинна здійснюватись експертами, не залученими до участі у виконанні проекту. Перевірка повинна включати наступні етапи: перевірка відповідності області застосування аналізу поставленим задачам; перевірка всіх важливих припущень на їх правдоподібність; підтвердження аналітиком вірності використовуваних методів, моделей і даних; перевірка результатів аналізу на повторюваність; перевірка результатів аналізу на стійкість до різних форматів даних.

Методи зниження ризику - це методи, спрямовані на зменшення величини ризику, тобто або методи зниження ймовірності виникнення небажаної події, або методи зниження величини очікуваних збитків, або те й інше разом.

На практиці рідко виникає така сприятлива ситуація, коли можна повністю усунути ризик, тобто знизити величину ризика до

0. Як правило, проводяться лише заходи, які знижують ризики до прийнятної величини, але не усувають ризик повністю. При цьому потрібно враховувати вартість заходів зі зменшення або усунення ризику. Як правило, керуються наступним прости правилом. Нехай оцінки ймовірності та збитків ризику без проведення заходу із зниження ризику дорівнюють *p1* і *d1*, з проведенням заходу - *p2* і *d2* відповідно, а вартість заходу із зменшення ризику - *c*. Тоді такий захід має сенс, якщо *p1*d1>=p2*d2+c.*

Наприклад, ймовірність ризикиової події "недостатні навички програмування" складає 10%, збитки - 50%, варість заходу із скорочення ризику "попереднє навчання" - 1%, а ймовірність та збитки в разі навчання складають 5% і 20% відповідно. Тоді: 0.1*0.5=0.05 (5%), 0.05*0.2+0.01=0.02 (2%), 5%>2%, отже, навчання має сенс.

Заходи, спрямовані на зниження ризику, можуть бути найрізнішими. Вони можуть усувати первинні причини, які викликали появу ризикових ситуацій, або зменшувати ймовірність та/або збитки від ризикових подій. Наприклад:

- ризик появи нових вимог в процесі роботи над проектом можна зменшити узгодженням детального переліку вимог із замовником, включенням цього списку в договір і точним слідуванням цим вимогам;
- ризик істотної зміни ринкової ситуації, яка робить виконання початкового плану беззмістовним, знижується попереднім дослідження ринку, експертною оцінкою та/або консультацією досвідченого сторннього консультанта;
- ризик недостатніх навичок володіння виконавцями інструментами розроблення можна зменшити спеціальним тренінгом;
- …

Не варто нехтувати одним важливим способом мінімізації ризиків - укладанням договору страхування, який повністю або частково покриває можливі збитки. Так можна зменшити ризики, які є "зовнішніми" по відношенню до компанії-розробника (наприклад, ризики зриву поставок або платежів, ризики зміни ринкової ситуації і т.і.) При виконанні проектів менеджер постійно

накопичує досвід з управління ризиками, і за наявності такого досвіду може створити для себе "шаблони" планів відповіді на ризики - списки можливих ризиків та дії з їх зменшення.

Планування ризиків полягає у визначенні стратегії управління кожним значущим ризиком, відібраним для моніторингу після аналізу ризиків. Приклад планування (вибору стратегії управління) ризиків представлено у таблиці 9.5.

Таблиця 9.5 - Приклад планування ризиків

Ризик	**Стратегія**
Фінансові проблеми організації	Підготувати короткий документ для керівництва організації, який показує важливість даного проекту для досягнення фінансових цілей організації
Проблеми некваліфікованого персоналу	Попередити замовника про потенційні труднощі та можливу затримку проекту, розглянути питання про купівлю компонентів системи
Хвороби персоналу	Реорганізувати роботу колективу розробників таким чином, щоб обов’язки та робота членів колективу перекривали один одного, внаслідок цього розробники будуть знати та розуміти задачі, виконувані іншими співробітниками
Дефектні системні компоненти	Замінити потенційно дефектні системні компоненти купленими компонентами, які гарантують якість роботи
Реорганізація компанії-розробника	Підготувати короткий документ для керівництва компанії, який показує важливість даного проекту для досягнення фінансових цілей компанії
Недостатня продуктивність бази даних	Розглянути можливість купівлі більш продуктивної бази даних
Недооцінки часу виконання проекту	Розглянути питання про купівлю системних компонентів, дослідити можливість використання генератора програмного коду

Детальний план управління ризиками проекту представляє собою таблицю із вказанням:

1) виявлених ризиків для кожної роботи (етапу) проекту або для проекту в цілому;

2) категорії ризику (наприклад, ризики управління вимогами, бюджертні ризики і т.і.);

3) ймовірності та оцінки збитків;

4) планованих заходів з їх зменшення з вказанням відповідальних осіб.

Ризики в плані, як правило, наводяться в порядку спадання пріоритету!

Слід зазначити, що управління ризиками є неперервним процесом в межах ЖЦ проекту. Відстежування та контроль ризиків, внесених до списку, повинні виконуватись на регулярній основі. Для кожного ризику або категорії ризиків слід призначити відповідальних за заходи із зниження ризиків. Також слід встановити формат звіту про управління ризиками для кожної категорії ризиків. Ці звіти розглядаються регулярно.

Моніторинг ризиків. Дуже важливо відзначити, що всі величини, пов'язані із ризиком, не є постійними в проекті. Ймовірність ризикової події та можливі збитки можуть збільшуватись та зменшуватись. В зв'язку із цим необхідно постійно:

- оцінювати результати виконаних заходів плану управління ризиками;

- відстежувати ризики, існуючі в проекті на поточний момент;

- коригувати плани із зниження ризику відповідно до поточної ситуації.

Нехай, наприклад, на початок проекту в плані управління ризиком було відзначено ризик недостатньої кваліфікації колективу розробників та запропоновані відповідні заходи з його зниження. Однак після завершення етапу розроблення, цей ризик вже не актуальний - він вже або повністю усунений (підібрано кваліфікований колектив), або перетворився на постійно діючий

фактор (колектив дійсно не має достатньої кваліфікації, але менджер про це точно знає).

Моніторинг ризиків полягає в регулярному перерахунку ймовірностей ризиків та збитку, який вони можуть завдати.

Висновки

Результат будь-якого проекту залежить від великої кількості факторів невизначеності. Тому управління ризиками повинне бути однією з основ управління проектами. Відмінні риси управління ризиками в програмних проектах полягають в основному у великій кількості ризиків, ймовірнісна оцінка яких складна. Загалом управління ризиками в таких проектах в основному підлягає загальним принципам управління ризиками.

Лекція №10
ФОРМАЛЬНІ СПЕЦИФІКАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Формальні специфікації як засіб підвищення якості ПЗ
2. Специфікування інтерфейсів
3. Специфікація поведінки систем
4. Мови розроблення формальних специфікацій

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.

4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

5. Петрухин В.А., Лаврищева Е.М. Методы и средства инженерии программного обеспечения // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/2190/237/lecture/3272

Вступ

Математика забезпечує якісні рішення для вираження статичного стану, іншими словами, для відповіді на запитання «що?». Це відрізняється від питань типу «як?», на які відповідають процедури та алгоритми. Оскільки специфікації вимог (SRS) в основному описують стан програми до та після дій, математична нотація може бути більш прийнятною, ніж природня мова, для визначення детальних вимог. Використання математики в цьому контексті є частиною так званих *формальних методів*. Формальні

методи зручні для інженерів, які володіють математикою. Багато вчених вірять, що математика відіграє важливу роль у визначенні всіх важливих деталей, які так легко можуть зіпсувати навіть якісно продуманий проект. Вони вважають, що таке використання математики може запобігти порожнім тратам коштів, руйнуванню майна, і навіть людським втратам (у життєво важливих програмах).

Для прикладу розглянемо наступне завдання (вимогу) до процедури: «процедура повертає відсортований масив, який складається з елементів масиву А». Ця проста на перший погляд вимога переповнена невизначеностями. Елементи А можуть мати декілька ключів сортування, тому термін «сортування» повинен бути визначений – сортувати за принципом парності-непарності, за зростанням чи спаданням. Наступна невизначеність полягає у тому, що для А=(4,6,3,4,6,8) може бути три масиви, які задовольняють поставленій вимозі: (3,4,6,8), (3,4,4,6,8), (3,4,4,6,6,8). Можна усунути цю невизначеність наступним формулюванням вимоги – «процедура повертає відсортований масив всіх окремих елементів масиву А», але слово «окремий» також неоднозначне. Так можна нескінченно перефразувати вимогу, але це буде лише боротьбою із математикою.

Формальна специфікація – це математичний опис програмної системи, яка може бути реалізована відповідно до цього опису. Специфікується, *що* повинна робити система, але не те, *як* вона повинна це робити. Якщо існує специфікація системи, можливо перевірити, чи буде конкретна спроектована модель задовольняти вимогам після реалізації (*валідація програмної системи*) та чи буде система задовольняти специфікації (*верифікація програмної системи*).

1. Формальні специфікації як засіб підвищення якості програмного забезпечення

Традиційні технічні дисципліни, як правило, легко адаптують математичні методи. Однак інженерія ПЗ не йде таким шляхом. Пройшло більше 25 років досліджень з використання математичних методів в процесі створення ПЗ, але вплив цих методів й досі обмежений. Так звані формальні методи

розроблення програмних систем широко не використовуються. Софтверні компанії не вважають економіно вигідним застосування цих методів у процесі розроблення.

Термін «формальні методи» передбачає ряд операцій, до складу яких входять створення формальної специфікації системи, аналіз та доведення специфікації, реалізація системи на основі перетворення формальної специфікації у програми та верифікація програм. Всі ці дії залежать від формальної специфікації ПЗ. *Формальна специфікація* – це системна специфікація, записана мовою, словник, синтаксис та семантика якої визначені формально. Необхідність формального визначення мови передбачає, що ця мова заснована на математичних концепціях. Тут використовується галузь математики – дискретна математика, яка базується на алгебрі, теорії множин та алгебрі логіки.

У 80-х роках XX століття дослідники вважали, що *формальні специфікації та формальні методи є найбільш ефективним шляхом покращення якості ПЗ*. Вони привели вагомі доводи на користь того, що суворий та детальний аналіз, який є невід'ємною частиною формальних методів, призведе до створення програм з малою кількістю помилок. Вони прогнозували, що до XXI століття більша частина ПЗ розроблятиметься з використанням формальних методів.

Тепер зрозуміло, що *ці прогнози не справдились, причому через цілий ряд причин*:

1) успіхи інженерії ПЗ – використання в процесі проектування та розроблення ПЗ новітніх технологій інженерії ПЗ призвело до певного підвищення якості програмних продуктів, що суперечить уявленням, що підвищення якості можливо досягнути лише шляхом доведення правильності програм;

2) зміни на ринку програмних продуктів – наразі головним критерієм при розробленні багатьох класів ПЗ є не якість, а час постачання його на ринок, тобто клієнти готові прийняти ПЗ з дефектами, якщо буде забезпечено швидку поставку. А методи швидкого розроблення ПЗ не дуже добре узгоджуються із формальними специфікаціями. Звісно, що не для всіх класів ПЗ

можна йти на незабезпечення належної якості на користь швидкої поставки;

3) обмежена галузь застосування формальних методів – формальні методи, як правило, погано підходять для опису взаємодії користувача та визначення інтерфейсів користувача. Інтерфейси користувача є складовою частиною більшості сучасних систем, але для них користь від застосування формальних методів обмежена;

4) обмежена масштабованість формальних методів – в успішних проектах формальні методи, як правило, використовувались для розроблення лише відносно невеликого критичного ядра системи. Проблема посилюється браком засобів підтримки таких методів.

Ці фактори призвели до того, що ризики використання формальних методів в більшості проектів ПЗ наразі перевищують можливі вигоди від їх використання. Витрати та проблеми впровадження формальних методів в процес розроблення дуже великі. Однак *формальна специфікація є прекрасним способом виявлення помилок у вимогах і засобом, який забезпечує однозначність системної специфікації*. В усіх успішних проектах, які використовували формальні методи, наявний факт зменшення кількості помилок в завершених програмних системах. Таким чином, в системах, де можливе використання формальних методів, воно може бути виправданим та, ймовірно, буде рентабельним. *Використання формальних методів зростає у спеціальній галузі розроблення критичних систем*, де дуже важливі такі властивості, як безпека, безвідмовність та захищеність. Для таких систем атестація вимагає великих витрат, а вартість відмов вельми велика. В цьому випадку рентабельно використовувати формальні методи, оскільки вони можуть зменшити ці витрати. Прикладами критичних систем, при розробленні яких успішно використовувались формальні методи, є інформаційні системи управління повітряним транспортом, системи сигналізації на залізниці, бортові системи космічних кораблів, медичні системи управління. Вони також були використані для специфікування пакетів програмних засобів, частини системи CISC (абонентська

інформаційно-керуюча система) компанії IBM та ядра систем, які працюють в режимі реального часу. Використовуваний в цих системах метод «чистої кімнати» розроблення ПЗ базується на формальних доведеннях того, що програма відповідає своїй специфікації.

Сучасні напрямки в галузі перевірки правильності програм – формальні специфікації та методи доведення їх вірності. Для доведення того, що специфікація програми задає вірне рішення деякої задачі, для якої вона розроблена, залучається математичний апарат. У формальних методах немає рутинного написання специфікації, а є аналіз тексту та опис поведінки програми в стилі, близькому математичній нотації, шляхом розмірковувань та доведень, прийнятих у математиці. Формальні методи у програмуванні з'явились разом із мовами програмування. Формальні специфікації надають засоби, які полегшують проведення розмірковувань про властивості формальних текстів та наближують їх до математичних нотацій.

Формальні методи носили в основному академічний, теоретичний характер, оскільки спочатку одержати з них практичну користь не вдавалось через великі витрати на формальну специфікацію програм та необхідність розроблення додаткових аксіом, тверджень та умов, які називають передумовами (попередніми умовами), і постумов, які визначають заключні правила одержання коректного результату.

Під *специфікацією* розуміють формальний опис функцій та даних програми, з якими ці функції оперують. Розрізняють видимі дані – вхідні та вихідні параметри, та невидимі дані – приховані дані, які не прив'язані до реалізації і визначають інтерфейс з іншими функціями. *Передумови* – це обмеження на сукупність вхідних параметрів, *постумови* – обмеження на вихідні параметри. Передумови та постумови задаються предикатами, тобто функціями, результатом яких є логічне значення (true або false). Передумова істинна тоді, коли вхідні параметри входять в область допустимих значень функції, постумова істинні тоді, коли сукупність значень задовольняють вимогам, які задають формальне визначення критерію коректності одержання результату.

Доведення проводиться за допомогою тверджень, які складаються формальною мовою та служать способом перевірки правильності програми в заданих точках. Набір тверджень використовує передумови та послідовність операцій, які призводять до перевірки результату відносно зазначеної точки програми, для якої сформульовано заключне твердження. Якщо твердження відповідає кінцевому оператору програми, де потрібно одержати кінцевий результат, то за допомогою заключного твердження та постумов робиться кінцевий висновок про часткову чи повну коректність роботи програми.

Формальні методи тісно пов'язані з математичними техніками специфікацій, верифікацією та *доведенням правильності програм*. Ці методи містять математичну символіку, формальну нотацію та апарат виведення. Правила доведення є громіздкими, тому на практиці рідко використовуються рядовими програмістами. Однак з теоретичної точки зору вони розвивають логіку використання математичного методу індукції при перевірці правильності програм. На основі специфікації програм проводиться часткове та повне доведення правильності програм.

Під *доведенням часткової правильності* розуміється перевірка виконання властивостей даних програми за допомогою тверджень, які описують те, що повинна одержати ця програма, коли завершиться її виконання відповідно до умов заключного твердження. *Повністю правильною* програмою відносно її опису і заданих тверджень буде програма, якщо вона частково правильна, і завершується її виконання при всіх даних, які їй задовольняють.

Для доведення часткової правильності використовується *метод індуктивних тверджень*, який полягає у підтвердженні виконання всіх дій методу індуктивних тверджень для підтвердження часткової правильності програми. Найбільш відомими формальними методами доведення правильності програм є метод рекурсивної індукції або тверджень Флойда, метод Наура, метод структурної індукції Хоара та інші. *Метод Флойда* засновано на визначенні умов для вхідних та вихідних даних і у виборі контрольних точок у програмі, що доводиться, так, щоб шлях проходження програмою перетинав хоча б одну контрольну точку.

Для цих точок формулюються твердження про стан та значення змінних у них. Доведення правильності застосовується для вже написаних програм і тих, які розробляються методом послідовної декомпозиції задачі на підзадачі. Даний метод доведення зменшує кількість помилок і час тестування програми, забезпечує відпрацювання специфікацій програми на повноту, однозначність, несуперечливість. *Метод Хоара* – це вдосконалений метод Флойда, заснований на аксіоматичному описі семантики мови програмування сирцевих програм. Кожна аксіома описує зміну значень змінних за допомогою операторів цієї мови. Формалізація операторів забезпечується за допомогою системи правил. Опис тверджень за допомогою правил – громіздкий і містить неповноту, не підлягає автоматизації. *Метод Маккарті* – полягає у структурній перевірці функцій, які працюють над структурними типами даних, структур даних та діаграм переходу під час символьного виконання програм. Ця техніка включає в себе моделювання виконання коду з використанням символів для змінюваних даних. Тестова програма має вхідний стан, дані та умови її виконання. Виконувана програма розглядається як серія змінів стану. Останній стан програми вважається вихідним станом, і якщо його одержано, то програма вважається вірною. Даний метод забезпечує високу якість сирцевого коду. *Метод Дейкстри* пропонує два підходи до доведення правильності програм. Перший підхід засновано на моделі обчислень, яка оперує історіями результатів обчислень програми, аналізом шляхів проходження та правил опрацювання великого обсягу інформації. Другий підхід базується на формальному дослідження тексту програми за допомогою предикатів першого порядку. Основу методу складає математична індукція, абстрактний опис програми та її обчислення.

Ще одним підходом доведення правильності програм є *валідація сценаріїв вимог* – підхід до перевірки вимог, заданих у моделі вимог, яка побудована з використанням сценаріїв та акторів як зовнішньої сутності відносно розроблюваної системи. Сценарій після трансформації – це послідовність взаємодій між одним або декількома акторами та системою, в якій актор виконує цілі сценарію при взаємодії з нею. В моделі вимог сценарій задає

декілька альтернативних подій, заданих мовами UML-діаграм. Вони поділюються на функційні (системні) та внутрішні, які визначають поведінку системи. На основі опису сценарних вимог проводиться їх валідація.

Валідація вимог – це процес виявлення помилок в представленні сценарних вимог. Він ітераційний та складається з наступних кроків:

1) формалізований опис вимог у вигляді сценаріїв;

2) створення виконуваної моделі вимог;

3) створення спеціальних сценаріїв для валідації вимог;

4) застосування валідаційних сценаріїв до моделі вимог;

5) оцінювання результатів поведінки моделі вимог;

6) перевірка умов завершення процесу валідації, при виявленні будь-яких неточностей проводиться повторення кроків, починаючи з кроку 2.

При виконанні сценаріїв виникають помилкові ситуації, за яких поведінка системи стає недетермінованою. З цією метою (з метою виявлення ризиків або помилок) проводиться контроль покриття сценаріїв у моделі вимог валідаційними сценаріями. Створюється модель помилок, яка покриває модель вимог системи і містить типові помилки, використовувані при виведенні сценаріїв. Складова частина валідації вимог у сценаріях – визначення класів еквівалентності вхідних та вихідних даних, використовуваних і для синтезу сценаріїв. Вхідна інформація для синтезу сценаріїв – сценарна модель – задається мовою взаємодії (рис.10.1). Ця інформація використовується при генерації додаткових сценаріїв з метою покращення процесу валідації, автоматичного синтезу сценаріїв моделі та одержання моделі поведінки системи. Модель перевіряє неповноту початкових вимог або суперечливості у вимогах за допомогою тестів та моделі помилок.

Автоматичний синтез засновано на наступних процедурах:

- валідація вимог шляхом виконання валідаційних сценаріїв;

- додавання перевірених сценаріїв до набору валідаційних сценаріїв та їх використання як вхідних даних для синтезу;

- пошук помилок у сценаріях та перевірка різних композицій сценаріїв.

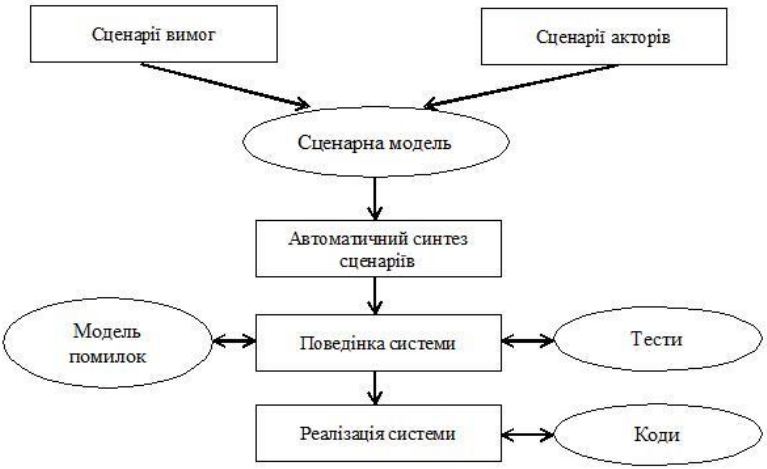


Рис.10.1 - Валідація сценаріїв вимог до системи

Також до методів доведення правильності програм належать і *методи аналізу структури програм*, які полягають у інспекції незалежними експертами за участю самих розробників. Вони перевіряють повноту, цілісність, однозначність та несуперечливість визначень у програмі. Сутність інспекції полягає у тому, що експерти намагаються поглянути на програму «з боку», піддати її всебічному критичному аналізу, розглянути словесні пояснення розробників про способи її розроблення. Мета інспекції – виявлення помилок у логіці та у програмі в статиці (статичний аналіз). До методів аналізу структури програм належать наскрізний контроль, який дозволяє виявити помилки при багаторазовому перегляді коду; метод простого структурного аналізу; метод аналізу потоків даних; метод символьної перевірки.

Існує три рівні специфікації ПЗ. Це вимоги користувача, системні вимоги та специфікації структури ПЗ. Вимоги користувача найбільш узагальнені, специфікація структури найбільш детальна. Формальні математичні специфікації

знаходяться між системними вимогами та специфікацією структури. Вони не містять деталей реалізації системи, але повинні представляти її повну математичну модель. На рис.10.2 показано етапи розроблення специфікації ПЗ та їх взаємозв'язки із процесом проектування. На рис.10.3 показано, що розроблення специфікації та проектування можуть виконуватись паралельно.

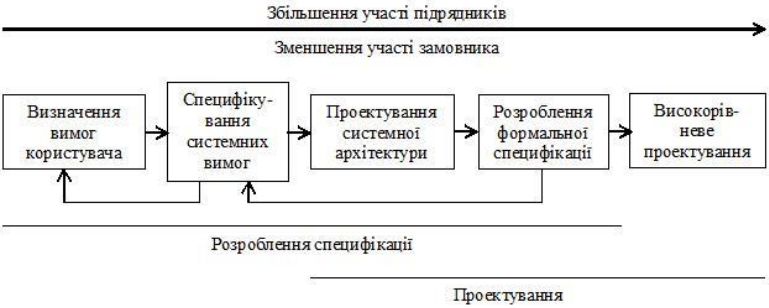


Рис.10.2 - Розроблення специфікації та проектування

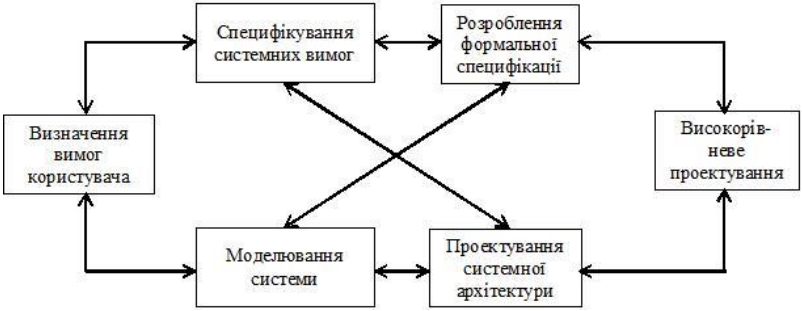


Рис.10.3 - Розроблення формальної специфікації

Створення формальної специфікації вимагає детального аналізу системи, який дозволяє виявити помилки та невідповідності у специфікації неформальних вимог. Ця можливість виявлення помилок – найважливіший аргумент для використання формальної специфікації, хоча її розроблення та аналіз вимагають додаткових витрат. При звичайному процесі розроблення ПЗ вартість атестації системи складає близько 50% всієї вартості розроблення, а вартість

проектування і реалізації системи вдвічі більше вартості розроблення специфікації. При використанні формальної специфікації вартості розроблення специфікації і реалізації системи практично однакові, а вартість атестації значно знижується, оскільки в процесі розроблення формальної специфікації виявляються і усуваються не доопрацювання у вимогах і виключається переробка системи на останніх стадіях її створення.

Існує *два основних підходи до розроблення формальної специфікації*:

1) алгебраїчний підхід, при якому система описується в термінах операцій та їх відношень;

2) підхід, орієнтований на моделювання, при якому модель системи будується з використанням математичних конструкцій, таких як множини і послідовності, а системні операції визначаються тим, як вони змінюють стан системи.

2. Специфікування інтерфейсів

Великі системи, як правило, розбиваються на підсистеми, які розробляються незалежно одна від одної. Підсистеми можуть використовувати інші підсистеми, тому необхідною частиною процесу специфікування є визначення інтерфейсів підсистем. Якщо інтерфейси визначені і узгоджені, підсистеми можна розробляти незалежно одна від одної. Інтерфейс підсистеми часто визначається як набір абстрактних типів даних та об’єктів, при цьому лише через інтерфейс доступні опис даних та операції над ними. Тому специфікацію інтерфейсу підсистеми можна розглядати як об’єднання специфікацій компонентів, що в результаті і складе опис інтерфейсу підсистеми.

Точні специфікації інтерфейсів підсистем необхідні для розробників, які пишуть програмний код, який звертається до сервісів інших підсистем. Специфікації інтерфейсів містять інформацію про те, які сервіси доступні в інших підсистемах і як одержати до них доступ. Ясний та однозначний інтерфейс підсистем зменшує ймовірність помилок у взаємовідносинах між ними.

Алгебраїчний підхід початково був розроблений для опису інтерфейсів абстрактних типів даних, де типи даних визначаються швидше специфікаціями операцій над даними, ніж способом представлення самих даних. Це дуже схоже на визначення класів об'єктів. Алгебраїчний підхід до формальних специфікацій визначає абстрактний тип даних в термінах операцій над даними.

Структуру специфікації об'єкту показано на рис.10.4.

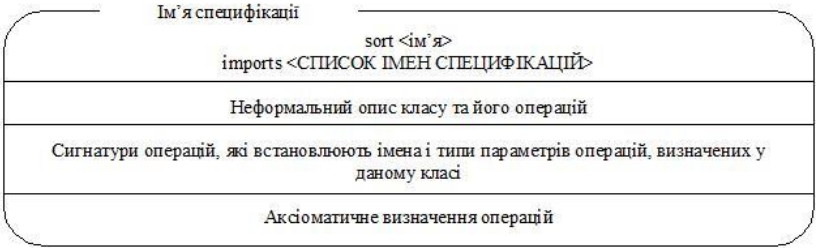


Рис.10.4 - Структура алгебраїчної специфікації

Специфікація об'єкту складається з 4-х компонентів:

- вступ, де оголошується клас об'єктів, а також відбувається включення оголошень імпорту з іменами специфікацій для інших класів;

- описова частина, в якій неформально описуються операції, асоційовані з класом, із забезпеченням однозначного синтаксису та семантики операцій;

- частина сигнатур, в якій визначається синтаксис інтерфейсу об'єктного класу або абстрактного типу даних (описуються імена операцій, кількість та типи їх параметрів, класи вихідних результатів операцій);

- частини аксіом, де визначається семантика операцій завдяки створення ряду аксіом, які характеризують поведінку абстрактного типу даних; ці аксіоми зв'язують операції створення об'єктів класу з операціями, які перевіряють їх значення.

Процес розроблення формальної специфікації інтерфейсу підсистеми включає наступні дії:

1) структурування специфікації – представлення неформальної специфікації у вигляді множини абстрактних типів даних або об'єктних класів;

2) іменування специфікацій – задаються імена кожної специфікації абстрактного типу, визначаються параметри специфікацій та імена класів;

3) визначення операцій – на основі списку виконуваних інтерфейсом функцій для кожної специфікації визначається зв'язаний з нею набір операцій; операції над абстрактним типом даних належать до операцій конструювання, які створюють або змінюють об'єкти класу, або до операцій перевірки, які повертають атрибути класу; добрим емпіричним правилом для написання алгебраїчної специфікації є створення аксіом для кожної операції конструювання із застосуванням всіх операцій перевірки - це означає, що, якщо є m операцій конструювання та n операцій перевірки, то повинно бути визначено m*n аксіом.

4) неформальна специфікація операцій – написання неформальної специфікації для кожної операції із вказанням, як операції впливають на клас;

5) визначення синтаксису і параметрів операцій – це частина сигнатури формальної специфікації;

6) визначення аксіом – визначення семантики операцій шляхом опису умов, які повинні виконуватись для різних комбінацій операцій.

3. Специфікація поведінки систем

Прості алгебраїчні методи підходять для опису інтерфейсів, коли операції, асоційовані з об'єктом, не залежать від стану об'єкта. Тоді результати будь-якої операції не залежать від результатів попередніх операцій. Якщо ця умова не виконується, алгебраїчні методи можуть стати громіздкими. Крім того, алгебраїчні описи поведінки систем часто штучні на важкі для розуміння. Альтернативним підходом до створення формальних специфікацій, який широко використовується в програмних проектах, є специфікація, заснована на моделях системи. Такі специфікації використовують моделі станів системи. Системні

операції визначаються за допомогою змін станів системної моделі. Таким чином визначається поведінка системи. Для розроблення специфікацій, заснованих на системних моделях, використовуються системи нотацій методів VDM та Z, в яких система описується на основі теорії множин із додатковими логічними структурами, розробленими для специфікування програмних систем.Формальні специфікації можуть бути важкими для читання та громіздкими, особливо якщо використовуються великі математичні формули. Це сповільнює розроблення ПЗ. Тому в методі Z специфікації представляються як неформальний текст, доповнений формальними описами. Формальна частина специфікації складається з невеликих простих описів (схем), які візуально відокремлюють специфікації від іншого тексту (рис.10.5). Схеми використовуються для введення змінних станів та визначення обмежень і операцій над станами. В методі також передбачені операції, виконувані над схемами, зокрема, для побудови, перейменування та приховування схем. Сигнатура схеми визначає сутності, які складають стан системи, схемні предикати (набір умов, які повинні бути завжди істинні для цих сутностей – наприклад, перед- та постумови для схеми, яка визначає операції).

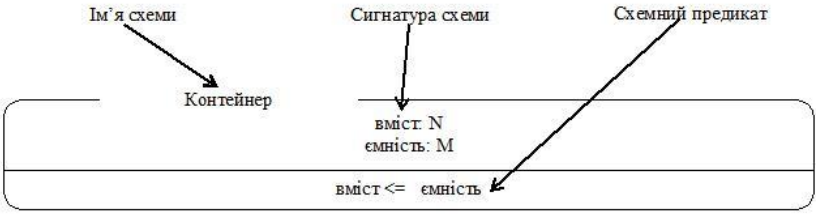


Рис.10.5 – Структура Z-схеми

Предикати в усіх Z-схемах повинні бути узгоджені, тобто не повинно бути умов в одній схемі, які суперечать предикатам іншої схеми. Якщо в специфікації помічені суперечливості, можна застосувати різні математичні методи аналізу Z-специфікації.

Основною перевагою використання формальної специфікації є можливість виявлення проблем в системних

вимогах. В неформальній специфікації легко пропустити ці проблеми, які однаково будуть вирішені, але на більш пізньому етапі ЖЦ ПЗ відповідно з більшими витратами.

4. Мови розроблення формальних специфікацій

Специфікація програми – це точний, однозначний та недвозмістовний опис програми за допомогою математичних понять, термінів, правил синтаксису та семантики мови специфікації. В *мові специфікацій* можуть бути поняття та конструкції, які не можна виконати на комп'ютері, вони представляються послідовністю операцій, функцій, зрозумілих для інтерпретації. *Опис задачі мовою специфікацій* – це опис загального контексту всіх понять, через які визначаються поняття, що беруть участь у формулюванні задачі або в описі моделі (домену). Опис задачі надається у вигляді аксіом, тверджень, перед- та постумов, які вимагають для їхньої реалізації не систем програмування, а спеціального апарату для доведення або верифікації опису задач, зокрема інтерпретаторів або метасистем.

Мови специфікацій, використовувані для формального опису властивостей програм, мають більш високий рівень, ніж мови програмування. Їх можна класифікувати за такими категоріями: універсальні мови із загальноматематичною основою (засновані на моделях); мови специфікацій предметних галузей, спеціалізовані мови специфікацій (мови опису засобів); мови, орієнтовані на специфікацію паралельних процесів.

Універсальні мови специфікацій (VDM, Z, RAISE) мають загально математичну основу та наступні види засобів:

- логіки першого порядку, включаючи квантори;
- арифметичні операції;
- засоби утворення множин за допомогою логічних формул та операцій над множинами;
- засоби опису кінцевих послідовностей (кортежів, списків) та операції над ними;
- засоби опису кінцевих функцій та операції над ними;
- засоби опису деревовидних структур;

- засоби побудови областей або множини об'єктів, включаючи добутки, об'єднання та рекурсивні визначення;

- визначення функцій за допомогою виразів та рівностей, включаючи рекурсивні визначення;

- процедурні властивості мов програмування (оператори присвоєння, циклу, вибору, виходу);

- операції композиці, аргументами і результатами яких можуть бути функції, вирази, оператори.

В мовах VDM і RAISE немає засобів опису графових структур, керування та паралелізму, однак є механізм конструювання нових структур даних.

Мови специфікації галузей включають в себе наступні мови:

- специфікація доменів;

- опис взаємодії;

- специфікації мов програмування і трансляторів;

- специфікації баз даних та знань;

- специфікації пакетів прикладних програм.

Кожна з цих мов має спеціалізовані засоби, які відображають специфічні особливості відповідної галузі.

Мови опису засобів програмування включають в себе мови, засновані на рівностях та підстановках з операційною семантикою (Lisp, Refal), логічні мови, мови операцій над послідовностями і матрицями, табличні мови, мережі, графи.

Загальна нотація для формального вираження вимог називається *Z-специфікацією*. Z-специфікації є стандартним способом опису потрібного стану до та після процедури.

Нижче наведено деяку вибірку з Z-нотації:

$\Longrightarrow$ - означає імплікацію або логічне слідування;

$\forall$ - означає «для будь-якого»;

$\wedge$ - означає «ТА»;

$\vee$ - означає «АБО»;

N - множина натуральних чисел (ім'я множини завжди велика літера);

$\in$ - операція належності елемента множині;

$\notin$ - відповідно операція неналежності елемента множині;

$\subseteq$ - операція належності підмножини множині;

$\times$ - операція прямого добутку множин, тобто утворення множини пар елементів двох множин;

{} – позначення кінцевих множин (елементи перераховуються через кому);

$\{x : P\}$ - позначення множини елементів x, які мають властивість P;

$card(S)$ - кількість елементів кінцевої множини S;

$f : A \to B$ - функція з множини у множину, результат – підмножина прямого добутку множин, причому жоден елемент з A не зустрічається більше одного разу в якості першого елементу пари;

$f^{-1}(y)$ - множина елементів з області визначення, які відображаються на елемент y, який належить області значень функції, тобто $f^{-1}(y) = \{x : f(x) = y\}$. Область визначення функції – множина елементів, які зустрічаються в якості перших елементів множини пар, які утворюють функцію. Областю значень функції називають множину елементів, які зустрічаються в якості других елементів множини пар, які утворюють функцію;

$f : R \mapsto P$ - означає, що f є частковою функцією з R в P;

$a \mapsto b$ - означає, що a відображається в b в контексті функції, яка визначається;

$f \oplus g$ - означає розширення (до визначення) функції g функцією f для $f : A \to B$, $g : A \to B$.

Висновки

Методи формальної специфікації доповнюють методи неформальної специфікації. Вони можуть використовуватись для деталізації специфікації неформальних системних вимог. Формальні специфікації є мостом між системними вимогами та системною архітектурою. Формальні специфікації точні та однозначні. Вони усувають «темні» області в специфікації та

проблеми неоднозначного трактування вимог. Разом з тим формальні специфікації важкі для розуміння нефахівцями.

Основною перевагою формальних методів є те, що аналіз системних вимог проводиться на ранньому етапі розроблення ПЗ. Виправлення помилок на цьому етапі дешевше, ніж внесення змін у завершену систему.

Методи формальної специфікації найбільш підходять для розроблення критичних систем, де безпека, безвідмовність та захищеність системи особливо важливі. Вони можуть також використовуватись для розроблення стандартів.

Алгебраїчні методи формальних специфікацій особливо підходять для розроблення інтерфейсів, коли інтерфейс визначено як набір класів об’єктів або абстрактних типів даних. Ці методи приховують стан системи і визначають її в термінах відношень між інтерфейс ними операціями.

Методи формальної специфікації, засновані на системних моделях, використовують математичні конструкції теорії множин. Операції визначаються за способом впливу на стани системи. Це спрощує створення специфікації поведінки системи.

Лекції №11-13
МЕТОДИ ТА ЗАСОБИ КОЛЕКТИВНОГО РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Методологія розроблення ПЗ Microsoft Solutions Framework (MSF). Принципи створення бібліотеки MSF. Модель команди в MSF, ролеві кластери, масштабованість команд та керування компромісами у MSF
2. Гнучкий підхід до створення ПЗ, основні принципи гнучкого розроблення
3. Реалізація концепції керування програмним проектом на всіх етапах життєвого циклу у Visual Studio 2012
4. Функціональні можливості та архітектура TeamFoundationServer 2012 (TFS)
5. Способи розгортання TFS на одному або декількох серверах, в одному домені, робочі групі або в декількох доменах. Шаблони командних проектів TFS, області керування командними проектами
6. Питання створення командного проекту, зміст програмної інфраструктури проекту, склад і призначення робочих елементів
7. Аналіз методології Scrum, робочі елементи шаблону MicrosoftVisualStudioScrum 2.2
8. Організація колективу у методології Scrum. Життєвий цикл проекту

Рекомендована література:

1. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002
2. Ф.А.Новиков, Э.А.Опалева, Е.О.Степанов. Учебно-методическое пособие по дисциплине «Управление проектами и разработкой ПО» // [Електронний ресурс] - Режим доступу: http://window.edu.ru/resource/366/60366/files/itmo305.pdf

3. Технологии командной разработки программного обеспечения информационных систем // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/4806/1054/lecture/9998

4. Д.Андреев. Организация процессов разработки программного обеспечения с использованием Team Foundation Server 2010 // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/649/505/info

5. Бьерк А., ДелаМазаМ., Резник С. ScrumTeamFoundationServer 2010. Профессиональный подход. - М. ЭКОМ Паблишерз, 2012.

6. Левинсон Дж. Тестирование ПО с помощью Visual Studio 2010. - М.: ЭКОМ-Паблишерз, 2012.

7. Шаблоны и средства Microsoft — Microsoft Visual Studio 2010 // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/vstudio//aa718795.aspx

8. Шаблон процесса гибкой разработки для Visual Studio ALM // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/dd380647.aspx

9. Шаблон процесса Scrum для Visual Studio ALM // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/ff731587.aspx

10. Управления жизненным циклом приложений с помощью Visual Studio и Team Foundation Server // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/fda2bad5.aspx

11. Мейер Дж. Д. Командная разработка с использованием Visual Studio Team Foundation Server / Дж. Д.Мейер, Дж. Тейлор, А. Макман, П. Бансод, К. Джонс - Изд. Корпорация Microsoft, 2007.

Вступ

Сучасний стан бізнесу вимагає від розробників ПЗ розроблення програмних продуктів високої якості в межах відведеного бюджету і в термін. У створенні програмних продуктів, як правило, приймають участь різні фахівці, які об’єднуються в

колективи (команди). Команди можуть включати співробітників організації розробника і замовника, тимчасово залучених співробітників та субпідрядників. Члени команди розробників ПЗ можуть територіально знаходитись в одному або різних місцях (розподілене розроблення). Ефективний розв'язок задач створення якісного ПЗ передбачає використання інструментальних засобів, методик та технологій управління процесами ЖЦ ПЗ: формування вимог, моделювання, проектування, розроблення, тестування, побудови та розгортки систем.

Раціональна організація процесів розроблення ПЗ описується в стандартах (міжнародних, державних, копроративних), які часто називають методологіями розроблення ПЗ. Методології створення ПЗ, як правило, розробляються провідними виробниками програмних систем та їх спільнотами з врахуванням особливостей програмних продуктів, а також сфери впровадження. Методології описують підходи до організації раціональної стратегії та можливого набору процесів створення ПЗ.

Наразі все більше поширення одержують гнучкі методології розроблення ПЗ, де основна увага зосереджена на створенні якісного продукту, а не на підготовці вичерпної документації за проектом. При цьому акцент робиться на організації ефективного управління командою. Як зазначає Еріх Гамма, «ключ до своєчасної поставки продукту – не процеси, а люди». Самоорганізація та цілеспрямованість команди розробників дозволяє створювати високоякісні програмні продукти в стислі терміни.

Інструменти управління ЖЦ додатків в основному сприяють успішності програмних проектів. Компанія Microsoft надає розробникам гнучкий інструментарій для управління ЖЦ додатків – ALMVisualStudio і TeamFoundationServer. Традиційні засоби розроблення програм у VisualStudio доповнені засобами архітектурного проектування і тестування. Інструментарій TeamFoundationServer дозволяє формувати та відстежувати вимоги до програмної системи, зв'язувати їх із задачами та реалізацією, розподіляти між членами команди, проводити побудову програмного продукту, керувати тестуванням, проводити контроль

версій надавати засоби комунікації з членами команди та замовниками, підготовляти численні звіти.

Дані лекції мають за мету представити основні положення командного розроблення ПЗ, управління ЖЦ додатків, гнучкої методології створення програмних систем, а також можливості інструментарію VisualStudio2012 і TeamFoundationServer для управління ЖЦ додатків.

1. Методологія розроблення ПЗ Microsoft Solutions Framework (MSF). Принципи створення бібліотеки MSF. Модель команди в MSF, рольові кластери, масштабованість команд та керування компромісами у MSF

Microsoft Solutions Framework (MSF) є методологією розроблення ПЗ, яка представляє собою узагальнення кращих проектних практик, які використовувались командами розробників Microsoft. Дана методологія описує керування людьми та робочими процесами при розробленні ІТ-рішень.

ІТ-рішення – розуміється як скоординоване постачання набору елементів (таких як програмні засоби, документація, навчання та супровід), необхідних для задоволення бізнес-потреб конкретного замовника.

Концепція управління ЖЦ додатків, прийнята розробниками ПЗ, призвела до того, що методологія MSF стала складовою частиною продукту Visual Studio Team System (VSTS), який реалізовував підхід Microsoft. До продукту VSTS ввійшли шаблони процесів для реалізації положень моделі CMMI - MSF for CMMI і моделей гнучкого розроблення ПЗ - MSF for Agile та Scrum. Таким чином, VSTS є інструментарієм управління ЖЦ додатків, який дозволяє створювати програмні системи різного призначення в командах, які дотримуються різних підходів до управління процесами розроблення ПЗ.

Основними є наступні принципи MSF:

1) єдине бачення проекту, яке передбачає розуміння всіма зацікавленими особами цілей та задач створення ПЗ;

2) гнучкість – готовність до змін, що забезпечує можливість уточнення та зміни вимог в процесі розроблення ПЗ, оперативного

та швидкого реагування на поточні зміни умов проекту при незмінній ефективності управлінської діяльності;

3) концентрація на бізнес-пріоритетах, що передбачає створення продукту з високою якістю для споживача та формування певної вигоди або віддачі. Для організацій, як правило, це одержання прибутку;

4) заохочення вільного спілкування, що передбачає відкритий та чесний обмін інформацією як всередині команди, так і з ключовими зацікавленими особами.

Універсальність моделі MSF визначається тим, що завдяки своїй гнучкості та відсутності жорстко встановлених зв'язків і процедур, вона може бути застосована при розробленні різних програмних додатків, які можуть використовуватись в бізнесі та повсякденному житті.

Модель MSF базується на сполученні двох моделей ЖЦ програмних систем: каскадної та спіральної (рис.11.1).

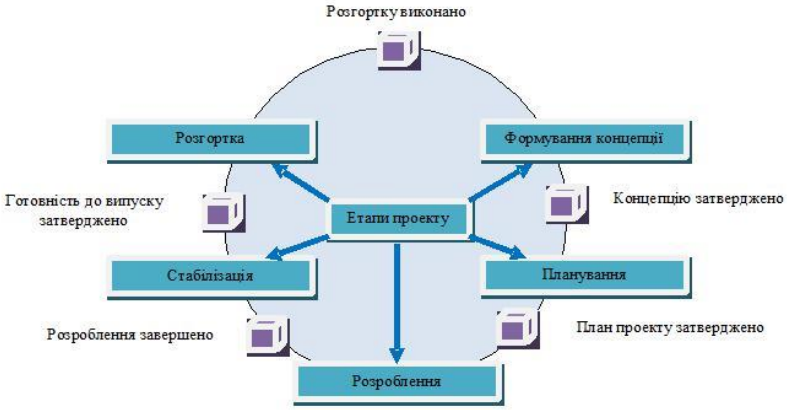


Рис.11.1 - Модель життєвого циклу рішення MSF

В основі методології MSF лежить *ітеративний інтегрований підхід до створення та впровадження рішень*, який базується на фазах та віхах.

Ітеративність підходу передбачає поетапне створення робото здатної програмної системи з визначеною функційністю,

яка відбиває вимоги до кінцевого продукту на даному етапі розроблення. Кожен виток спіралі складається з ідентичних фаз, на яких виконуються етапи робіт з формування концепції фази, планування, розроблення, стабілізації та впровадження. Набір вимог програмної системи та відповідних їм задач, які реалізуються на заданому витку спіралі, визначається менеджером проекту на основі ранжування вимог до системи. Кожен наступний виток спіралі додає до програмної системи функційність, яка відбиває вимоги замовника.

Інтеграція в межах одного проекту процедур розроблення та впровадження системи дозволяє представити замовнику різні проміжні робото здатні версії програмного продукту, оперативно вносити зміни, які відбивають бачення замовником функційності та дизайну проектованої системи, знизити ризики проекту за рахунок раннього виявлення проблем та можливості своєчасного їх усунення.

Фази проекту визначають послідовно вирішувані задачі, а *віхи* (milestones) – ключові точки проекту, які характеризують досягнення будь-якого істотного результату.

У MSF використовуються *два види віх*: головні та проміжні. Вони мають наступні *характеристики*:

1) *головні віхи* служать точками переходу від однієї фази до іншої і визначають зміни в поточних задачах ролевих кластерів проектної команди; в MSF головні віхи є достатньо універсальними для застосування в будь-якому IT-проекті;

2) *проміжні віхи* показують досягнення визначеного прогресу у виконанні фази проекту і розчленовують великі сегменти роботи на менші ділянки, які піддаються огляду та керуванню; проміжні віхи можуть варіюватись в залежності від характеру проекта.

Головна особливість моделі команди у MSF є те, що вона не має офіційного лідера. Всі відповідають за проект в рівному ступені, рівень зацікавленості кожного в результаті дуже високий, а комунікації всередині групи чіткі, ясні, дружні та відповідальні. Таке можливе при високому рівні самосвідомості та зацікавленості

кожного члена команди, а також при достатньо високому рівні професіоналізму.

Однією з особливостей відношень всередині команди є висока культура дисципліни обов'язків:

- готовність працівників приймати на себе обов'язки перед іншими;

- чітке визначення тих обов'язків, які вони на себе беруть;

- прагнення прикладати необхідні зусилля до виконання своїх обов'язків;

- готовність чесно та невідкладно інформувати про загрози виконання своїх обов'язків.

Рольові кластери MSF засновані на семи якісних цілях, досягнення яких визначає успішність проекту. Ці цілі обумовлюють модель проектної групи та утворюють рольові кластери (або просто ролі) в проекті. Кожен кластер може включати одного або декількох фахівців. Кожний рольовий кластер представляє унікальну точку зору на проект, і в той же час жоден з членів проектної групи не в стані сам успішно представляти всі можливі погляди, які відбивають якісно різні цілі.

У MSF є наступні *рольові кластери*:

1) управління продуктом – основна задача кластеру забезпечити, щоб замовник залишився задоволеним в результаті виконання проекту; цей рольовий кластер в проекті представляє інтереси замовника; ивін представляє бізнес-сторону проекту та забезпечує його узгодженість зі стратегічними цілями замовника;

2) управління програмою – кластер забезпечує управлінські функції – відстежування планів та їх виконання, відповідальність за бюджет, ресурси проекту, вирішення проблем та труднощів процесу, створення умов, за яких команда може працювати ефективно, долаючи мінімум імамкратичних перешкод;

3) розроблення – кластер забезпечує розроблення коду додатку;

4) тестування – кластер відповідає за тестування ПЗ;

5) задоволення споживача – кластер розв'язує задачі дизайну інтерфейсу користувача та забезпечення зручності

експлуатації ПЗ, навчання всіх користувачів роботі з ПЗ, створення документації для користувача;

6) управління випуском – кластер відповідає за впровадження проекту та його функціювання, бере на себе зв'язок між розробленням рішення, його впровадженням та наступним супроводом, забезпечуючи інформованість членів проектної групи про наслідки їхніх рішень;

7) архітектура – кластер відповідає за організацію та виконання високорівневого проектування рішення, створення функційної специфікації ПЗ та керування цією специфікацією в процесі розроблення, визначення меж проекту та ключових компромісних рішень.

Зміни в задачах рольових кластерів проектної команди відбуваються із змінами фаз проекту. Перехід від однієї фази до іншої включає в себе також перенос основної відповідальності від одних рольових кластерів до інших.

В залежності від розміру та складності проекту модель команд MSF припускає масштабування. Для невеликих та нескладних проектів один співробітник може об'єднувати декілька ролей. При цьому деякі ролі не можна об'єднувати. Зокрема, не можна суміщати розроблення та тестування, оскільки необхідно, щоб у тестувальників було сформовано свій, незалежний, погляд на систему, який базується на вивченні вимог. В таблиці 11.1 представлені рекомендації MSF відносно суміщення ролей в межах одного проекту одним членом команди. «+» означає, що суміщення можливе, «+-» - що суміщення можливе, але небажане, «-» означає, що суміщення не рекомендується.

Для великих команд (більше 10 чоловік) модель проектної групи MSF пропонує розбиття на малі багатопрофільні групи напрямків. Ці малі колективи працюють паралельно, регулярно синхронізуючи свої зусилля, кожна з яких базується на основі моделі кластерів. Такі команди мають чітко визначену задачу і відповідальні за всі питання, які до неї належать, починаючи від проектування та складання календарного графіку.

Таблиця 11.1 - Рекомендації MSF відносно суміщення ролей в команді проекту

	Управління продуктом	Управління програмою	Розроблення	Тестування	Задоволення користувача	Управління випуском	Архітектура
Управління продуктом		-	-	+	+	+-	-
Управління програмою	-		-	+-	+-	+	+
Розроблення	-	-		-	-	-	+
Тестування	+	+-	-		+	+	+-
Задоволення користувача	+	+-	-	+		+-	+-
Управління випуском	+-	+	-	+	+-		+
Архітектура	-	+	+	+-	+-	+	

Крім того, коли рольовому кластеру необхідно багато ресурсів, формуються так звані функційні групи, які потім об'єднуються в рольові кластери. Вони створюються у великих проектах, коли необхідно згрупувати працівників всередині рольових кластерів за їх галузями компетенції. Часто функційні групи мають внутрішню ієрархічну структуру. Наприклад, менеджери програми можуть бути підзвітними провідним менеджерам програми, які, в свою чергу, звітують перед головним менеджером програми. Подібні структури можуть також з'являтись всередині галузей компетенцій.

Керування компромісами. Добре відома взаємозалежність між ресурсами проекту (людськими та фінансовими), його календарним графіком (часом) та реалізованими можливостями (функційність). Ці три змінні утворюють трикутник, показаний на рис.11.2. Після досягнення рівноваги в цьому трикутнику зміна будь-якої з його сторін для підтримки балансу вимагає модифікацій на іншій (двох інших) сторонах та/або на зміненій стороні.

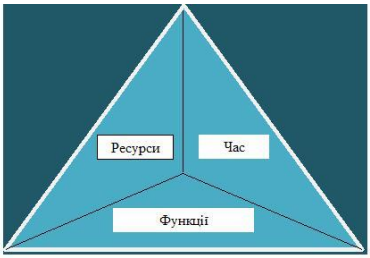


Рис.11.2 - Трикутник компромісів

В проектній практиці може фіксуватись один з ресурсів, тоді в процесі проектування системи вимірюванню підлягають лише два ресурси, що залишились. Дана ситуація задається матрицею компромісів, приведеною на рис.11.3. Так, якщо фіксуються ресурси, то час виконання проекту узгоджується, а функційні можливості підлягають зміні.

	Узгоджується	Фіксується	Приймається
Ресурси	*		
Час		*	
Функції			*

Рис.11.3 – Матриця компромісів

2. Гнучкий підхід до створення ПЗ, основні принципи гнучкого розроблення

Гнучка методологія розроблення ПЗ орієнтована на використання ітеративного підходу, при якому програмний продукт створюється поступово, невеликими кроками, які включають реалізацію певного набору вимог. При цьому передбачається, що вимоги можуть змінюватись. Команди, які використовують гнучкі методології, формуються з універсальних розробників, які виконують різні задачі в процесі створення програмного продукту.

При використанні гнучких методологій мінімізація ризиків здійснюється шляхом зведення розроблення до серії коротких циклів, які називаються *ітераціями*, тривалістю 2-3 тижні. Ітерація представляє собою набір задач, запланованих на виконання в певний період часу. В кожній ітерації створюється роботоздатний варіант програмної системи, в якій реалізуються найбільш пріоритетні (для даної ітерації) вимоги замовника. На кожній ітерації виконуються всі задачі, необхідні для створення роботоздатного ПЗ: планування, аналіз вимог, проектування, кодування, тестування та документування. Хоча окрема ітерація, як правило, недостатня для випуску нової версії продукту, мається на увазі, що поточний програмний продукт готовий до випуску в кінці кожної ітерації. По завершенню кожної ітерації команда виконує переоцінку пріоритетів вимог до програмного продукту, можливо, вносить корективи у розроблення системи.

Для методології гнучкого розроблення декларовані *ключові постулати*, які дозволяють командам досягати високої продуктивності:

- люди та їх взаємодія;
- доставка працюючого ПЗ;
- співробітництво із замовником;
- реакція на зміни.

Люди та взаємодія. Люди – найважливіша складова частина успіху. Окремі члени команди та якісні комунікації важливі для високопродуктивних команд. Для сприяння комунікації гнучкі методи передбачають часті обговорення результатів роботи та внесення змін в рішення. Обговорення можуть проводитись щоденно тривалістю по декілька хвилин і по завершенню кожної ітерації - аналізом результатів робіт та ретроспективою. Для ефективних комунікацій при проведенні зборів учасники команд повинні дотримуватись наступних *ключових правил поведінки*:

- повага думки кожного учасника команди;
- бути правдивим при будь-якому спілкуванні;
- прозорість всіх даних, дій та рішень;
- впевненість, що кожен учасник підтримає команду;

- відданість команді та її цілям.

Для створення високопродуктивних команд в гнучких методологіях, крім ефективної команди та якісних комунікацій, необхідний досконалий програмний інструментарій.

Працююче ПЗ важливіше за всеосяжну документацію. Всі гнучкі методології виділяють необхідність доставки замовнику невеликих фрагментів працюючого ПЗ через задані інтервали. ПЗ, як правило, повинне пройти рівень модульного тестування, тестування на рівні системи. При цьому обсяг документації повинен бути мінімальним. В процесі проектування команда повинні підтримувати в актуальному стані короткий документ, який містить обґрунтування розв'язку та опис структури.

Співробітництво із замовником важливіше формальних домовленостей за контрактом. Щоб проект успішно завершився, необхідне регулярне та часте спілкування із замовником. Замовник повинен регулярно приймати участь в обговоренні прийнятих рішень з ПЗ, висловлювати свої побажання та зауваження. Залучення замовника у процес розроблення ПЗ необхідне для створення якісного продукту.

Оперативне реагування на зміни важливіше за слідування планом. Здатність реагування на зміни багато в чому визначає успішність програмного проекту. В процесі створення програмного продукту дуже часто змінюються вимоги замовника. Замовники дуже часто точно не знають, чого хочуть, доки не побачать працююче ПЗ. Гнучкі методології шукають зворотній зв'язок віз замовників в процесі створення програмного продукту. Оперативне реагування на зміни необхідне для створення продукту, який задовольнить замовника та забезпечить цінність для бізнесу.

Постулати гнучкого розроблення підтримуються 12 принципами. В конкретних методологіях гнучкого розроблення визначено процеси та правила, які в більшому або меншому степені відповідають цим принципам. *Гнучкі методології створення ПЗ базуються на наступних принципах*:

1) вищий пріоритет надавати задоволенню побажань замовника за допомогою поставки корисного ПЗ в стислі терміни з наступним неперервним оновленням. Гнучкі методики

передбачають швидку поставку початкової версії та часті оновлення. Метою команди є поставка робото здатної версії протягом декількох тижнів з моменту початку проекту. Далі програмні системи з поступово нарощуваною функційністю повинні постачатись кожні декілька тижнів. Замовник може почати промислову експлуатацію системи, якщо вважатиме, що вона достатньо функційна. Також замовник може просто ознайомитись з поточною версією ПЗ, надати свій відгук із зауваженнями;

2) не ігнорувати зміну вимог, нехай навіть на пізніх етапах розроблення. Гнучкі процеси дозволяють враховувати зміни для забезпечення конкурентних переваг замовника. Команди, які використовують гнучкі методики, прагнуть зробити структуру програми якісною, з мінімальним впливом змін на систему в цілому;

3) постачати нові працюючі версії ПЗ часто, з інтервалом від одного тижня до двох місяців, надаючи перевагу меншим термінам. При цьому ставиться мета постачати програму, яка задовольняє потреби користувача, з мінімумом супровідної документації;

4) замовники і розробники повинні працювати сумісно протягом всього проекту. Вважається, що для успішного проекту замовники, розробники та всі зацікавлені особи повинні спілкуватись часто і багато для цілеспрямованого вдосконалення програмного продукту;

5) проекти повинні втілювати в життя цілеспрямовані люди. Створіть команді проекту нормальні умови роботи, забезпечте необхідну підтримку та сподівайтесь, що члени команди доведуть справу до завершення;

6) найефективніший та найпродуктивніший метод передачі інформації команді розробників та обміну думками всередині неї – розмова обличчям до обличчя. В гнучких проектах основний спосіб комунікації – просте людське спілкування. Письмові документи створюються та оновлюються поступово по мірі розроблення ПЗ і лише за необхідності;

7) працююча програма – основний показник прогресу в проекті. Про наближення гнучкого проекту до завершення роблять

висновки за тим, наскільки наявна в даний момент програма відповідає вимогам замовника;

8) гнучкі процеси сприяють довготерміновому розробленню. Замовники, розробники та користувачі повинні бути в стані підтримувати незмінний темп як завгодно довго;

9) незмінна увага до технічної досконалості та якісного проектування підвищує віддачу від гнучких технологій. Члени гнучкої команди прагнуть створювати якісний код, регулярно проводячи рефакторинг;

10) простота – мистецтво досягати більшого, роблячи менше. Члени команди вирішують поточні задачі максимально просто та якісно. Якщо в майбутньому виникне якась проблема, то в якісний код є можливість внести зміни без великих витрат;

11) найкращі архітектури, вимоги та проекти видають команди, які самостійно організуються. В гнучких командах задачі доручаються не окремим членам, а команді в цілому. Команада самостійно вирішує, як найкраще реалізувати вимоги замовника. Члени команди сумісно працюють над всіма аспектами проекту. Кожному учаснику дозволено вносити свій внесок у спільну справу. Немає такого члену команди, який одноосібно відповідав би за архітектуру, вимоги або тести;

12) команда повинна регулярно замислюватись над тим, як стати ще більш ефективною, а потім відповідно коригувати та підлаштовувати свою поведінку. Гнучка команда постійно коригує свою організацію, правила, угоди та стосунки.

Вищенаведеним принципам відповідають *низка методологій розроблення ПЗ*:

- AgileModeling – набір понять, принципів та прийомів (практик), які дозволяють швидко та просто виконувати моделювання та документування в проектах розроблення ПЗ;

- AgileUnifiedProcess(AUP) – спрощена версія IBM RationalUnifiedProcess(RUP), яка описує просте та зрозуміле наближення (модель) для створення ПЗ бізнес-додатків;

- OpenUP – це ітеративно-інкрементальний метод розроблення ПЗ. Позиціонується як легкий та гнучкий варіант RUP;

- AgileDataMethod – група ітеративних методів розроблення ПЗ, в яких вимоги та рішення досягаються в рамках співробітництва різних крос-функційних команд;

- DSDM – методика розроблення динамічних систем, заснована на концепції швидкого розроблення додатків (RapidApplicationDevelopment, RAD). Представляє собою ітеративний та інкрементний підхід, який надає особливого значення тривалій участі в процесі користувача/споживача;

- Extremeprogramming (XP) – екстремальне програмування;

- Adaptive software development (ADD) – адаптивне розроблення програм;

- Featuredrivendevelopment (FDD) - розроблення, орієнтоване на поступове нарощування функційності;

- GettingReal - ітеративний підхід без функційних специфікацій, який використовується для web-додатків;

- MSFfogAgileSoftwareDevelopment – гнучка методологія розроблення ПЗ компанії Microsoft;

- Scrum – встановлює правила керування процесом розроблення та дозволяє використовувати вже існуючі практики кодування, з коригуванням вимог або з внесенням тактичних змін.

Слід зазначити, що в чистому вигляді методології гнучкого програмування рідко використовуються командами розробників. Як правило, успішні команди застосовують корисні прийоми та властивості декількох процесів, підлаштовуючи їх під конкретне уявлення команди про гнучкість процесу розроблення.

3. Реалізація концепції керування програмним проектом на всіх етапах життєвого циклу у Visual Studio 2012

Управління життєвим циклом додатків (application lifecycle management - ALM) – це концепція управління програмним проектом на всіх етапах його життя. Для реалізації цієї концепції компанія Microsoft пропонує рішення на основі VisualStudio та TeamFoundationServer (TFS). Технології ALM у Visual Studio дозволяють розробникам контролювати життєвий цикл створення ПЗ, скорочуючи час розроблення, усуваючи

видатки та впроваджуючи неперервний цикл реалізації бізнес-цінностей.

Управління життєвим циклом додатку у Visual Studio базується на наступних принципах:

- продуктивність (productivity);
- інтеграція (integration);
- можливість розширення (extensibility).

Продуктивність забезпечується можливістю сумісної роботи та керуванням складністю продукту. Всі елементи проекту (вимоги, задачі, тестові випадки, помилки, код і побудови) та звіти централізовано керуються через TFS. Інструменти візуального моделювання архітектури, можливості управління якістю коду, інструменти тестування дозволяють керувати складністю продукту.

Інтеграція забезпечується можливостями Visual Studio щодо надання всім учасникам проекту інформації про стан справ, що спрощує комунікацію між членами команди та забезпечує прозорість ходу процесу проектування.

Можливість розширення забезпечується API-інтерфейсом служб TFS та інтегрованим середовищем розроблення (integrated development environment - IDE). API-інтерфейс служб TFS дозволяє створювати власні інструменти та розширяти існуючі, а IDE – кінцевим користувачам та стороннім розробникам додавати інструменти з додатковими функціями.

При створенні програмного продукту необхідно спочатку спроектувати архітектуру, що покладається на архітектора програмного продукту. На основі архітектури здійснюється розроблення, що є призначенням розробника ПЗ. Створений продукт необхідно тестувати на його відповідність вимогам замовника, що здійснює тестувальник. Visual Studio і TFS забезпечують сумісну командну роботу архітектора, розробника та тестувальника, надаючи їм необхідний інструментарій та функційні можливості для виконання необхідних робіт.

У Visual Studio для архітектурного проектування використовуються *інструменти візуального проектування на основі мови UML*, призначені для:

- візуалізації архітектурних аспектів проектованої системи;

- створення моделей структури та поведінки системи;
- розроблення шаблонів для проектування системи;
- документування прийнятих рішень.

Діаграми UML дозволяють візуально описувати додаток, наочно представляти архітектуру та документувати вимоги до додатка.

Архітектурні інструменти у Visual Studio 2012 Ultimate дозволяють створювати *шість видів схем та документ-орієнтованих графів*:

- схема класів UML;
- схема послідовностей UML;
- схема варіантів використання UML;
- схема активності UML;
- схема компонентів UML;
- схема шарів.

Схеми (діаграми) класів UML описують об'єкти в прикладній системі. Діаграми класів відбивають ієрархію всередині додатку або системи та зв'язки між ними.

Схеми (діаграми) послідовностей UML показують взаємодію між різними об'єктами. Вони використовуються для демонстрації взаємодії між класами, компонентами, підсистемами або суб'єктами.

Схеми (діаграми) варіантів використання UML визначають функційність системи та описують з точки зору користувачів їх можливі дії з програмним продуктом. Дані діаграми визначають зв'язки між функційними вимогами, користувачами та основними компонентами системи.

Схеми (діаграми) активності UML описують бізнес-процес або програмний процес у вигляді потоку робіт через послідовні дії. Діаграми активності використовуються для моделювання логіки в конкретному варіанті використання або для моделювання подробиць бізнес-логіки.

Схеми (діаграми) компонентів UML описують розподіл програмних складових додатку, дозволяючи наочно відобразити на високому рівні структуру компонентів та служб. За допомогою цих схем можна візуалізувати компоненти та інші системи, показуючи

зв'язки між ними. В якості компонентів можуть виступати виконавчі модулі, DLL-бібліотеки та інші системи.

Схеми (діаграми) шарів використовуються для опису логічної архітектури системи. Діаграми шарів можуть використовуватись для перевірки того, що розроблений код відповідає високорівневому проекту на схемі шарів. Діаграми дозволяють перевіряти архітектуру додатка на відповідність базі коду.

Документ-орієнтовані графи дозволяють створити *граф залежностей*, який відбиває відношення між компонентами архітектурних артефактів. Графи залежностей забезпечують візуальні способи перевірки коду, аналізу залежностей між файлами.

Основним засобом розроблення у VisualStudio2012 є *інтегроване середовище розроблення (IDE)*. IDE-середовище інтегроване із засобами модульного тестування та забезпечує можливості виявлення неефективного, небезпечного або погано написаного коду, керування змінами та модульне тестування як коду, так і бази даних.

Якість програмного продукту визначається за декількома критеріями: якісний програмний продукт повинен відповідати функційним та нефункційним вимогам, відповідно до яких він створювався, мати цінність для бізнесу, відповідати очікуванням користувачів.

У ЖЦ управління додатками якість повинна відстежуватись на всіх етапах ЖЦ ПЗ. Вона починає формуватись із визначення необхідних вимог. При заданні вимог необхідно вказувати бажану функційність та способи перевірки її досягнення.

Якісний програмний продукт повинен мати високу споживацьку якість, незалежно від галузі застосування: внутрішнє використання розробником, бізнес, наука та освіта, медицина, комерційні продажі, соціальна сфера, розваги та ін. Для користувача програмний продукт повинен задовольняти певному рівню його потреб.

Важливим аспектом створення якісного ПЗ є забезпечення нефункційних вимог, таких як зручність в експлуатації, надійність,

продуктивність, захищеність, зручність супроводу. Надійність ПЗ визначає здатність без збоїв виконувати задані функції в заданих умовах протягом заданого відрізку часу. Продуктивність характеризується часом виконання заданих транзакцій або тривалих операцій. Захищеність визначає ступінь безпеки системи від пошкоджень, втрати, несанкціонованого доступу та злочинної діяльності. Зручність супроводу визначає легкість, з якою обслуговується продукт, в плані простоти виправлення дефектів, внесення коректив для відповідності новим вимогам, керування зміненим середовищем.

Управління ЖЦ програмного продукту допомагає розробникам цілеспрямовано добиватись створення якісного ПЗ, уникати втрат часу на переробку, повторне проектування та перепрограмування ПЗ.

Тестування програмного продукту дозволяє протягом всього ЖЦ ПЗ гарантувати, що програмні проекти відповідають заданим параметрам якості. Головна мета тестування – визначити відхилення в реалізації функцій них вимог, виявити помилки у виконання програм та виправити їх якомога раніше в процесі виконання проекту.

Важливим інструментом розробника ПЗ є *модульне тестування*, яке реалізується і середовищі UnitTestFramework. Призначенням модульних тестів є перевірка того, що код працює вірно з точки зору програміста. Модульні тести формуються на більш низькому рівні, ніж інші види тестування, і перевіряють чи працюють функції, які лежать в їх основі, так, як очікується. Для модульного тестування використовується метод прозорої скриньки, для якого потрібне знання внутрішніх структур.

Модульні тести допомагають виявити проблеми проектування та реалізації. Крім того, модульний тест є гарною документацією з використання проектованої системи. Хоча модульне тестування вимагає додаткового програмування, але його застосування окупається за рахунок скорочення витрат на від лагодження додатку.

Модульні тести є важливим елементом регресійного тестування. *Регресійне тестування* представляє собою повторне

тестування частини програми після внесення в неї змін або доповнень. Мета регресійного тестування – виявлення помилок, які можуть з'явитись при внесення змін до програми.

У VisualStudio 2012 є *функція "Анализ покриття коду"*, яка проводить моніторинг того, які рядки коду виконувались під час модульного тестування. Результатом аналізу покриття коду є виявлення ділянок коду, які не покриті тестами.

Важливим аспектом створення якісного програмного продукту є дотримання розробниками правил та стандартів організації у написанні коду. У VisualStudio 2012 є *функції аналізу коду*, які дозволяють проаналізувати код, знайти типові помилки, порушення стандартів та запропонувати заходи з усунення помилок та порушень. Набори правил аналізу коду постачаються із VisualStudio 2012. Розробники можуть налаштувати свої проекти на певний набір правил, а також додати свої специфічні правила аналізу коду.

В процесі аналізу коду використовуються метрики коду, які дають кількісні оцінки різних характеристик коду. Метрики дозволяють визначити складність коду та його ізольовані ділянки, які можуть призвести до проблем при супроводі додатку. У VisualStudio 2012 використовуються наступні *метрики коду*:

1) складність організації циклів – визначає кількість різних шляхів коду;

2) глибина наслідування – визначає кількість рівнів в ієрархії наслідування об'єктів;

3) об'єднання класів – визначає кількість класів, на які є посилання;

4) рядки коду – визначає кількість рядків коду у виконуваному методі;

5) індекс зручності підтримки – оцінює простоту обслуговування коду.

Для аналізу продуктивності та ефективності використання ресурсів додатковими інструментами у VisualStudio 2012 є *інструменти профілювання*. Профілювання представляє собою процес спостереження та запису показників про поведінку додатку. Інструменти профілювання (профілювальними) дозволяють

виявити у додатку проблеми із продуктивністю. Такі проблеми, як правило, пов'язані із кодом, який виконується повільно, нефективно або надмірно використовує системну пам'ять. Профілювання, як правило, використовується для виялення ділянок коду, які під час виконання додатку виконуються часто або тривалий час.

Профілювальники бувають із вибіркою або з інструментуванням. Профілювальники із вибіркою роблять періодичні знімки виконуваного додатку та записують його стан. Профілювальники з інструментуванням додають маркери відстежування на початок та кінець кожної досліджуваної функції. В процесі роботи профілювальника маркери активізуються, коли потік виконання програми входить до досліджуваних функцій та виходить з них. Профілювальник записує дані про додаток і про те, які маркери були зачеплені під час виконання додатку.

Більшість корпоративних додатків працює із базами даних, що визначає необхідність розроблення та тестування додатків сумісно з базами даних командою проекту. У VisualStudio 2012 є *інструментарій створення баз даних* і розгортання змін у них. Для цього використовується автономне розроблення схем баз даних, яке дозволяє вносити зміни до схем без підключення до виробничої бази даних. Після внесення змін у середовище розроблення, VisualStudio 2012 дозволяє протестувати їх у самому середовищі та/або виділеному середовищі тестування. Крім того, VisualStudio 2012 дозволяє згенерувати псевдо реальні дані для проведення тестів. При позитивних результатах тестування VisualStudio 2012 дозволяє згенерувати сценарії для оновлення виробничої БД.

Цикл розроблення БД додатку складається із наступних кроків:

1) переведення схеми БД у автономний режим;

2) ітеративне розроблення додатку із БД;

3) тестування схеми БД;

4) побудова та розгортання БД та додатку.

Для вдосконалення процесу відлагодження додатків у VisualStudio 2012 є *функція інтелектуального відстежування*

роботи програми IntelliTrace. Функція IntelliTrace конфігурується за допомогою наступних розділів:

- Загальне (General) – включення/відключення функції, задання запису лише подій або додаткової інформації;
- Додаткове (Advanced) – задання розташування журналу та його розміру для генерованого файлу;
- Події IntelliTrace (IntelliTrace Events) – перераховуються всі події діагностики, які будуть збиратись під час відлагодження додатку;
- Модулі (Modules) – список модулів, для яких необхідно збирати дані в процесі відлагодження додатку.

При записі подій додатку відбувається їх перехоплення при роботі додатку, інформація про події фіксується в журналі. При відлагодженні з IntelliTrace можна призупинити інтерактивний сеанс відлагодження та переглянути події або виклики. Також є можливість зупинки виконання додатку та покрокового руху назад та вперед в інтерактивному сеансі відлагодження, а також відтворення записаного сеансу відлагодження.

Тестування додатку є необхідним етапом управління ЖЦ додатку. Тестування виконує розробник в процесі створення коду додатку, а також тестувальник при перевірці якості розроблюваного програмного продукту. VisualStudio 2012 Ultimate надає розробнику інструмент для створення та використання модульних тестів (низькорівневні програмні тести, які дозволяють швидко перевірити наявність логічних помилок в методах класів), навантажувальних тестів (використовуються для дослідження роботоздатності додатку шляхом моделювання множини користувачів, які працюють з програмою одночасно) та веб-тестів продуктивності, а також тестів для інтерфейсу користувача (дозволяють автоматично сформувати код тесту, шляхом запису дій користувача при роботі із додатком, і потім виконувати ці тести автоматично).

Центральне місце у *архітектурі засобів тестування VisualStudio 2012* посідає платформа модульного тестування, яка забезпечує підключення плагінів тестування (MS-TestManaged і MS-TestNative). Плагіни сторонніх розробників (NUnit, xUnit.net,

MbUnit) можна підключити до платформи юніт-тестування. Доступ до засобів тестування може здійснюватись з оглядача тестів, командного рядка та при побудові додатку через TeamBuild.

Для тестувальників у VisualStudio 2012 Ultimate є спеціалізований інструмент – *MicrosoftTestManager*, який дозволяє створювати плани тестування, формувати, додавати та видаляти тестові випадки, визначати та керувати фізичними та віртуальними тестовими середовищами, виконувати ручні та автоматичні тести.

Для тестувальників та розробників ПЗ VisualStudio 2012 включає диспетчер віртуального середовища *LabManagement*. Інструмент тестування LabManagement дозволяє створити інфраструктуру, яка максимально близько емулює реальне середовище планованого використання програмного продукту. Такі середовища можуть використовуватись для виконання автоматичних побудов, автоматизації тестів та виконання розроблюваного коду. Лабораторія тестування LabManagement інтегрована із середовищем TeamFoundationServer 2012. Адміністрування віртуального середовища LabManagement проводить *диспетчер MicrosoftSystemCenterVirtualMachineManager* (SCVMM), за допомогою якого виконують необхідні налаштування віртуальної тестової лабораторії.

У VisualStudio 2012 та TeamFoundationServer додано засіб взаємодії із користувачами розроблених програмних продуктів, яке на запит дозволяє сформувати відгук користувача в БД для наступного опрацювання командою проекту.

Протягом всього ЖЦ ПЗ застосовуються різні типи тестування для гарантування того, що проміжні версії відповідають заданим показникам якості.

Якість коду визначається також і тим, наскільки важко або легко вносити зміни в код та наскільки код доступний для розуміння. Створений програмний додаток може виконувати необхідні функції, але мати проблеми із внесенням змін або розумінням створеного коду. В такому випадку ПЗ не можна назвати якісним, оскільки на етапі його супроводу можуть виникнути проблеми з його модифікацією при зміні вимог користувача. Для покращення якості програмного коду

використовують *рефакторинг* – процес зміни ПЗ таким чином, що їїх зовнішня поведінка не змінюється, а внутрішня структура покращується.

Неякісний дизайн коду можна визначити за наступними ознаками:

- жорсткість – важке внесення змін в код;
- крихкість – пошкоджуваність програми в багатьох місцях із внесенням єдиної зміни;
- відсталість (інертність) – зусилля та ризики, пов'язані із спробою відокремити корисні для інших систем ділянки коду, надто великі;
- непотрібна складність – програма містить невикористовувані елементи;
- непотрібні повторення – повторення фрагментів коду у програмі;
- непрозорість – характеризує важкість коду для розуміння.

Для створення якісного дизайну коду доцільно використовувати деякі *принципи та паттерни проектування ПЗ*:

- принцип єдиного обов’язку – визначає, що в класу повинна бути лише єдина причина для зміни, тому класу доцільно доручати лише один обов’язок;
- принцип відкритості /закритості – визначає, що програмні сутності повинні бути відкритими для розширення, але закритими для модифікації;
- принцип підстановки – визначає, що повинна бути можливість замість базового класу підставити будь-який його підтип;
- принцип інверсії залежностей – визначає, що модулі верхнього рівня нее повинні залежати від модулів нижнього рівня (і ті, й інші повинні залежати від абстракцій), а абстракції не повинні залежати від деталей (деталі повинні залежати від абстракцій);
- принцип розділення інтерфейсів – визначає, що клієнти повинні знати лише про абстрактні інтерфейси, які мають властивість зчеплення;

- патерни проектування – пропонують універсальні, перевірені практикою рішення. Зі списку патернів слід виділити ті, які доцільно застосовувати при гнучкому розробленні ПЗ. Використання патернів дозволяє створювати ПЗ, яке легко модифікувати та супроводжувати.

4. Функціональні можливості та архітектура TeamFoundationServer 2012 (TFS)

Microsoft Visual Studio Team Foundation Server (TFS) призначений для забезпечення сумісної роботи колективів розробників ПЗ. Team Foundation Server надає наступні *функційні можливості*:

- управління проектами;
- відстежування робочих елементів;
- контроль за версіями;
- управління тестовими випадками;
- автоматизація побудови;
- звітність.

Архітектура Team Foundation Server 2012 є трирівневою сервіс-орієнтованою (рис.12.1). Рівень додатку підтримується веб-сервеорм ASP.NET, розташованим в середовищі IIS. Рівень даних підтримується сервером баз даних MS SQLServer 2012.

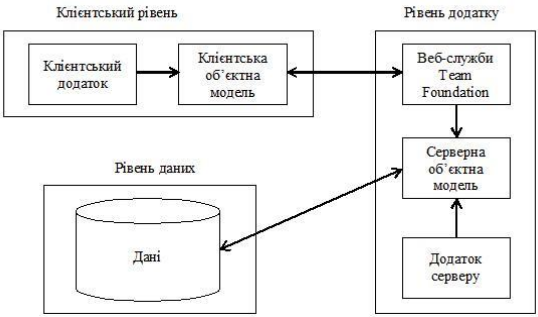


Рис.12.1 - Архітектура Team Foundation Server

Team Foundation Server – це, з логічної точки зору, веб-додаток, який складається з декількох веб-служб, які виконуються на *рівні додатку*. Дані служби реалізують функційність TFS. До складу веб-служб рівня додатку входять: управління версіями, служба побудови, відстежування робочих елементів, служби платформи TFS та лабораторії тестування LabManagement. Серверна об'єктна модель є інтерфейсом прикладного

програмування для TFS. За необхідності розширення функційності TFS доцільно будувати на базі серверної об'єктної моделі.

Рівень даних складається з декількох реляційних баз даних та сховища даних: бази даних конфігурації серверу, бази даних аналітики, бази даних колекції командних проектів та сховища даних. Інформація командних проектів зберігається в реляційних базах даних, і через заплановані інтервали часу розташовується у сховище даних.

Клієнтський рівень може реалізовуватись в оболонці VisualStudio та у веб-броузері. Клієнтський рівень взаємодіє з рівнем додатків через серверну об'єктну модель та використовує веб-служби. Крім того, клієнтський рівень включає інтеграцію з Microsoft Office.

5. Способи розгортання TFS на одному або декількох серверах, в одному домені, робочі групі або в декількох доменах. Шаблони командних проектів TFS, області керування командними проектами

Для Team Foundation Server можна виконати *розгортку декількома способами*: на одному сервері; на декількох серверах; в одному домені, робочій групі або в декількох доменах.

У *найпростішій серверній топології* для розташування компонентів, які складають логічні рівні Team Foundation, використовується один фізичний сервер (рис.12.2). При встановленні TFS з одним сервером всі компоненти (додаток TeamFoundationServer, SQLServer, ReportingServices і WidowsSharePointServices) встановлюються на одному комп'ютері. Така конфігурація передбачає виконання побудови (TeamFoundationBuild) та тестування або на сервері, або на клієнтських комп'ютерах. Загальна кількість користувачів для такої конфігурації, як правило, не більше 50.

Рис.12.2 - Найпростіша серверна топологія TFS

У *простій серверній топології* для розташування компонентів, які складають логічні рівні Team Foundation, також використовується один фізичний сервер (рис.12.3). Однак в такій топології враховується також додаткове навантаження на процесорні потужності, яке створюється програмним забезпеченням для побудови та тестування. На рис.12.3 веб-служби і бази даних для Team Foundation розташовуються на одному фізичному сервері, але служби побудови встановлюються на окремий комп'ютер. До Team Foundation Servermожна одержати доступ з клієнтських комп'ютерів, які належать до тієї ж робочої групи або до того ж домену. В даній конфігурації контролер побудови для виконання Team Foundation Build та контролер тестування встановлюються на окремих комп'ютерах. Загальна кількість користувачів для такої конфігурації, як правило, не більше 100.

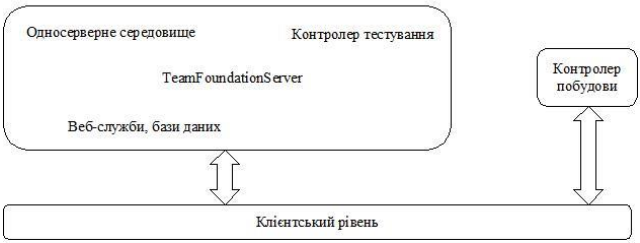


Рис.12.3 - Проста серверна топологія TFS

У *серверній топології середньої складності* використовуються два або більше серверів для розташування логічних компонентів на рівні даних та додатків Team Foundation (рис.12.4). На рис.12.4 служби рівня додатків для Team Foundation Server розгортаються на одному сервері, а бази даних для TFS встановлюються на окремому сервері. На окремих серверах розташовуються веб-додаток SharePoint та екземпляр служб звітів SQL Server. Портал джля кожного командного проекту розташовується у веб-додатку SharePoint. Сервер Team Foundation Build та тестові контролери команди розгортаються на додаткових

серверах. Загальна кількість користувачів для такої конфігурації, як правило, не більше 1000.

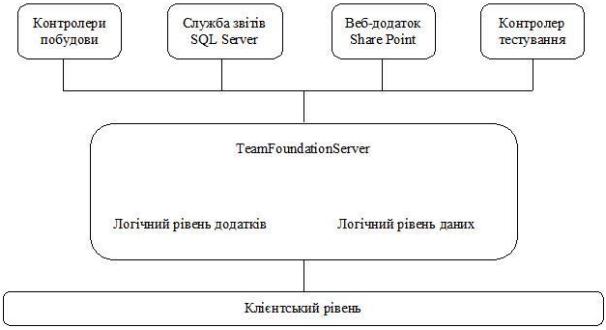


Рис.12.4 - Серверна топологія TFS середньої складності

Для управління командним розробленням Team Foundation Server містить одну або декілька колекцій командних проектів, кожна з яких може містити ноль або більше проектів.

Командний проект представляє колекцію робочих елементів, коду, тестів та побудов, які охоплюють всі артефакти, використовувані у життєвому циклі програмного проекту. Командний проект будується на основі шаблону, який представляє набір XML-файлів, які містять деталі того, як повинен здійснюватись процес. У TFS 2012 є наступні *шаблони проектів*:

- MSF for CMMI Process Improvement 6.0, призначений для великих команд з суворо формальним підходом до управління проектами на основі моделі CMM/CMMI;

- MSF for Agile Software Development 6.0, який визначає гнучкий підхід до управління проектами розроблення ПЗ;

- Microsoft Visual Studio Scrum 2.2., призначений для невеликих команд (до 7-10 учасників), які використовують гнучку методологію та термінологію Scrum.

При створенні командного проекту надається *можливість налагоджування та управління наступними областями проекту*:

- відстежування робочих елементів дозволяє визначити початкові типи робочих елементів, загальні запити та запити користувача, створити початкові робочі елементи;

- класифікації дозволяють визначити початкові області та ітерації проекту;

- веб-сайт на базі SharePoint дозволяє керувати бібліотекою документів проекту;

- система контролю версій дозволяє визначити початкові групи безпеки, дозволи, правила повернення версій;

- звіти формуються у папки, створювані на сайті командного проекту;

- групи та дозволи включають групи безпеки TFS та дозволи для кожної групи;

- побудови включають стандартні процеси побудови для створення нових побудов;

- лабораторія включає стандартні процеси лабораторії для використання, а також дозволи лабораторії для кожної групи;

- управління тестуванням включає стандартні конфігурації тестування, змінні, параметри та стани дозволів.

Робочими елементами у TeamFoundationServer є користувацький опис функційності, задачі, тестові випадки, помилки та перешкоди. Цими елементами потрібно керувати для виконання програмного проекту. Система відстежування робочих елементів дозволяє створювати робочі елементи, відстежувати їх стан, формувати історію їх зміни. Всі дані по робочих елементах зберігаються у базі даних TFS.

TeamFoundationServer включає централізовану *систему контролю за версіями*. Система контролю за версіями TFS надає наступні можливості:

- атомарні повернення – зміни, які вносяться у файли, запаковуються з «набором змін»; повернення файлу у наборі змін представляють собою неподільну транзакцію, чим гарантується узгодженість бази коду;

- асоціація операцій повернення з робочими елементами, що дозволяє відстежувати вимоги від початкової функції до задач та повернення у систему контролю за версіями коду;

- розгалуження та об’єднання при розробленні коду;

- набори відкладених змін дозволяють створювати копії коду, не записуючи їх у головне сховище системи контролю за версіями;

- використання підписів дозволяє позначити набір файлів у певній версії за допомогою текстового підпису;

- одночасне видобування дозволяє декільком членам команди одночасно редагувати файл з наступним об'єднанням змін;

- відстежування історії файлів;

- політики повернення, які надають можливість виконати код для перевірки припустимості повернення;

- примітки при поверненні дозволяють формувати метадані про повернення;

- проксі-сервер дозволяє оптимізувати роботу з регіональними центрами розроблення.

Побудова проекту програмного продукту у TFS реалізує сервер Team Foundation Build. Він дозволяє стандартизувати інфраструктуру побудов для команди проекту. *Побудова проекту може виконуватись у наступних режимах*:

-ручний, при якому побудова вручну ставиться у чергу побудов;

- неперервна інтеграція, яка дозволяє виконувати побудову при кожному поверненні в систему контролю версій;

- прокрутка побудов, при якому повернення групуються разом, щоб в побудову включались зміни за певний період часу;

- умовне повернення, при якому поверненн приймаються лише в тому випадку, якщо успішно пройшли злиття та побудови відправлених змін;

- розклад, який дозволяє налагодити час побудови на певні дні тижня.

Розробник має можливість взаємодіяти з ключовими службами TeamFoundationServer через:

- командний оглядач (TeamExplorer) VisualStudio 2012, який надає доступ до функцій керування життєвим циклом додатків, включаючи командні проекти, аналіз коду, керування версіями та побудовами;

- доступ через веб (Team Web Access), завдяки якому надається веб-доступ до функцій керування ЖЦ додатків, включаючи командні проекти, робочі групи, керування проектами, керування версіями та побудовами;

- MicrosoftExcel, завдяки якому надається можливість визначати та змінювати обочі елементами масивом, а також створювати звіти на основі запиту робочого елемента;

- MicrosoftProject, завдяки якому надається можливість керувати планом проекту, задачами розкладу, ресурсами, формувати календар проекту, діаграми Ганта та представлення ресурсів;

- консоль адміністрування Team Foundation Server, завдяки якій здійснюється налагодження TFS;

- інструменти командного рядка;

- сторонні засоби інтеграції.

6. Питання створення командного проекту, зміст програмної інфраструктури проекту, склад і призначення робочих елементів

При командному розробленні ПЗ важливими питаннями є планування робіт, складання розкладу, керування областю проекту, комунікації, складання звітів, аналіз та постійне вдосконалення процесу. Для вирішення цих питань TeamFoundationServer пропонує *наступні інструменти:*

- шаблон процесу, який визначає процес, використовуваний командним проектом;

- керівництво з процесу, яке містить опис шаблонів процесу;

- колекція командних проектів, яка представляє собою контейнер для декількох командних проектів;

- командний проект, який зберігає та організовує дані про весь життєвий цикл розроблення ПЗ;

- портал проекту/панелі моніторингу, на яких надається інформація по проекту для всіх членів колективу;

- елементи планування для управління списками вимог як на рівні проекту, так і на рівні ітерації;

- відстежування робочих елементів, яке дозволяє відстежувати стан робочих елементів та іншу інформацію, пов'язану з ними;

- звітність, яка дозволяє формувати звіти під час ЖЦ проекту;

- інтеграцію з MicrosoftProject та MicrosoftExcel для управління проектами з середовища MicrosoftOffice.

Розроблення ПЗ починається із створення командного проекту. Командний проект містить інформацію про кожен крок ЖЦ розроблення ПЗ, включаючи вимоги користувачів, тестові випадки, помилки, перешкоди, побудови.

При створенні командного проекту необхідно задаим його ім'я, опис, визначитись із шаблоном процесу, вимогами до системи контролю за версіями та необхідністю створення порталу для проекту.

В результаті створення командного проекту Team Foundation Server формує наступні папки:

- Моя работа – папка, в якій задаються виконувана та призупинена роботи, доступні робочі елементи та активні запитані задачі аналізу коду;

- Ожидающие изменения - папка, в якій забезпечується можливість збереження змін коду в базі даних TFS та видобування проектних рішень для редагування користувачем;

- Рабочие элементы проекта - папка, в якій зберігаються згруповані дані про поточні робочі елементи проекту;

- Построения - папка, в якій формуються та зберігаються визначення та результати побудови додатків;

- Отчеты - папка, в якій є підпапки із звітами та посилання на стандартні звіти;

- Документы - папка, в якій зберігаються документи проекту;

- Параметры - папка, в якій зберігаються посилання для переходу до вікон задання параметрів проекту (безпека, склад груп, система управління версіями, області та ітерації робочих елементів, параметри порталу та оповіщення проекту).

Після створення командного проекту керівник формує колектив (команду). Для кожного члена команди керівник проекту визначає доступ до проекту в цілому та до окремих артефактів, важливих для роботи кожного члена команди.

Для структурування проекту використовуються області та ітерації. Області можуть визначатись, виходячи з певного функційного призначення етапу робіт, а ітерації – у вигляді набору робіт на заданому часовому інтервалі.

При плануванні командного проекту ключовими сутностями є *робочі елементи*. Залежно від шаблону командного проекту набір та найменування робочих елементів дещо відрізняються. Робочий елемент – запис, створений у Visual Studio Team Foundation Server для задання визначення, призначення, пріоритету та стану елементу роботи. Робочі елементи можуть включати найменування, опис призначення, стан, кому призначено, цінність для бізнесу, приналежність до робочої області.

Так, *для гнучкої методології Agile робочими елементами є*:

- Користувацький опис функційності (UserStory) – вимога користувача, яку необхідно виконати при реалізації проекту; такі робочі елементи можуть бути пов'язані між собою, а також з дочірніми (Задача, Помилка) або батьківськими елементами.

- Задача (Task) – створюється в проекті для призначення та виконання роботи, надає деталі реалізації для вимог користувача;

- Помилка (Bug) – використовується для відстежування та моніторингу роблем у програмному продукті;

- Перешкода (Issue) – використовується дл фіксації у проекті подій або об'єктів, які створюють проблеми у виконанні проекту та повинні бути усунені під час поточної або майбутньої ітерації;

- Тестовий випадок (TestCase) – описує умови перевірки вірності виконання програмним продуктом вимог користувача;

- Невиконана робота – використовується для запису вимог користувача при плануванні командного проекту.

Розроблення програмного коду. Задачі призначаються розробникам програмного коду. Розробники проводять кодування та модульне тестування задач у середовищі Visual Studio.

Розроблювані коди знаходяться під контролем системи керування версіями у сховищі коду. Роботоздатні коди задач збираються в побудові за допомогою Team Service Build і передаються у систему керування версіями на Team Foundation Server. В процесі роботи розробник проводить видобування (checkingout) файлів з кодами задач додатку в свою робочу область на інструментальному комп'ютері. Після виконання певного етапу робіт розробник здійснює повернення (checkingin) у сховище TFS коду. При поверненні коду формується набір змін (changeset), який мітсить всю інформацію, пов'язану із поверненням (посилання на робочі елементи, виправлення, примітки, політики та дані про власника, дату та час). Це дає можливість розробникам переглядати різні версії файлів та аналізувати внесені зміни.

Тестування. Тестувальники на основі тестових випадків готують тести для заданих вимог користувача. Для тестування файли коду видобуваються зі сховища TFS і підлягають тестуванню. У випадку виявлення помилки формується робочий елемент Помилка, який спрямовується розробнику для виправлення. Якщо тести проходять, то відповідна вимогам користувача вважається виконаною та для неї встановлюють стан «Готово».

Звіти. Team Foundation Server 2012 має потужну систему збирання інформації під час проекту та підготовки звітів. Звіти призначеня як для керівників проекту, так і для членів команди. Система звітності базується на сховищі даних TFS. Дані можуть бути представлені у вигляді звітів, які дозволяють переглядати метрики проекту. Система звітів дозволяє відстежувати робочі елементи, побудови, статистику системи контролю за версіями, результати тестів, індикатори якості проекту. Team Foundation Server надає набір звітів, адле є можливість створювати також і звіти користувача.

У TFS є *три схвоища даних*:

- операційне сховище;
- сховище даних;
- OLAP-куб.

Операційне сховище представляє собою набір реляційних баз даних для зберігання такої інформації, як сирцевий код, звіти побудови, результати тестів та відстежування робочих елементів.

Схвоище даних призначене для виконання запитів та створення звітів. Сховище даних одержує дані з операційного сховища через певні проміжки часу. Сховище даних концептуально побудоване за схемою «зірка».

OLAP-куб TeamFoundationServer є багатовимірною базою даних, яка містить агреговані дані для підготовки аналітичних звітів. OLAP-куб одержує дані зі сховища даних через заплановані інтервали часу. Дані багатовимірної бази даних можуть використовуватись різними клієнтськими додатками, включаючи Microsoft Excel і конструктор SQL-звітів.

Team Foundation Server включає два *набори звітів*:

- звіти Microsoft Excel Reports
- звіти служби звітності SQL Reporting Services Reports.

MicrosoftExcel дозволяє створювати звіти з OLAP-кубу TFS або використовуючи запити робочих елементів проекту. Звіти можуть бути опубліковані на SharePoint-порталі проекту.

7. Аналіз методології Scrum, робочі елементи шаблону Microsoft Visual Studio Scrum 2.2

Методологія Scrum представляє собою ітеративний процес розроблення ПЗ. При такому розробленні для програмного продукту створюється багато послідовних випусків, в яких поступово додається потрібна функційність. Ітеративний підхід дозволяє по завершенню поточної ітерації продемонструвати замовнику роботоздатний програмний продукт, можливо з обмеженою функційністю, одержати відгук, зауваження та додаткові вимоги, які будуть варховані у наступних ітераціях. *Основними артефактами в методології Scrum* є робочі елементи, звіти, книги та панелі моніторингу.

Робочі елементи використовуються для відстежування, спостереження за станом ходу розроблення ПЗ та створення звітів. *Робочий елемент* – це запис, який створюється у Visual Studio Team Foundation Server для задання визначення, призначення, пріоритету та стану елементу роботи. Для шаблону Microsoft Visual Studio Scrum 2.2 визначаються наступні *типи робочих елементів*:

- Невиконана робота;
- Помилка;
- Задача;
- Перешкода;
- Тестовий випадок.

В методології Scrum вимоги користувача, які визначають функційність продукту, задаються *елементами заділу роботи продукту* (Product Backlog Item - PBI). Елементи заділу роботи продукту, які називають також *елементами роботи* (ЕР), представляють собою короткий опис функцій продукту та оформляються у довільній формі у вигляді коротких приміток. Спочатку задаються найбільш важливі та зрозумілі всім вимоги користувача. ЕР можуть деталізуватись у вигляді задач. В процесі створення програмного продукту ЕР можуть уточнюватись, додаватись або видалятись зі списку вимог.

Цикл випуску продукту в Scrum складається з множини ітерацій, які називаються *спринтами*. Спринт має фіксовану тривалість, як правило, 1-4 тижні. Елементи роботи, включені до

чергового спринту, не підлягають зміні до його завершення. Хоча спринт завершується підготовкою роботоздатного програмного продукту, його поточної функційності може бути недостатньо для оформлення випуску, який має цінність для замовника. Тому випуск роботоздатного програмного продукту, який замовник може використовувати, включає, як правило, декілька спринтів.

8. Організація колективу у методології Scrum. Життєвий цикл проекту

Організація команди в методології Scrum визначає три ролі:

- власник продукту (Product owner);
- керівник (ScrumMaster);
- члени команди (Team members).

Власник продукту відповідає за все, що пов'язано зі споживацькими якостями програмного продукту. Він визначає вимоги користувача, аналізує їх реалізацію, має право змін вимог, контролює якість продукту. Він може бути представником замовника в команді або членом команди розробників, який представляє інтереси замовника. Власник продукту виконує наступні основні задачі:

- визначення та пріоритезація вимог/функцій, тобто елементів робіт та задач;
- планування спринтів та випусків;
- тестування вимог/функцій.

Керівник відповідає за стан та координацію проекту, продуктивність команди та усунення перешкод, які заважають проекту. В обов'язки керівника входить:

- проведення щоденних Scrum-зборів;
- залучення співробітників поза командою;
- стимулювання ефективного спілкування членів команди;
- визначення розміру команди.

Члени команди відповідає за розроблення програмного продукту високої якості. Вони повинні володіти навичками в проектуванні та архітектурі програмного продукту, бізнес-аналізі, програмуванні, тестуванні, налагодженні баз даних та проектуванні інтерфейсу користувача. Члени команди приймають участь у

плануванні спринтів. Команда може включати досвідчених розробників та новачків, які в процесі роботи повинні вдосконалюватись при обміні знаннями з іншими членами команди. Члени команди відповідають за наступні задачі у проекті:

- обов'язкове виконання елементів робіт, включених в поточний спринт;
- акцент на взаємозв'язаних задачах спринту;
- вдосконалення команди.

Інструментальна та методична підтримка гнучкого підходу до створення програмних продуктів Scrum, реалізована у VisualStudio 2012, дозволяє керувати життєвим циклом проекту ПЗ (рис.13.1).

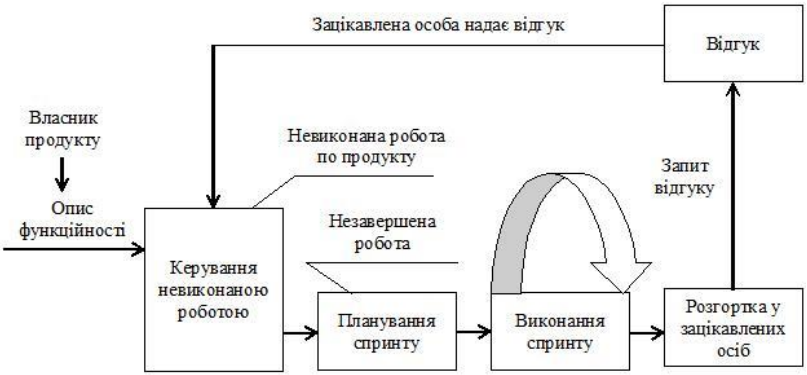


Рис.13.1 - Життєвий цикл проекту ПЗ

На початку проектування власник продукту та замовник формують *концепцію програмного продукту*, яка відображає, для кого призначено продукт, які переваги отримають користувачі та які існують конкуренти. Концепція продукту пов'язується з галуззю проекту та обмеженнями. Галузь проекту визначає масштаб робіт, а обмеження – умови, якими керуватимуться для перших спринтів та випусків. Далі власник продукту створює список всіх потенційних функцій продукту – *«Невиконана робота по продукту» (ProductBacklog)*. Невиконана робота по продукту,

яку надалі називатимемо невиконана робота – НВР, містить список елементів роботи – користувацьких описів функційності.

Керування невиконаною роботою по проекту зводиться до підтримки елементів робіт в актуальному стані. Окремі елементи робіт списку НВР можуть додаватись або вилучатись в процесі створення ПЗ. Це є результатом того, що колектив отримує додаткову інформацію про нові вимоги замовника до проектованого програмного продукту, а замовник з'ясовує, як реалізуються його очікування.

Для *елементів робіт (ЕР)* власник продукту сумісно з командою проекту визначає пріоритети. При планування спринту в нього включають найбільш значущі, з точки зору власника продукту, вимоги користувача, які характеризуються найбільшою споживацькою цінністю. Обрані елементи робіт переміщають у список *«Незавершена робота» (SprintBacklog)*. Список «Незавершена робота» (НЗР) відбиває склад робіт планованого спринту. Список НЗР є результатом процесу планування спринту.

Координацією робіт у спринті займається керівник спринту (ScrumMaster). Він організовує процес пріоритезації задач спринту, розподілу задач між членами команди. Керівник спринту проводить збори з планування робіт, щоденні збори для короткого обговорення результатів роботи та проблем, оглядові збори в кінці спринту та випуску.

Щоденні Scrum-збори мають тривалість 15-30 хвилин. Метою таких зборів є виявлення проблем, які гальмують процес розроблення, і визначити дії з їх нейтралізації. Для простих проблем приймається рішення щодо їх усунення, а складні проблеми відкладаються на наступні спринти. Протягом щоденних Scrum-зборів керівник задає темп спринту, акцентує команду на найбільш важливих елементах списку невиконаних робіт. Кожний член колективу повідомляє, що було зроблено вчора, що буде робити сьогодні і які перешкоди є у роботі. Якщо на щоденних Scrum-зборах виникають питання, для вирішення яких необхідні фахівці, яких в колективі немає, тоді керівник бере на себе аналіз та можливі шляхи вирішення даного питання.

Результатом спринту є роботоздатне ПЗ, яке можливо володіє лише частиною необхідних функцій програмного продукту. Випуск спринту може бути розгорнутий у зацікавлених осіб для попереднього аналізу відповідності очікуванням замовника. Результатом відгуку є формування «Відгук» (FeedBack), що може призвести до змін вмісту списку «Невиконана робота по продукту». Після виконання всіх робіт за програмним продуктом, тобто обнуління списків вимог «Невиконана робота по продукту» та «Незавершена робота», готується фінальний випуск програмного продукту.

Список «Невиконана робота по продукту» є одним з ключових артефактів у методології Scrum. Успіх Scrum-команди багато в чому визначається якісним вмістом даного списку. Список НВР включає користувацькі описи функційності – елементи роботи, а також може включати нефункційні вимоги. Для створення списку НВР можуть застосовуватись різні клієнтські сервіси (рис.13.2).

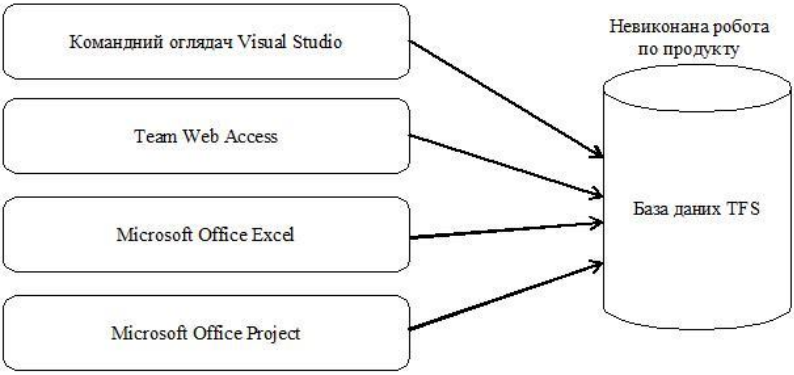


Рис.13.2 - Клієнтські сервіси TFS

Власник продукту на основі вимог та побажань клієнтів формує список функцій продукту у вигляді робочих елементів продукту, які розташовує у список «Невиконана робота по продукту». При створенні нового елементу невиконаної роботи для нього встановлюється стан «Новий» (рис.13.3).

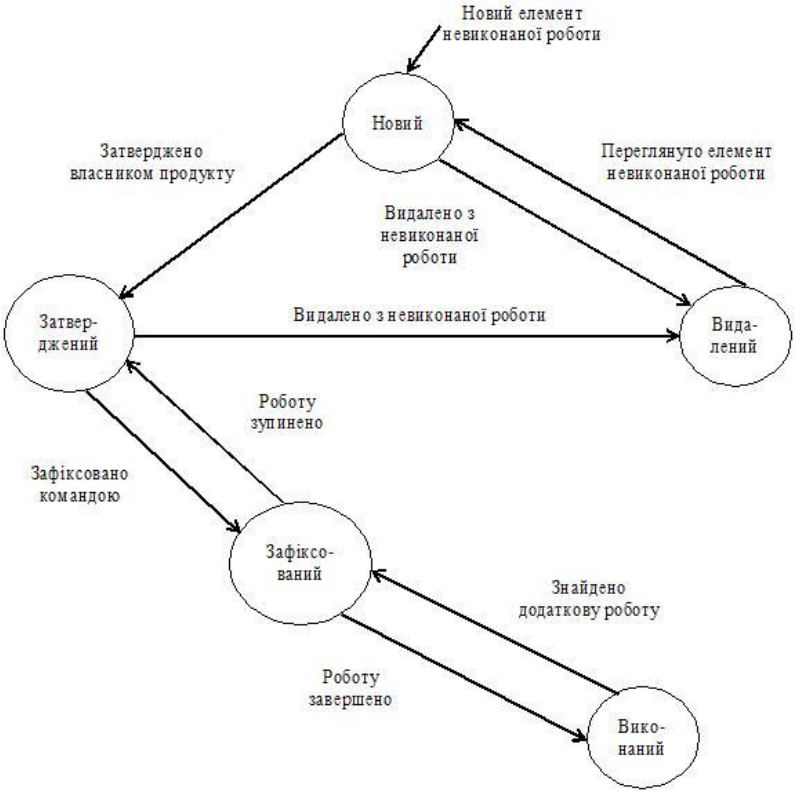


Рис.13.3 - Робочий процес елементу невиконаної роботи

Після встановлення елементу роботи пріоритету його стан змінюють на «Затверджений». На зборах з планування спринту команда переглядає найбільш пріоритетні елементи роботи і обирає ті, які будуть виконуватись у поточному спринті. Для елементів невиконаної роботи, які потрапили у поточний спринт, встановлюється стан «Зафіксований». Це означає той факт, що робочі елементи спринту не підлягають зміні до кінця спринту. При завершенні роботи за елементом його стан встановлюється як «Виконаний». Якщо для елементу роботи, який знаходиться у стані «Виконаний», виявляється додаткова робота, то цей елемент може

бути переведний у стан «Зафіксований». Для елементу роботи, який знаходиться у стані «Зафіксований», при виникненні проблем, які перешкоджають його завершенню у спринті, робота може бути призупинена і встановлено стан «Затверджений». Елемент невиконаної роботи може бути видалений зі списку «Невиконана робота по продукту» за рішенням власника продукту. Це може відбутись як зі стану «Новий», так і зі стану «Затверджений». Для видаленого елемента встановлюється стан «Видалений». В результаті перегляду елементу роботи зі станом «Видалений» для нього знов можливе переведення у стан «Новий».

Список «Невиконана робота по продукту» є головним документом для Scrum-команди. На основі даного списоу команда створює інші робочі елементи, які скаладають спринти й випуски. Для елементів невиконаної роботи команда створює задачі та тестові випадки. Задачі деталізують елемент роботи та визначають конкретну реалізацію вимог користувача. Тестові випадки необхідні для перевірки відповідності функційності коду вимогам користувача. Якщо тестовий випадок не проходить, то створюється робочий елемент «Помилка». При блокуванні задачі через неможливість її виконання в поточному спринті створюють робочий елемент «Перешкода». Scrum-команда може створювати допоміжні робочі елементи по відношенню до елементів, на які вони впливають (задачі та тестові випадки), та пов’язувати ці елементи (рис.13.4).

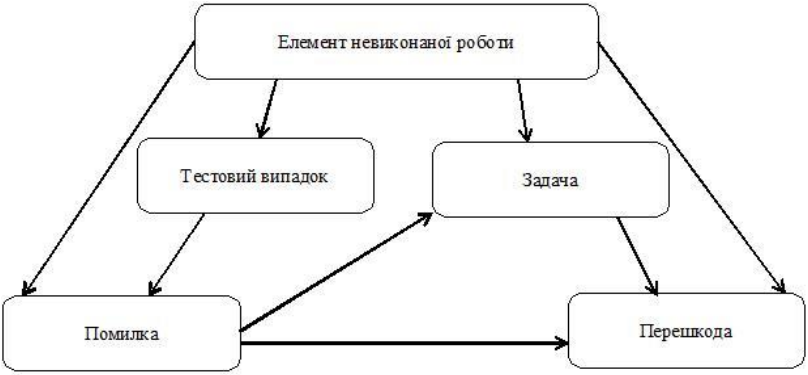


Рис.13.4 - Зв‘язок між робочими елементами

Для відстежування ходу виконання проекту, можна створювати звіти, які відображають найбільш важливі дані для поточного проекту. В процесі створення ПЗ можна користуватись стандартними звітами або створювати власні звіти. Звіти можна створювати, налагоджувати і перевіряти за допомогою MS Excel, MS Project або служб Reporting ServicesSQL Server.

Методологія Scrum має наступні позитивні сторони:

- користувач іпочинають бачити систему вже через декілька тижнів і можуть виявляти проблеми на ранніх стадіях розроблення програмного продукту;

- інтеграція технічних компонентів відбувається під час кожного спринту і тому пробоеми проекту (якщо вони виникають) виявляються практично одразу;

- в кожному спринті команда фокусується на контролюванні якості;

- гнучка робота із змінами в проекті на рівні спринту.

Висновки

Microsoft Solutions Framework є методологією розроблення ПЗ, яка представляє собою узагальнення кращих проектних практик, які використовувались командами розробників Microsoft. Методологія MSF є складовою частиною продукту Visual Studio Team System. В основі методології MSF лежить ітеративний інтегрований підхід до створення та впровадження IT-рішень, який базується на фазах та віхах. В моделі команди MSF відсутній офіційний лідер, всі відповідають за проект в рівній мірі. В залежності від розміру та складності проекту модель команд MSF припускає масштабування.

Гнучка методологія розроблення ПЗ орієнтована на використання ітеративного підходу, при якому програмний продукт створюється поступово, за декілька ітерацій, які включають реалізацію певного набору вимог. Результатом ітерації є проміжний варіант роботоздатного ПЗ.

Visual Studio і Team Foundation Server прдставляють рішення компанії Microsoft щодо керування життєвим циклом додатків. Керування життєвим циклом у Visual Studio базується на

принципах продуктивності, інтеграції та можливості розширення. У Visual Studio для архітектурного проектування використовуються інструменти візуального моделювання на основі мови UML. Team Foundation Server призначений для забезпечення сумісної роботи команд розробників ПЗ та має трирівневу сервіс-орієнтовану архітектуру.

Методологія Scrum представляє собою ітеративний процес розроблення ПЗ. Робочі елементи використовуються для відстежування, спостереження за станом ходу розроблення ПЗ та створення звітів. Організація команди в методології Scrum визначає ролі власника продукту, керівника та членів команди. Життєвий цикл проекту ПЗ включає формування та керування невиконаною роботою, планування та виконання спринту, розгортку ПЗ у зацікавлених осіб, одержання відгуків для покращення програмного продукту. При керуванні робочим процесом постійно відстежуються зв'язки та змінюються стани елементів невиконаної роботи по продукту.

Лекція №14
ОЦІНКА ТРУДОМІСТКОСТІ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План:

1. Методи оцінки трудомісткості
2. Методика оцінки трудомісткості розроблення ПЗ на основі функційних точок
3. Алгоритмічне моделювання трудомісткості розроблення програмного забезпечення

Рекомендована література:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.

4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

5. Петрухин В.А., Лаврищева Е.М. Методы и средства инженерии программного обеспечения // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/2190/237/lecture/3272

6. Вендров, А.М. Проектирование программного обеспечения экономических информационных систем / А.М. Вендров. - М.: Финансы и статистика, 2002

Вступ

Оцінка трудомісткості створення ПЗ є одним з найбільш важливих видів діяльності в процесі створення ПЗ. Моделі та методи оцінки трудомісткості використовуються для вирішення багатьох задач, серед яких можна виділити наступні:

- розроблення бюджету проекту (необхідна точність);

- аналіз ступеня рихику та вибір компромісного рішення (уточнюються масштаби проекту, можливість повторного використання, кількість розробників, використовуване обладнання);

- планування та управління проектом (одержані результати забезпечують розподіл та класифікацію витрат за компонентами, етапами та операціями);

- аналіз витрат на покращення якості ПЗ (це дозволяє оцінити витрати та прибуток від стратегії інвестування у вдосконалення технологій та можливості повторного використання).

За відсутності адекватної та достовірної оцінки неможливо забезпечити чітке планування та управління проектом. В цілому ситуація в даній галузі далеко не блиискуча. Недооцінка вартості, тривалості та ресурсів, необхідних для створення ПЗ, тягне за собою недостатню чисельність проектної команди, надмірно стислі терміни розроблення і, як результат, втрату довіри до розробників у випадку порушення графіку. З іншого боку, перестраховка та переоцінка можуть виявитись нічим не краще. Якщо для проекту виділено більше ресурсів, ніж реально необхідно, причому без належного контролю за їх використання, то про жодну економію не може йти мова. Такий проект виявиться більш дорогим, ніж повинен був бути при грамотній оцінці, і призведе до запізнення початку наступного проекту.

1. Методи оцінки трудомісткості

Методи оцінки трудомісткості:

1) *алгоритмічне моделювання* - метод засновано на аналізі статистичних даних про раніше виконані проекти, при цьому визначається залежність трудомісткості проекту від якогось кількісного показника ПЗ (як правило, це розмір коду). Проводиться оцінка цього показника для даного проекту, після чого за допомогою моделі прогнозуються майбутні витрати;

2) *експертні оцінки* - проводиться опитування декількох експертів з технології розроблення ПЗ, які знають область

використання створюваного програмного продукту. Кожен з них дає свою оцінку трудомісткості проекту. Потім всі оцінки порівнюються і обговорюються. Цей процес повторюється доти, доки не буде досягнуто угоди з кінцевого варіанту попередньої трудомісткості;

3) *оцінки за аналогією* - цей метод використовується в тому випадку, якщо в даній галузі застосування створюваного ПЗ вже реалізовано аналогічні проекти. Метод засновано на порівнянні планованого проекту з попередніми проектами, які мають подібні характеристики. Він використовує експертні дані або збережені дані про проект. Експерти обчсилюють високу, низьку та найбільш ймовірну оцінки трудомісткості, базуючись на різниці між новими та попередніми проектами. Оцінка може бути достатньо детальною в залежності від глибини аналогії. Слабкість моделі полягає в тому, що ступінь подібності нового проекту та попередніх, як правило, не надто велика. Найкращий варіант - це використання накопичених в організації історичних даних, які дозволяють співставити трудомісткість проекту із трудомісткістю попередніх проектів аналогічного розміру. Однак це можливо лише за умови ретельного документування результатів попередніх проектів, аналогічного характеру та розміру попереднього проекту та подібності ЖЦ, використовуваних методів та засобів розроблення, кваліфікації та досвіду проектного колективу;

4) *закон Паркінсона* - зусилля, витрачені на роботу, розподіляються рівномірно за виділеним на проект часом. Критерієм для оцінки витрат з проекту є людські ресурси, а не цільова оцінка самого ПЗ. Якщо проект, над яким працюють 5 чоловік, повинен бути завершений за 12 місяців, то витрати на його виконання рахуються як 60 людиномісцяів;

5) *оцінка з метою виграти контракт* - витрати на проект визначаються наявністю тих засобів, які є в замовника. Тому трудомісткість проекту залежить від бюджету замовника, а не від функційних характеристик створюваного продукту. Вимоги доводиться змінювати так, щоб не вийти за межі прийнятого бюджету.

Кожен метод оцінки має слабкі та сильні сторони. Для роботи над великими проектами необхідно застосовувати декілька методів оцінки для їх наступного порівняння. Якщо при цьому одержано суттєво різні результати, то інформації для одержання точної оцінки недостатньо. В цьому випадку необхідно скористатись додатковою інформацією, після чого повторити оцінку, і так доти, доки результати різних методів на стануть достатньо близькими. Описані методи оцінки застосовні, якщо документовані вимоги до мабйбутньої системи. В такому випадку є можливість визначити функційні характеристики розроблюваної системи. Однак в багатьох проектах оцінка витрат проводиться лише на основі попередніх вимог до системи. В цьому випадку особи, які приймають участь в оцінці вартості проекту, матимуть мінімум інформації для роботи.

Якісна оцінка трудомісткості розроблення ПЗ створюється і підтримується менеджером проекту та командами архітекторів, розробників і тестувальників, відповідальними за виконання роботи; сприймається всіма виконавцями як амбіційна, але виконувана; базується на детально описаній та обгрунтованій моделі оцінки; базується на даних за аналогічними проектами, які включають в себе аналогічні процеси, технології, середовище, вимоги щодо якості та кваліфікації робітників; детально описується таким чином, щоб всі ключові галузі ризику були очевидні, а ймовірність успіху оцінювалась об'єктивно.

Ідеальну оцінку можна одержати шляхом екстраполяції якісної оцінки, яка одержана на основі моделі трудомісткості та використовує досвід виконання множини аналогічних проектів, підготовлених тим самим колективом, який використовував ті ж зрілі процеси та інструментарій. Якісні оцінки можуть бути одержані на більш пізніх етапах ЖЦ проекту.

В *алгоритмічному моделюванні трудомісткості розроблення ПЗ* існує в основному два підходи до моделювання: теоретичні моделі та статистичні моделі. Більшість моделей для визначення трудомісткості розроблення ПЗ можуть бути зведені до функції *п'яти основних параметрів*:

- розміру кінцевого продукту – як правило, кількість рядків коду або кількість функційних точок, необхідних для реалізації даної функційності;

- особливостей процесу, використовуваного для одержання кінцевого продукту, зокрема, його здатність уникати непродуктивних видів діяльності (переробок, витрат на взаємодію);

- можливостей персоналу, який бере участь у розробленні ПЗ, особливо його професійного досвіду та знання предметної галузі проекту;

- середовища, яке складається з інструментів та методів, використовуваних для ефективного виконання розроблення ПЗ та автоматизації процесу;

- потрібної якості продукту, яка включає в себе його функційні можливості, продуктивність, надійність та адаптованість.

Співвідношення між цими параметрами та трудомісткістю можна визначити наступним чином:

Трудомісткість = (Персонал) • (Середовище) • (Якість) • (Розмір^{Процес}).

Найбільш впливовий фактор оцінки трудомісткості в цих моделях – розмір програмного продукту.

Процедура оцінки трудомісткості розроблення ПЗ складається з наступних дій:

1) оцінка розміру розроблюваного продукту;

2) оцінка трудомісткості в людино-місяцях або людино-годинах;

3) оцінка тривалості проекту в календарних місяцях;

4) оцінка вартості проекту.

Оцінка розміру продукту базується на знанні вимог до системи. Для такої оцінки існує два основних способи:

1) за аналогією – якщо раніше доводилось мати справу з подібним проектом, і його оцінки відомі, то можна, відштовхуючись від нх, приблизно оцінити свій новий проект;

2) шляхом підрахунку розміру за певними алгоритмами на основі початкових даних – вимог до системи.

Проблеми оцінки розміру ПЗ:

- проблема може бути недостатньо добре зрозумілою розробникам та замовникам через упущення чи спотворення деяких фактів;
- нестача чи повна відсутність історичних даних не дозволяє створити базу для оцінок в майбутньому;
- проектуючи організація не володіє стандартами, за допомогою яких можна виконувати процес оцінювання або, в разі наявності стандартів, організація їх просто не дотримується; в результаті спостерігається нестача сумісності при виконанні процесу оцінювання;
- менеджери проектів вважають, що непогано було б фікусвати вимоги на початку проекту, замовники ж вважають, що не варто витрачати час на розроблення специфікації вимог;
- помилки, як правило, приховуються, замість того, щоб оцінюватисьі та відображатись, в результаті чого створюється хибне враження про фактичну продуктивність;
- можливості оцінювання істотно залежать від суб'єктів, залучених до процесу оцінювання;
- менеджери, аналітики, розробники, тестувальники та ті, хто впроваджує проект, можуть мати різні уявлення про процеси оцінювання та про можливості вдосконалення продукту.

Точність оцінки вартості та розміру ПЗ в залежності від стадії проекту визначається наступним графіком (рис.14.1).

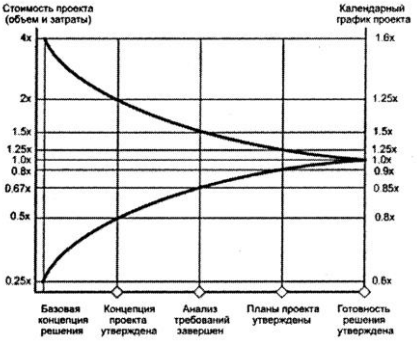


Рис.14.1 - Точність оцінки вартості та розміру ПЗ в залежності від стадії проекту

Основними одиницями вимірювання розміру ПЗ є:

- кількість рядків коду (LOC — Lines of Code);
- функційні точки (FP — Function Points).

Кількість рядків коду – найвідоміша та донедавна найпоширеніша одиниця вимірювання. Однак при її використанні виникає ряд питань: яким чином можна визначити кількість рядків коду до того, як вони фактично будуть написані або спроектовані?; як показник кількості рядків коду може відбивати величину трудовитрат, якщо не буед враховуватись складність продукту, здібності програміста та можливості застосовуваної мови програмування?; яким чином різниця в кількості рядків коду може бути трансформована у обсяг еквівалентної роботи? Ці та інші питання призвели до того, що рядки коду як одиниці вимірювання одержали «погану репутацію», хоча, як і раніше, залишаються найбільш широко вживаними. Взаємозв'язок між LOC та зусиллями, що витрачаються, не є лінійною.

Незважаючи на появу нових мов програмування, середня продуктивність роботи програмістів за 20 років залишилась незмінною і складає близько 3000 рядків коду на одного програміста на рік. Це говорить про те, що зменшення часу, який витрачається на цикл розроблення, не може бути досягнуте за рахунок значного підвищення продуктивності праці програміста. Причому це не залежить від вдосконалень мов програмування, зусиль з боку менеджерів або понаднормових робіт. Насправді найважливіше значення має набір функційних точок та якість ПЗ, а не кількість рядків коду.

Переваги використання LOC в якості одиниць вимірювання:

- широке поширення та легка адаптованість;
- можливість співставлення методів вимірювання розмірів та продуктивності в різних групах розробників;
- безпосередній зв'язок з кінцевим продуктом;
- легка оцінка до завершення проекту;
- оцінка розмірів ПЗ на основі точки зору розробника – фізична оцінка створеного продукту (кількість написаних рядків коду).

Недоліки застосування LOC:

- LOC важкі для застосування при оцінювання розміру ПЗ на ранніх етапах розроблення;
- рядки сирцевого коду можуть розрізнятись в залежності від типів мов програмування, методів проектування, стилю та здібностей програміста;
- застосування методів оцінки за допомогою підрахунку кількості рядків коду не регламентується промисловими стандартами;
- розроблення ПЗ може бути пов'язане із великими витратами, які прямо не залежать від розмірів програмного коду – «фіксованими витратами» такими, як складання специфікацій вимог та документів користувача, не включених у прямі витрати на кодування;
- програмісти можуть бути незаслужено премійовані за досягнення високих показників LOC у випадку, якщо керівництво помилково вважатиме це ознакою високої продуктивності, але при цьому відсутнім буде ретельно розроблений проект; сирцевий код не є самоціллю при створенні продукту – головну роль мають функційні властивості та показники продуктивності;
- при підрахунку кількості LOC слід розрізняти автоматично та вручну створений код – ця задача є більш складною, ніж простий підрахунок, який може бути виконаний на основі лістингу, згенерованого компілятором, або за допомогою утиліти, яка виконує підрахунок рядків коду;
- показники LOC не можуть застосовуватись при здійсненні нормалізації у випадку, якщо застосовувані платформи розроблення або мови є різними;
- єдиний спосіб врахування за допомогою LOC по відношенню до розроблюваного ПЗ полягає у використанні методу аналогії на основі порівняння функційних властивостей в подібних програмних продуктах, або у використанні думок експертів (однак ці методи не є точними);
- генератори коду часто продукують надмірний обсяг коду, в результаті чого спотворюються показники LOC.

Результатом цих міркувань є усвідомлення необхідності іншої одиниці вимірювання, в якості якої і стали виступати функційні точки.

2. Методика оцінки трудомісткості розроблення ПЗ на основі функційних точок

Визначення кількості функційних точок є методом кількісної оцінки ПЗ, який застосовується для вимірювання функційних характеристик процесів його розроблення та супроводу незалежно від технології, використаної для його реалізації. Підрахунок функційних точок, крім засобу для об'єктивної оцінки ресурсів, необхідних для розроблення та супроводу ПЗ, застосовується також в якості засобу для визначення складності продукту, що купується, з метою прийняття рішення про купівлю або власне розроблення. Метод розроблено на основі досвіду реалізації багатьох проектів створення ПЗ та підтримується Міжнародною *організацією IFPUG (International Function Point User Group)*. Розглядуваний далі скорочений варіант методики оцінки трудомісткості розроблення ПЗ заснований на матеріалах IFPUG та компанії SPR (Software Productivity Research), яка є одним з лідерів в галузі методів та засобів оцінки характеристик ПЗ.

Згідно даної методики, трудомісткість обраховується на основі функційності розроблюваної системи, яка, в свою чергу, визначається на основі виявлення функцій них типів – логічних груп взаємопов'язаних даних, які використовуються та підтримуються додатком, а також елементарних процесів, пов'язаних із введенням та виведенням інформації (рис.14.2).

Порядок розрахунку трудомісткості розроблення ПЗ:

- визначення кількості та складності функційних типів додатку;

- визначення кількості пов'язаних із кожним функційним типом елементарних даних (DET), елементарних записів (RET) та файлів типу посилань (FTR);

- визначення складності (в залежності від кількості DET, RET і FTR);

- підрахунок кількості функційних точок додатку;

- підрахунок кількості функційних точок з врахуванням загальних характеристик системи (рис.14.3);

- оцінка трудомісткості розроблення (з використанням різних статистичних даних).

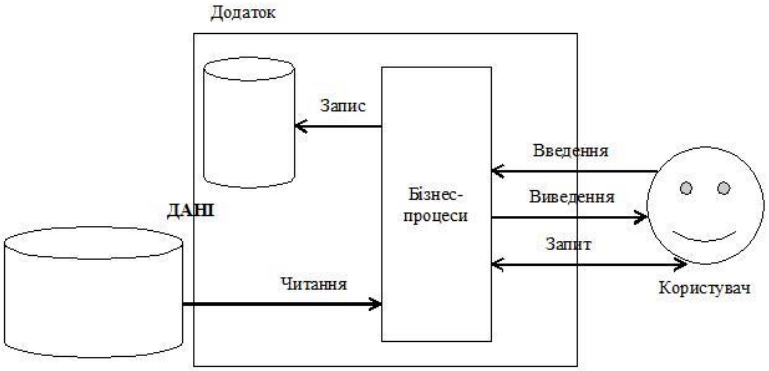


Рис.14.2 - Виявлення функційних типів

Рис.14.3 - Визначення кількості функційних точок

До складу функційних типів входять наступні елементи додатків розроблюваної системи:

1) *внутрішній логічний файл (internal logical file, ILF)* – ідентифікована сукупність логічно взаємопов'язаних записів даних, яка підтримується всередині додатку завдяки елементарному процесу;

2) *зовнішній інтерфейсний файл (external interface file, EIF)* – ідентифікована сукупність логічно взаємопов'язаних записів даних, які передаються іншому додатку або одержуються від нього та підтримуються поза даним додатком;

3) *вхідний елемент додатку (external input, EI)* – елементарний процес, пов'язаний з обробленням вхідної інформації

додатку – вхідного документа або екранної форми. Оброблювані дані можуть відповідати одному або більше ILF;

4) *вихідний елемент додатку (external output, EO)* – елементарний процес, пов'язаний із обробленням вихідної інформації додатку – вихідного звіту, документу, екранної форми;

5) *зовнішній запит (external query, EQ)* – елементарний процес, який складається з комбінації «запит/відповідь», не пов'язаний з обчисленням похідних даних або оновленням ILF (бази даних).

Кількість функційних типів за даними (внутрішніх логічних файлів та зовнішніх інтерфейсних файлів) визначається на основі діаграм «сутність-зв'язок» (для структурного підходу) та діаграм класів (для об'єктно-орієнтованого підходу). В останньому випадку в розрахунках приймають участь лише стійкі класи або класи-сутності. Стійкий клас відповідає ILF (якщо його об'єкти обов'язково створюються всередині самого додатку) або EIF (якщо його об'єкти не створюються всередині самого додатку, а одержуються в результаті запитів до баз даних).

Для кожного виявленого функційного типу (ILF и EIF) визначається його *складність* (низька, середня або висока). Вона залежить від кількості пов'язаних з цим функційним типом елементарних даних (data element types, DET) та елементарних записів (record element types, RET), які, в свою чергу, визначаються наступним чином:

- *DET* – унікальний ідентифікований нерекурсивний елемент даних (включаючи зовнішні ключі), який входить до ILF або EIF;

- *RET* – ідентифікована підгрупа елементів даних, яка входить до ILF. На діаграмах «сутність-зв'язок» така підгрупа представляється у вигляді сутності-підтипу в зв'язку «супертип-підтип».

Один DET відповідає окремому атрибуту або зв'язку класу. Кількість DET не залежить від кількості об'єктів класу або кількості зв'язаних об'єктів. Якщо даний клас пов'язаний із деяким іншим класом, який має явно заданий ідентифікатор, що складається більш ніж з одного трибуту, то для кожного такого

атрибуту визначається один окремий DET (а не один DET на весь зв'язок). Похідні атрибути можуть ігноруватись. Повторювані атрибути однакового формату розглядаються як один DET.

Одна RET на діаграмі стійких класів відповідає або абстрактному класу в зв'язку узагальнення, або класу – «частині цілого» в композиції, або класу з рекурсивним зв'язком «батько-нащадок» (агрегацією).

Залежність складності функційних типіввід кількості DET та RET визначається наступною таблицею – таблиця 14.1.

Таблиця 14.1 - Складність ILF та EIF

Кількість RET	Кількість DET		
	1-19	20-50	51 +
1	Низька	Низька	Середня
2-5	Низька	Середня	Висока
6 +	Середня	Висока	Висока

Кількість транзакційних функційних типів (вхідних елементів додатку, вихідних елементів додатку та зовнішніх запитів) визначається на основі виявлення вхідних та вихідних документів, екранних форм, звітів, а також за діаграмами класів (в розрахунку беруть участь граничні класи).

Далі для кожного виявленого функційного типу (EI, EO або EQ) визначається його складність (низька, середня або висока). Вона залежить від кількості пов'язаних з цим функційним типом DET, RET і файлів типу посилань (file type referenced, FTR) - ILF або EIF, які читаються або модифікуються функційним типом.

Залежність складності функцій них типів від кількості DET, RET або FTR визначається за таблицями 14.2 і 14.3.

Таблиця 14.2 - Складність EI

Кількість FTR	Кількість DET		
	1-4	5-15	16 +
0-1	Низька	Низька	Середня
2	Низька	Середня	Висока
3 +	Середня	Висока	Висока

Таблиця 14.3 - Складність ЕО

Кількість FTR	Кількість DET		
	1-5	6-19	20 +
0-1	Низька	Низька	Середня
2-3	Низька	Середня	Висока
4 +	Середня	Висока	Висока

Складність EQ визначається як максимальна із складностей EI та EO, пов'язаних із даним запитом.

Для кожного функційного типу підраховується кількість функційних точок FP, які входять до його складу, - умовних елементарних одиниць. Цей підрахунок виконується відповідно до таблиці 14.4.

Таблиця 14.4 - Залежність кількості FP від складності функційного типу

Функційний тип	Складність		
	Низька	Середня	Висока
ILF	7	10	15
EIF	5	7	10
EI	3	4	6
EO	4	5	7
EQ	3	4	6

В результаті додавання кількості FP за всіма функційними типами одержується загальна кількість FP (UFP, Unadjusted Function Points) без врахування коефіцієнту поправки.

Значення коефіцієнту поправки (*VAF*, Value Adjustment Factor) визначається набором з 14 загальних характеристик системи (GSC, General System Characteristics) і обчислюється за наступною формулою: *VAF = (0,65 + (sum GSC * 0,01))*. Значення GSC варіюються в діапазоні від 0 до 5 і обчислюються за спеціальними даними, наведеними у класичній фаховій літературі.

Після визначення всіх значень GSC і обчислення коефіцієнту поправки VAF обчислюється підсумкова оцінка кількості функційних точок (Adjusted Function Points, AFP): *AFP = UFP * VAF*.

У таблиці 14.5 наведено кількість рядків коду на одну функційну точку в залежності від використовуваної мови програмування.

Таблиця 14.5 – Кількість рядків коду (SLOC) на одну функційну точку

Мова (засіб)	Кількість SLOC на FP
Access	38
ANSI SQL	13
C++	53
Clarion	58
Data base default	40
Delphi 5	18
Excel 5	6
FoxPro 2.5	34
Oracle Developer	23
PowerBuilder	16
Smalltalk	21
Visual Basic 6	24
Visual C++	34
HTML 4	14
Java 2	46

Добуток AFP та кількості SLOC (на FP) дають кількість SLOC у додатку.

Далі для оцінки трудомісткості та часу розроблення може використовуватись один з варіантів відомої моделі оцінки трудомісткості розроблення ПЗ під назвою COCOMO та її сучасної версії COCOMO II.

В таблиці 14.6 наведено усереднені статистичні дані розміру ПЗ за деякими видами додатків.

Таблиця 14.6 - Розмір програмного забезпечення в FP і LOC

Тип ПЗ	Розмір, FP	Розмір, LOC	Кількість LOC на одну FP
Текстові процесори	3500	437500	125

Електронні таблиці	3500	437500	125
Клієнт-серверні додатки	7500	675000	90
ПЗ баз даних	7500	937500	125
Виробничі додатки	7500	937500	125
Великі бізнес-додатки	10000	1050000	105
Великі корпоративні додатки	25000	2625000	105
Великі додатки в держустановах	50000	5250000	105
Операційні системи	75000	11250000	150
Системи масштабу підприємства	150000	18750000	125
Великі оборонні системи	250000	25000000	100

На реалізацію проектів розміром в *1 функційну точку* потрібен один день, і вонаи практично завжди завершуються успішно. Це, як правило, невеликі утиліти для тимчасових потреб. Обсяг в *10 функційних точок* – це типовий обсяг невеликих додатків та доповнень, які вносяться в готові системи. Такі проекти потребують до 1 місяця робіт і теж завжди успішні. Обсяг в *100 функційних точок* близький до меж можливостей програміста-одинака. Проект доводиться до завершення за 6 місяців у 85% випадків. Обсяг проекту в *1000 функційних точок* характерний для більшості сьогоднішніх комерційних невеликих клієнт-серверних додатків. Помітну частку загального обсягу роботи займає документація. Для розроблення проекту необхідна група приблизно з 10 програмістів, проектувальників, фахівців з якості та близько 1 року роботи. Провальними є 15% подібних колективних проектів та 65% спроб програмістів-одинкаів. У 20% випадків не вдається вкластись в термін. Перевитрати коштів відбуваються в 25% проектів. Для проекту обсягом в *10000 функційних точок* потрібні близько 100 розробників. Гостро стоять питання організації колективної роботи. Роботи тривають від 1,5 до 5 років, при цьому заплановані терміни частіше всього не витримуються. Гарну якість

системи неможливо забезпечити без використання формальної технології тестування. До 50% проектів завершуються невдачею, а реалізація таких проектів одним програмістом взагалі неможлива. Обсяг в *100000 функційних точок* – поки що межа для більшості сьогоднішніх проектів. Це йобсяг сучасних операційних систем та найбільших військових систем. На їх створення йде 5-8 років. Над проектом працюють сотні розробників іноді з різних країн, і ефективно координувати їх роботу не вдається. В завершених системах багато помилок. До 65% проектів завершуються провалом. В усіх успішних проектах використовуються системи керування конфігурацією.

Таблиця 14.7 ілюструє різницю в розподілі часових витрат за стадіями життєвого циклу у випадку великого та маленького проекту. Маленький проект (наприклад, додаток Windows обсягом 50000 рядків сирцевого коду мовою Visual Basic, створюваний колективом з 5 чоловік) може вимагати 1 місяць на початкову стадію, 2 місяці – на проектування, 5 місяців – на розроблення і 2 місяці – на введення в дію. Великий, складний проект (наприклад, бортова програма для літального апарату обсягом 300000 рядків сирцевого коду, створювана колективом з 40 чоловік) може вимагати 8 місяців на початкову стадію, 14 місяців – на проектування, 20 місяців – на розроблення і 8 місяців – на введення в дію. Порівняння часток життєвого циклу, які доводяться на кожну стадію, дозволяє виявити очевидні різниці.

Таблиця 14.7 - Розподіл часових витрат за стадіями для маленьких та великих проектів

Масштаб проекту	Початкова стадія, %	Проекту-вання, %	Розробле-ння, %	Введення в дію, %
Малень-кий комерцій-ний проект	10	20	50	20
Великий складний проект	15	30	40	15

Час розроблення може буи змінений з врахуванням статистичних даниз, накопичених в реальних проектах і відображених у таблиці 14.8, де співставлені планований та реальний терміни виконання проекту в залежності від його розміру, вираженого в кількості функційних точок.

Таблиця 14.8 – Статистичні дані

Розмір проекту	<100 FP	100-1000 FP	1000-10000 FP	>10000 FP
Планований термін (міс.)	6	12	18	24
Реальний термін (міс.)	8	16	24	36
Відставання	2	4	6	12

Частково відставання пояснюється неточною оцінкою, частково – зростанням кількості вимог до системи після того, як виконано початкову оцінку.

3. Алгоритмічне моделювання трудомісткості розроблення програмного забезпечення

Математичне моделювання трудомісткості розроблення ПЗ засноване на співставленні експериментальних даних з формою існуючої математичної функції. На початку 60-х років XX століття Пітер Норден прийшов до висновку, що в проектах з дослідження та розроблення може застосовуватись добре прогнозований розподіл трудових ресурсів, який базується на розподілі ймовірності, що називається кривою Релея (Rayleigh distribution). Пізніше, в 70-х роках XX століття Лоуренс Патнем застосував результати Нордена до розроблення ПЗ. Використовуючи статистичний аналіз проектів, Патнем виявив, що взаємозв'язок між трьома основними параметрами проекту (розміром, часом та трудомісткістю) нагадує функцію Нордена-Релея (рис.4), яка відбиває розподіл трудових ресурсів проекту в залежності від часу.

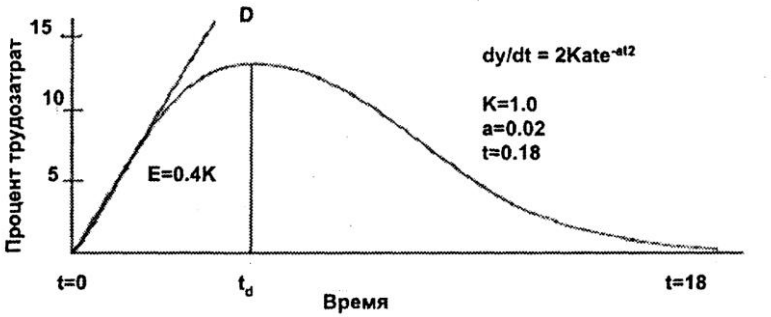


Рис.4 - Функція Релея

Функція Релея моделюється диференційним рівнянням:

$$\frac{dy}{dt} = 2 \cdot K \cdot a \cdot t \cdot \exp(-a \cdot t^2),$$

де *dy/dt* — швидкість зростання персоналу проекту, *t* — час, який пройшов від початку проекту до вилучення продукту з експлуатації, *K* — область під кривою – представляє повну трудомісткість протягом всього життєвого циклу (включаючи супровід), виражену в людино-роках, *a* — константан, яка визначає форму кривої (фактор прискорення) і обчислюється за формулою:

$$a = \frac{1}{2} \cdot t_d^2,$$

де t_d – час розроблення.

Після прийняття ряду припущень, *Патнем* одержав наступне рівняння:

$$E = 0.4 \cdot [\frac{S}{C}]^3 \cdot \frac{1}{(t_d)^4},$$

де *E* — трудомісткість розроблення ПЗ, *S* — розмір ПЗ в LOC, t_d — планований термін розроблення, *C* — технологічний фактор, який враховує різні апаратні обмеження, досвід персоналу та характеристики середовища програмування. Він визначається на основі хронологічних даних за минулими проектами і, згідно рекомендацій Патнема, визначається для різних типів проектів наступним чином: проект, впроваджений в стислі терміни без

ретельного пропрацювання – 1500; проект, виконаний відповідно до чіткого плану, – 5000; проект, який передбачає оптимальну організацію та підтримку, - 10000. Оптимальний термін розроблення визначається як: $t_d = 2.4 \cdot E^{\frac{1}{3}}$

Статистичні моделі використовують накопичені хронологічні дані, щоб одержати значення для коефіцієнтів моделі. Для визначення співвідношень між параметрами моделі та трудомісткістю розроблення ПЗ використовується регресійний аналіз. Існує 2 форми статистичних моделей: лінійні та нелінійні.

Лінійні статистичні моделі мають наступний вигляд:

$$Трудомісткість = b_0 + \sum_{i=1}^{n} b_i \cdot x_i ,$$

де x_i - фактори, які впливають на трудомісткість, b_i — коефіцієнти моделі. Лінійні моделі працюють не надто добре, оскільки практика показує, що співвідношення між трудомісткістю та розміром ПЗ нелінійні. Із зростанням розміру ПЗ виникає експоненційний від'ємний ефект масштабу.

Нелінійні статистичні моделі мають наступний вигляд:

$$Трудомісткість = A \cdot (Розмір\ \ ПЗ)^b ,$$

де A — комбінація факторів, які впливають на трудомісткість, b — експоненційний коефіцієнт масштабу.

Статистичні моделі прості для розуміння, але мають наступний недолік: результати справедливі в основному лише для конкретної ситуації. Інший недолік – при збільшенні кількості вхідних параметрів кількість даних, необхідних для калібрування моделі, також зростає.

Модель COCOMO (Constructive COst Model — конструктивна модель вартості), розроблена Барі Боемом, є однією з найвідоміших моделей оцінки трудомісткості розроблення ПЗ. Початкова модель COCOMO базувалась на 56 виконаних проектах, а її різні варіанти відбивали різницю між процесами в різних галузях ПЗ. Ця модель включає ряд припущень: 1) сирцевий код продукту включає в себе всі (крім коментарів) рядки коду; 2)

початок циклу розроблення співпадає з початком розроблення продукту, завершення співпадає із завершенням приймального тестування, яке завершує стадію інтеграції та тестування (робота і час, які витрачаються на аналіз вимог, оцінюються окремо як додатковий процент від оцінки розробки в цілому); 3) види діяльності включають в себе лише безпосередньо спрямовані на виконання проекту роботи, в них не входять звичайні допоміжні види діяльності – такі, як адміністративна підтримка, технічне забезпечення, капітальне обладнання; 4) людино-місяць складається зі 152 годин; 5) проект керується належним чином, в ньому використовуються стабільні вимоги.

Проект *COCOMO II* (сучасний варіант моделі COCOMO) був виконаний в Центрі з розроблення ПЗ Південно-Каліфорнійського університету (USC Centre for Software Engineering) і переслідував наступні цілі: 1) розробити модель для оцінювання трудомісткості та термінів створення ПЗ для ітераційної моделі життєвого циклу ПЗ; 2) створити базу даних трудомісткостей ПЗ; 3) розробити інструментальну підтримку для вдосконалення моделі; 4) створити кількісну аналітичну схему для оцінювання технологій створення ПЗ та їх економічного ефекту.

Рівняння COCOMO II для оцінки номінальних значень трудомісткості та часу мають наступний вигляд:

1) трудомісткість (в людино-місяцях):

$$PM_{NS} = A \cdot Size^{E} \cdot \prod_{i=1}^{n} EM_i \ ,$$

де $E = B + 0.01 \cdot \sum_{j=1}^{5} SF_j$;

2) календарний час:

$$TDEV_{NS} = C \cdot (PM_{NS})^{F} \ ,$$

де $F = D + 0.2 \cdot 0.01 \cdot \sum_{j=1}^{5} SF_j = D + 0.2 \cdot (E - B)$, EM_i - мультиплікативні коефіцієнти трудомісткості, SF_j - експоненційні

коефіцієнти масштабу, *Size* - розмір ПЗ, виражений в тисячах рядків сирцевого коду або кількості функційних точок без врахування коефіцієнтів поправки; *A, B, C, D* - калібрувальні змінні моделі COCOMO II (за версією 2000 року мають наступні значення: $A = 2.94, B = 0.91, C = 3.67, D = 0.28$). Коефіцієнти EM_i відбивають сумісний вплив багатьох параметрів. Вони дозволяють характеризувати та нормувати середовище розроблення за параметрами, які містяться в базі даних проектів моделі COCOMO II (наразі більше 160 проектів). Кожен коефіцієнт в залежності від встановленого значення вносить свій внесок у вигляді множника з певним діапазоном значент. Результат врахування цих 17 коефіцієнтів використовується при обчисленні в рівнянні трудомісткості. Склад коефіцієнтів наведено у таблиці 9.

Таблиця 9 - Мультиплікативні коефіцієнти трудомісткості

Ідентифікатор	Опис коефіцієнту	Діапазон значень
RELY	Потрібна надійність	0.82-1.26
DATA	Розмір бази даних	0.90-1.28
CPLX	Складність продукту	0.73-1.74
RUSE	Потрібний рівень повторного використання	0.95-1.24
DOCU	Відповідність документації вимогам життєвого циклу	0.81-1.23
TIME	Обмеження часу виконання	1.00-1.63
STOR	Обмеження за обсягом основної пам’яті	1.00-1.46
PVOL	Мніливість платформи	0.87-1.30
ACAP	Здібності аналітика	1.42-0.71
PCAP	Здібності програміста	1.34-0.76
APEX	Знання додатків	1.22-0.81
PLEX	Знання платформи	1.19-0.85
PCON	Послідовність персоналу	1.29-0.81

LTEX	Знання мови / інструментальних засобів	1.20-0.84
TOOL	Використання інструментальних засобів	1.17-0.78
SCED	Потрібні терміни розроблення	1.43-1.00
SITE	Розсередженість колективу розробників	1.22-0.80

Показник експоненти процесу *E* може змінюватись в діапазоні від 0.91 до 1.23 і визначається як сполучення наступних параметрів: наявність прецедентів в додатку (ступінь досвідченості організації-розробника в даній галузі); гнучкість процесу (ступінь суворості контракту, порядок його виконання, присутня контракту свобода внесення змін, види діяльності протягом всього ЖЦ і взаємодія між зацікавленими сторонами); дозвіл ризиків, присутніх архітектурі (ступінь технічної здійсненності, продемонстрованої до переходу до повномасштабного впровадження); єдність колективу (ступінь співробітництва і того, наскільки всі зацікавлені сторони розділяють загальну концепцію); зрілість процесу (рівень зрілості організації-розробника, який визначається відповідно до моделі CMM).

В таблиці 10 наведено можливі значення експоненційних коефіцієнтів масштабу, які складають в сумі показник *E*. Сумарний вплив цих коефіцієнтів може виявитись вельми істотним.

Таблиця 10 – Експоненційні коефіцієнти масштабу моделі COCOMO II

Параметр	Характеристика	Значення
PREC – наявність прецедентів	Повна відсутність прецедентів, повністю непередбачуваний проект	6.20
	Майже повна відсутність прецедентів, в значному ступені непередбачуваний проект	4.96
	Наявність деякої кількості прецедентів	3.72

	Загальне знайомство з проектом	2.48
	Значне знайомство з проектом	1.24
	Повне знайомство з проектом	0.00
FLEX – гнучкість процесу розроблення	Точний, суворий процес розроблення	5.07
	Випадкові послаблення у процесі	4.05
	Деякі послаблення в процесі	3.04
	Здебільшого узгоджений процес	2.03
	Деяка узгодженість процесу	1.01
	Замовник визначив лише загальні цілі	0.00
RESL – дозвіл ризиків в архітектурі	Малий (20%)	7.07
	Деякий (40%)	5.65
	Частий (60%)	4.24
	В цілому (75%)	2.83
	Майже повний (90%)	1.41
	Повний (100%)	0.00
TEAM – єдність команди	Сильно ускладнена взаємодія	5.48
	Дещо ускладнена взаємодія	4.38
	Деяка узгодженість	3.29
	Підвищена узгодженість	2.19
	Висока узгодженість	1.10
	Взаємодія як в єдиному цілому	0.00
PMAT – зрілість процесів	Рівень 1 CMM	7.80
	Рівень 1+ CMM	6.24
	Рівень 2 CMM	4.68
	Рівень 3 CMM	3.12
	Рівень 4 CMM	1.56
	Рівень 5 CMM	0.00

Висновки

Оцінка трудомісткості створення ПЗ є одним з найбільш важливих видів діяльності в процесі створення ПЗ. Моделі і методи оцінки трудомісткості необхідні для розроблення бюджету проекта, аналізу ступеня ризиків та вибору компромісного рішення, планування та управління проектом, аналізу витрат на покращення якості ПЗ. Більшість моделей для визначення трудомісткості розроблення ПЗ можуть бути зведені до функції п’яти основних параметрів – розміру кінцевого продукту, особливостей процесу, можливостей персоналу, середовища та потрібної якості продукту.

Лекції №15-17
МЕТОДИ ТА ЗАСОБИ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ЕТАПІ ПРОЕКТУВАННЯ

План:

1. Поняття та моделі якості ПЗ. Методи оцінки значень показників якості та керування якістю програмного забезпечення
2. Надійність ПЗ. Моделі надійності. Розрахунок надійності за відомими моделями
3. Складність ПЗ. Моделі та методи складності

4. Основні принципи метричного аналізу
5. Вибір та розрахунок метрик, придатних до використання на етапі проектування ПЗ
6. Дослідження результатів метричного аналізу

7. Поняття та методи статичного аналізу програмного коду
8. Засоби статичного аналізу ПЗ
9. Дослідження результатів статичного аналізу
10. Основи тестування ПЗ

Рекомендована література:

1. Мищенко В.О., Поморова О.В., Говорущенко Т.А. CASE-оценка критических программных систем. В 3-х томах. Том 1. Качество / Под ред. Харченко В.С. - Харьков: Нац.аэрокосмический университет "ХАИ", 2012. - 201 с.

2. Свідоцтво №48495 про реєстрацію авторського права на твір "Нейромережний спосіб оцінювання результатів проектування і прогнозування характеристик складності та якості програмного забезпечення" / авт. Говорущенко Т.О.

3. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

4. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

5. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

6. Петрухин В.А., Лаврищева Е.М. Методы и средства инженерии программного обеспечения // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/2190/237/lecture/3272

7. М.Моисеев. Методы анализа и обеспечения качества ПО // [Електронний ресурс] - Режим доступу: http://kspt.ftk.spbstu.ru/media/files/2012/course/quality/lec2.pdf

8. Скляр В.В. Оценка качества и экспертиза программного обеспечения. Лекционный материал. - Харьков: НАУ "ХАИ", 2008. - 204 с.

9. Липаев В.В. Программная инженерия. Методологические основы. - М.: ТЕИС, 2006. - 608 с.

10. Ковалевская Е.В. Материалы к курсу "Метрология, качество и сертификация ПО" - М.: Московский государственный университет экономики, статистики и информатики, 2002. - 38 с. // [Електронний ресурс] - Режим доступу: http://bookinist.net/books/bookid-333504.html

11. ISO/IEC 25010:2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models / ISO/IEC, 2011

12. С.Макконнелл. Совершенный код. Мастер-класс - М.: Издательство "Русская редакция", 2013 - 896 с.

Вступ

Наразі розроблення ПЗ — найбільша галузь світової економіки, в якій зайнято близько 3 млн. фахівців (програмістів, розробників ПЗ та ін.), а інші 5 млрд. осіб безпосередньо залежать від їхньої успішної діяльності.

За час розроблення програмного забезпечення вирішувані задачі, рівень їх складності та форми представлення отриманих результатів кардинально змінилися. Але й дотепер розробка якісних програмних продуктів не стала нормою.

Криза у галузі забезпечення якості ПЗ існує - великі проекти виконуються з відставанням від графіка або з

перевищенням кошторису витрат, розроблені продукти не мають необхідних функціональних можливостей, продуктивність ПЗ низька, якість ПЗ не влаштовує споживачів. За наближеними оцінками витрати на розроблення ПЗ складають близько 275 мільярдів доларів, але лише 72% програмних проектів досягають етапу впровадження і всього 26% програмних проектів завершуються успіхом, тобто лише 71,5 мільярд доларів витрачається на успішні проекти, а решта 200 мільярдів витрачаються на провальні або незавершені проекти (рис.15.1).

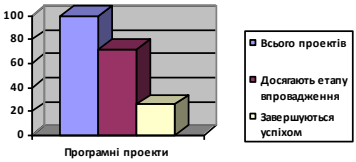


Рис.15.1 – Статистика щодо впровадження програмних проектів згідно оцінок Standish Group International

За статистикою, 18% програмних проектів ніколи не завершуються; 53% проектів по розробленню ПЗ завершуються з перевитратами на 56% і перевищенням термінів на 84%; і лише 29% проектів вкладаються в терміни та бюджет (рис.15.2).

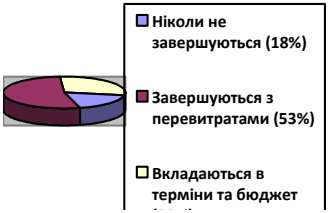


Рис.15.2 – Статистика щодо успішності програмних проектів

За даними, опублікованими в ISO/IEC TR 15504 Information Technology – Software Process Assessment (Part 1–9), щорічно помилки в ПЗ США коштують 60 млрд.доларів.

Розроблено чимало методів та засобів, для розроблення технологій та стандартів забезпечення якості програмних

комплексів залучені кращі фахівці, але якість ПЗ, як і раніше, залежить від знань та досвіду розробників.

Програмні проекти часто зазнають невдач через неадекватне формулювання вимог, невдале проектування або неефективне планування, невірне розуміння або недостатній аналіз специфікації та проекту, тобто через помилки на ранніх етапах життєвого циклу ПЗ. Дослідження 50 проектів, на які витрачено понад 400 людинороків і які включили майже 3000000 рядків коду, проведені у Лабораторії проектування ПЗ NASA, показали, що підвищена увага до раннього контролю якості дозволяє істотно знизити рівень помилок, але не підвищує загальних витрат на розроблення. Отже, забезпечення можливості раннього виявлення помилок та оцінювання якості проекту і прогнозування рівня якості розроблюваного за проектом ПЗ на етапі проектування дали б можливість зменшити витрати на розроблення ПЗ, а то й уникнути ряду катастроф та інцидентів, причини яких були внесені на етапах формулювання вимог та проектування.

1. Поняття та моделі якості ПЗ. Методи оцінки значень показників якості та керування якістю програмного забезпечення

Взагалі, *основною характеристикою ПЗ з точки зору розробника є його складність, а з точки зору користувачів – його надійність та якість*.

Якість ПЗ - це характеристика ПЗ, яка відображає ступінь його відповідності вимогам. При цьому вимоги можуть трактуватись досить широко, що породжує цілий ряд незалежних визначень поняття якості. Згідно визначення ISO 9001:1994, якість - це ступінь відповідності присутніх характеристик вимогам. Згідно ISO 8402:1994, якість - це повнота властивостей і характеристик продукту, процесу або послуги, які забезпечують здатність задовольняти оголошеним або передбачуваним потребам. Згідно IEEE Std. 1061-1998, якість ПЗ - це ступінь, в якій воно володіє потрібною комбінацією властивостей. Згідно ГОСТ 28806-90, якість програмного засобу - це сукупність властивостей програмного засобу, які обумовлюють його придатність

задовільняти задані або передбачувані потреби у відповідності до його призначення. Стандарт ГОСТ Р ИСО/МЭК 9126-93 дає наступне визначення якості ПЗ - весь обсяг ознак та характеристик програмної продукції, який належить до її здатності задовольняти встановлені або передбачувані потреби.

У відповідності до стандартів «Сборник действующих международных стандартов ИСО серии 9000», *забезпечення якості* - це сукупність планованих та систематичних заходів, необхідних для впевненості в тому, що продукція або процеси задовольняють певним вимогам.

Критерій оцінки якості програмного засобу - сукупність прийнятих у встановленому порядку правил і умов, за допомогою яких встановлюється прийнятність в цілому якості програмного засобу [ГОСТ 28806-90]. Згідно ГОСТ Р ИСО/МЭК 9126-93, критерій оцінки якості ПЗ - набір визначених та задокументованих правил і умов, які використовуються для висновку про прийнятність загальної якості конкретної програмної продукції.

Характеристика якості програмного засобу - набір властивостей програмного засобу, за допомогою яких описується та оцінюється його якість [ГОСТ 28806-90]. Згідно ГОСТ Р ИСО/МЭК 9126-93, характеристика якості ПЗ - набір властивостей (атрибутів) програмної продукції, за якими її якість описується і оцінюється. *Показник якості програмного засобу* - характеристика якості програмного засобу, яка має кількісне значення [ГОСТ 28806-90].

Метрика якості ПЗ - кількісний масштаб і метод, які можуть бути використані для визначення значення ознаки, прийнятої для конкретної програмної продукції [ГОСТ Р ИСО/МЭК 9126-93]. Згідно IEEE Standard 610.12-1990, метрика визначається як міра ступеня володіння властивістю, яка має числове значення. Взагалі, метрика ПЗ - це міра, яка дозволяє одержати числове значення деякої властивості ПЗ або його специфікацій.

Основним стандартом якості в галузі інженерії ПЗ є стандарт ISO/IEC 25010:2011, який визначає номенклатуру, атрибути і метрики вимог якості ПЗ. Цей стандарт є одним з

визначальних факторів при моделюванні якості ПЗ. На додаток до нього випущено набір стандартів ISO/IEC 14598, які регламентують способи оцінки цих характеристик. В сукупності вони утворюють *модель якості*, відому під назвою *SQuaRE* (Software Quality Requirements and Evaluation).

В межах моделі SquaRE виділяються наступні характеристики якості (рис.15.3):

1) функційна придатність;
2) ефективність;
3) сумісність;
4) зручність використання;
5) надійність;
6) безпека;
7) супроводжуваність;
8) можливість переносу.

Тоді якість ПЗ можна представити функцією від восьми основних характеристик якості: функційної придатності, ефективності, сумісності, зручності використання, надійності, безпеки, супроводжуваності, можливості переносу, які представляють собою значення з певного діапазону. Але кожна з основних характеристик якості є функцією від декількох показників якості (рис.15.3).

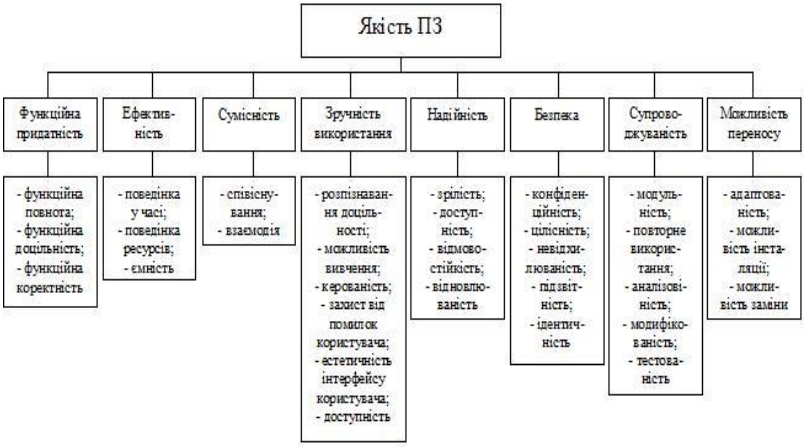


Рис.15.3 – Модель якості ПЗ за стандартом ISO/IEC 25010:2011

У таблиці 15.1 представлено зміст показників якості ПЗ згідно ISO/IEC 25010:2011.

Таблиця 15.1 – Зміст показників якості ПЗ

Характеристика якості ПЗ	Показник якості ПЗ	Зміст показника
Функційна придатність	функційна повнота	ступінь, в якому множина функцій покриває всі визначені завдання та цілі користувача
	функційна коректність	ступінь, в якому ПЗ забезпечує правильний результат з необхідним ступенем точності
	функційна доцільність	ступінь, в якому ПЗ сприяє досягненню визначених цілей та завдань
Ефективність	поведінка у часі	ступінь відповідності вимогам результата, часу обробки і пропускної здатності ПЗ при виконанні своїх функцій
	поведінка ресурсів	ступінь відповідності вимогам кількості та типів ресурсів, використовуваних ПЗ при виконанні своїх функцій
	ємність	ступінь відповідності вимогам максимальних меж параметрів ПЗ
Зручність використання	розпізнавання доцільності	можливість визначення користувачами, чи підтримуватиме ПЗ їх потреби ще до його реалізації
	можливість вивчення	навчання по використанню ПЗ, можливість використання ПЗ для досягнення поставлених цілей
	керованість	наявність в ПЗ атрибутів, які дозволяють легко ним керувати та контролювати його

	захист від помилок користувача	ступінь захисту користувачів від помилкових рішень
	естетичність інтерфейсу користувача	приємність інтерфейсу користувача та задоволення користувачів
	доступність	можливість використання ПЗ користувачами з найширшим діапазоном характеристик та можливостей
Надійність	зрілість	ступінь задоволення програмним забезпеченням потреб в надійності в умовах нормальної роботи
	наявність, доступність	функціонування та доступність ПЗ, коли воно потрібне
	відмовостійкість	можливість роботи ПЗ як передбачалося, незважаючи на наявність апаратних або програмних збоїв
	відновлюваність	можливість відновлення "постраждалих" даних та відновлення бажаного стану ПЗ в разі переривання або невдачі
Сумісність	співіснування	ефективність виконання функцій програмним забезпеченням при спільному використанні ресурсів з іншим ПЗ
	взаємодія	можливість обміну інформацією з іншим ПЗ та використання одержаної інформації
Безпека	конфіденційність	можливість гарантування, що дані доступні лише користувачам, уповноваженим мати до них доступ

	цілісність	можливість запобігання несанкціонованому доступу і зміні ПЗ та даних
	невідхилюваність	неможливість відхилення дій або подій, для яких доведено, що вони мали місце,
	підзвітність	можливість унікального відстеження дій користувача
	ідентичність	можливість доведенння ідентичності суб'єкта або ресурса заявленому об'єкту
Супроводжу-ваність	модульність	ПЗ складається з таких компонентів, що зміна одного з компонентів надає мінімальний вплив на інші компоненти
	повторне використання	основа (актив) ПЗ може бути використана при побудові іншого ПЗ
	аналізованість	ефективність, з якою можна оцінити вплив передбачуваних змін
	модифікованість	можливість ефективної зміни ПЗ без введення дефектів та без зниження якості
	тестованість	ефективність, з якою критерії випробувань можуть бути встановлені для ПЗ
Можливість переносу	адаптованість	можливість ПЗ ефективно адаптуватись до різного апаратного і програмного забезпечення або до різних оперативних середовищ
	можливість інсталяції	ефективність, з якою ПЗ може бути успішно встановлене та/або видалене
	можливість заміни	можливість заміни ПЗ на інший вказаний програмний продукт з тими ж цілями в тому ж середовищі

Така модель якості визначається загальними характеристиками продукту. Характеристики можуть розбиватись на під характеристики (показники) якості, які підлягають точному опису та виміру. Оцінка якості ПЗ відбувається наступним чином. Спочатку оцінюються показники якості ПЗ - обирається метрика, градуюється шкала оцінки в залежності від можливих ступенів відповідності атрибуту накладеним обмеженням. Набір "виміряних" показників представляє собою критерій для оцінки характеристики, а набір оцінок характеристик якості ПЗ є критерієм для комплексного оцінювання якості. При цьому проблемою є визначення і врахування взаємовпливів показників при визначенні характеристик якості та визначення і врахування взаємовпливів характеристик при оцінюванні якості ПЗ.

Існують десятки різних підходів до забезпечення якості ПЗ, і в кожного є свої переваги. Однією з перших моделей якості став стандарт ISO 9000. Сертифікати ISO серії 9000 зберігають популярність та визнаються в усьому світі. Але методики, покладені в основу стандартів серії ISO 9000 поступово застарівають. Недоліки стандартів серії ISO 9000:

1) недостатня деталізованість стандарту, можливість різних його трактувань в залежності від уявлень аудитора;

2) неточність оцінки якості процесів, задіяних при створенні та впровадженні ПЗ;

3) відсутність в стандарті механізмів, які сприяють покращенню існуючих процесів.

Перераховані проблеми змусили експертів розробляти більш досконалі рішення в галузі забезпечення якості ПЗ, що призвело до створення цілої низки нових стандартів та методологій (Capability Maturity Model (CMM), ISO/IEC 15504 (SPICE), Bootstrap, Trillium, ISO 12207). Найбільш вдалі та змістовні стнадарти - Capability Maturity Model (CMM) та ISO/IEC 15504 (SPICE).

Для визначення загального рівня розвитку технологічних процесів в програмних організаціях розробили спеціальну систему оцінки зрілості технологічних процесів в софтверних організаціях -

модель Capability Maturity Model (CMM), засновану на так званих рівнях зрілості (maturity levels) - рис.15.4.

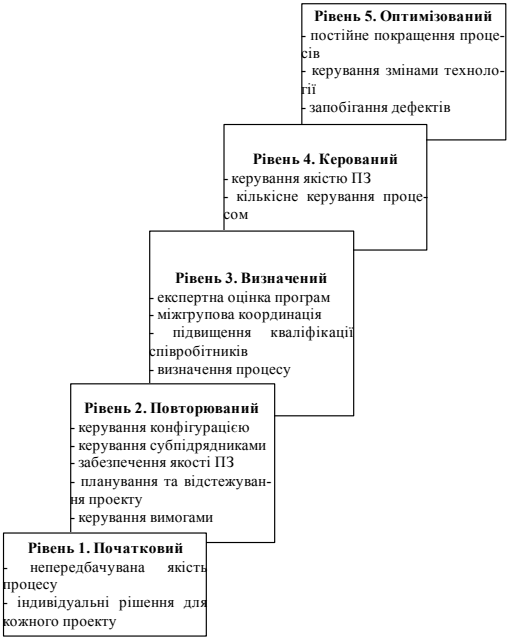


Рис.15.4 - Модель Capability Maturity Model

Очевидно, що модель CMM має *п'ять рівнів зрілості*, і кожен з них характеризує певний ступінь якості програмних продуктів:

1) початковий (initial) - відсутнє стабільне середовище розробки і супроводження; не витримуються терміни випуску продуктів; всі сили спрямовано на кодування і тестування програми (75% софтверних організацій);

2) повторюваний (repeatable) - жорстке керування, планування і контроль; акцент робиться на початкові вимоги, методи оцінки і конфігураційний менеджмент (15% софтверних організацій);

3) фіксований (defined) - процеси повністю документовані, стандартизовані і інтегровані в єдиний технологічний потік (8% софтверних організацій);

4) керований (managed) - намагання оцінити якість процесів і готового продукту кількісно; для контролю над процесами використовуються метрики (1.5% софтверних організацій);

5) оптимізований (optimizable) - намагання покращення роботи, керуючись кількісними критеріями якості; основна мета - випуск бездефектних продуктів, в яких помилки усунені ще на стадії внутрішнього тестування (0.5% софтверних організацій).

5-й рівень CMM вимагає постійного самостійного покращення процесу. Для покращення процесу керування проектом його ефективність вимірюється за допомогою метрик процесу. Ці метрики дозволяють виміряти ефективність організації процесу, а також аналізу вимог, проектування, програмування і тестування.

Використання CMM ускладнюють наступні проблеми:

1) стандарт CMM є власністю Software Engineering Institute і не є загальнодоступним, відтак подальша розробка стандарту ведеться самим інститутом без залучення іншої частини програмістської спільноти;

2) оцінка якості процесів організацій може проводитись лише спеціалістами, які пройшли спеціальне навчання та акредитовані SEI;

3) стандарт орієнтований на застосування у відносно великих софтверних компаніях.

Для оцінки рівня компанії за шкалою CMM розроблено декілька програмних пакетів: CMM Live - електронний довідник та експерт-радник по технологіях CMM; SoftGuide - орієнтований на розробників ПЗ, які намагаються покращити якість своїх процесів; SCOPE*PROCEPT - охоплює інформаційні моделі процесів створення ПЗ, включаючи методи оцінки якості на основі метрик.

Для одержання оцінки значень показників якості за стандартом ГОСТ 28195-89 використовуються такі *методи оцінки значень показників якості*:

1) вимірювальний - базується на використанні інструментальних, вимірювальних та спеціальних програмних засобів для одержання інформації про властивості та характеристики ПЗ (обсяг ПЗ, кількість рядків коду, кількість

операторів, кількість гілок в програмі, кількість точок входу/виходу та ін);

2) реєстраційний - заснований на одержанні інформації під час випробувань або функціювання ПЗ, коли реєструються або підраховуються певні події (час і кількість збоїв та відмов, час передачі керування від одного модуля до іншого, час початку і завершення роботи ПЗ);

3) органолептичний - заснований на використанні інформації, одержаної в результаті аналізу сприйняття органів чуття, і застосовується для визначення таких показників як зручність застосування, ефективність;

4) розрахунковий - базується на використанні теоретичних та емпіричних залежностей (на ранніх етапах розробки), статистичних даних, зібраних при проведенні випробувань, експлуатації та супроводженні ПЗ. Розрахунковими методами оцінюються показники надійності, точності, стійкості, час реакції, необхідні ресурси та ін.;

5) експертний - здійснюється групою експертів. Їх оцінка базується на досвіді та інтуїції, а не на безпосередніх результатах розрахунків або експериментів. Цей метод реалізується шляхом перегляду програм, кодів, супровідних документів, описів вимог до ПЗ групою експертів. Для цього встановлюються контрольовані ознаки, корельовані з одним або декількома показниками якості і включені в карти опитування експертів. Метод застосовується при оцінці таких показників, як наочність, повнота та доступність програмної документації, легкість засвоєння, аналізованість, документованість, структурованість ПЗ та ін.

З опису методів вимірювання показників (метрик) якості зрозуміло, що на етапі проектування ПЗ неможливо виміряти жодної характеристики ще не розробленого ПЗ, неможливо реєструвати моменти процесу виконання ще не існуючого ПЗ і неможливо сприйняти органами чуття інформацію щодо нерозробленого ПЗ. Отже, на етапі проектування є можливість визначати якість ПЗ лише із застосуванням розрахункових та експертних методів.

Методи керування якістю програмного забезпечення:

1) статичні методи (Static techniques) - передбачають детальне дослідження проектної документації, програмного забезпечення та іншої інформації про ПЗ без його виконання. Ці методи можуть включати інші методи (методи колективної оцінки, аналітичні методи) незалежно від рівня використання засобів автоматизації;

2) методи колективної оцінки (People-intensive techniques) - ідея полягає в формі прямої ("очної") взаємодії множини фахівців. Форма таких методів, включаючи оцінку і аудти, може варіюватись від формальних зборів до неформальних зустрічей або обговорення продукту навіть без звернень до його коду. Такі зустрічі можуть вимагати попередньої підготовки - формування переліку питань, які виносяться на обговорення. До ресурсів, які використовуються в таких методах, поряд з досліджуваними артефактами, можуть належати аркуші перевірки (checklists) та результати аналітичних методів та робіт по тестуванню.

3) аналітичні методи (Analytical techniques) - іноді декілька інженерів використовують однакові методи, але по відношенню до різних частин продукту. Деякі методи базуються на специфіці застосовуваних інструментів, інші - передбачають "ручну" роботу. Методи можуть допомагати знаходити дефекти напряму, але частіше вони використовуються для підтримки інших методів (наприклад, статичних). Ряд методів також включає різнорідну експертизу (assessment) як складовий елемент загального аналізу якості. Приклади таких методів - аналіз складності, аналіз керуючої логіки або аналіз контролю потоків керування, алгоритмічний аналіз. Кожний тип аналізу має конкретне призначення і не всі типи застосовні до будь-якого проекту. Інші, більш формальні, типи аналітичних методів відомі як формальні методи. Вони застосовуються для перевірки вимог, перевірки коректності (застосовно до критичних фрагментів ПЗ) та верифікації особливо важливих частин кричтино-важливих систем, наприклад, для перевірки конкретних вимог інформаційної безпеки та надійності;

4) динамічні методи (Dynamic techniques) - в основному, це методи тестування, а також методи симуляції, перевірки моделей та

"символічного" виконання (symbolic execution). Детально тестування ПЗ обговорюватиметься далі.

2. Надійність ПЗ. Моделі надійності. Розрахунок надійності за відомими моделями

Надійність ПЗ - це властивість програми виконувати задані функції в заданих умовах роботи і на заданій ЕОМ. За імовірнісним підходом під надійністю розуміють імовірність того, що при функціонуванні системи протягом певного часу не буде виявлено помилок. Проте тут не враховано відмінність між помилками різних типів. Надійність ПЗ краще розглядати з точки зору впливу помилок на користувача системи. Усе залежить від того, на якому етапі (розроблення чи експлуатація) і в якому компоненті припущено помилку. Надійність ПЗ не варто розглядати як внутрішню властивість програми. Вона тісно пов'язана з тим, як використовують програму. З цього погляду надійністю ПЗ є ймовірність його роботи без помилок і відмов протягом певного часу. Надійність є комплексним поняттям, яке залежно від призначення об'єкта і умов його застосування відображає такі властивості, як безвідмовність, довговічність, ремонтопридатність і збережуваність, або їх поєднання.

Надійне ПЗ повинне забезпечувати достатньо низьку ймовірність відмови в процесі функціонування в реальному часі. Швидке реагування на спотворення програм, даних або обчислювального процесу та відновлення роботоздатності за час, менший, ніж поріг між збоєм та відмовою, забезпечують високу надійність ПЗ. При цьому некоректне ПЗ може функціонувати абсолютно надійно. Отже, надійність функціонування ПЗ є поняттям динамічним, яке проявляється в часі та істотно відрізняється від поняття коректності ПЗ (коректність ПЗ - це відповідність його специфікації).

Надійність ПЗ певною мірою залежить від кількості помилок, внесених і не усунутих у процесі розроблення. У складних ПЗ неможливо забезпечити абсолютну відсутність дефектів проектування, внаслідок чого надійність їх функціонування завжди має обмежене значення.

Отже, *надійність ПЗ є показником якості*, який характеризує властивість ПЗ виявляти в процесі експлуатації помилки, що залишились в ньому, за певної сукупності початкових даних.

Показники надійності ПЗ:

1) критерій тривалості напрацювання на відмову - вимірюється час роботоздатного стану системи між двома послідовними відмовами або початком нормального функціонування системи після них;

2) тривалість відновлення - враховує можливість багаторазових відмов та відновлень;

3) коефіцієнт готовності - відображає ймовірність мати відновлювану систему в роботоздатному стані в довільний момент часу; значення коефіцієнту готовності відповідає долі часу корисної роботи системи на достатньо великому інтервалі, який містить відмови і відновлення;

4) напрацювання на ситуацію відмови, або стійкість - значення тривалості між втратами роботоздатності ПЗ незалежно від того, наскільки швидко відбулось відновлення.

Методи забезпечення надійності ПЗ:

1) попередження помилок - принципи і методи, які дозволяють мінімізувати або взагалі виключити помилки;

2) виявлення помилок - зосереджені на функціях самого ПЗ, які допомагають виявляти помилки;

3) виправлення помилок - функції ПЗ, призначені для виправлення помилок або їх наслідків;

4) забезпечення стійкості до помилок - це міра здатності ПЗ продовжувати функціонування при наявності помилок.

До *факторів гарантії надійності* належать:

1) ризик як сукупність загроз, які призводять до несприятливих наслідків та школи системи або середовища;

2) загроза як прояв нестійкості, яка порушує безпечність системи;

3) аналіз ризику - вивчення загрози або ризику, їх частота та наслідки;

4) цілісність - здатність системи зберігати стійкість роботи і не мати ризику.

Модель надійності ПЗ - це математична модель, побудована для оцінювання залежності надійності ПЗ від деяких певних параметрів. Значення таких параметрів або відомі, або можуть бути виміряні під час спостережень або експериментального дослідження процесу функціонування ПЗ. Термін "модель надійності ПЗ" може бути також використаний застосовно до математичної залежності між певними параметрами, які хоча й мають відношення до оцінки надійності ПЗ, але не містять її характеристик в явному вигляді.

Класифікація моделей надійності ПЗ наведена на рис.15.5.

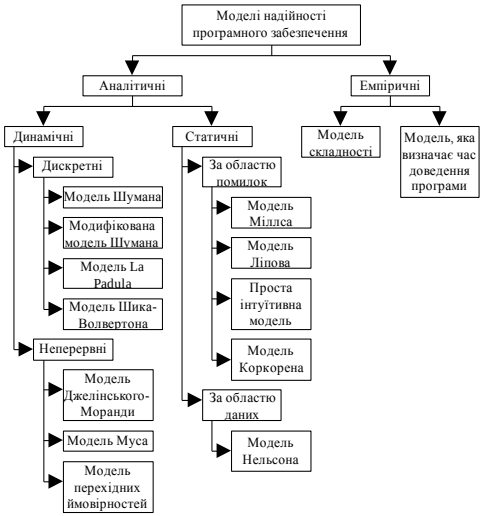


Рис.15.5 - Класифікація моделей надійності ПЗ

Моделі надійності програмного забезпечення (МНПЗ) поділяються на аналітичні та емпіричні. *Аналітичні моделі* дають змогу розрахувати кількісні показники надійності, базуючись на даних про поведінку ПЗ в процесі тестування (моделі вимірювання та оцінювання). *Емпіричні моделі* базуються на аналізі структурних особливостей програм. Вони розглядають залежність показників надійності від кількості міжмодульних зв'язків, кількості циклів в

модулях, відношення кількості прямолінійних ділянок програми до кількості точок розгалуження і т.і. Саме розвиток емпіричних моделей дозволяє виявляти взаємозв'язок між складністю ПЗ та його надійністю. Ці моделі можна використовувати на етапі проектування ПЗ, коли виконане розбиття на модулі та відома структура ПЗ.

Аналітичні моделі поділяються на дві групи: динамічні та статичні. В *динамічних МНПЗ* поява відмов ПЗ розглядається у часі. В *статичних моделях* поява відмов не пов'язана із часом, враховується лише залежність кількості помилок від кількості тестових прогонів (за областю помилок) або залежність кількості помилок від характеристики вхідних даних (за областю даних).

Особливий інтерес серед статичних моделей представляють моделі Коркорена та Нельсона.

Модель Коркорена не використовує параметри часу тестування, а враховує лише результат N випробувань, в яких виявлено N_i помилок i-го типу. Модель використовує ймовірності відмов, які змінюються для різних типів помилок. За моделлю Коркорена оцінюється ймовірність безвідмовного виконання програми на момент оцінки:

$$R = \frac{N_0}{N} + \sum_{i=1}^{K} (Y_i \cdot \frac{(N_i - 1)}{N}),$$

де N_0 - кількість безвідмовних виконань програми; N - загальна кількість прогонів; K - апріорі відома кількість типів; $Y_i = \begin{cases} a_i (N_i > 0) \\ 0 (N_i = 0) \end{cases}$, a_i - ймовірність виявлення при тестуванні помилки i-го типу, яка оцінюється на основі апріорної інформації або даних попереднього періоду функціонування типових програмних засобів.

Модель Нельсона враховує ймовірність вибору певного тестового набору для чергового виконання програми. Припускається, що область даних, необхідних для виконання тестування ПЗ, розділяється на K взаємовиключаючих

підобластей Z_i. Нехай P_i - ймовірність того, що набір даних Z_i буде обраний для чергового виконання програми. Припускаючи, що до моменту оцінювання надійності було виконано N_i прогонів програми на Z_i наборі даних, і з них n_i кількість прогонів завершилась відмовою, надійність ПЗ в цьому випадку становить:

$$R = 1 - \sum_{i=1}^{K} \frac{n_i}{N_i} \cdot P_i \ .$$

Модель Нельсона була розроблена з врахуванням основних властивостей ПЗ та використовує методи теорії ймовірностей лише в тих випадках, коли неможливим є одержання повної інформації про той чи інший фактор, наприклад, при відповіді на питання, який набір вхідних даних слід обрати при наступному прогоні програми. Внаслідок зазначених особливостей моделі її можна розглядати в цілому як математичну теорію надійності ПЗ, а не як просту модель надійності.

На базі моделі Нельсона можна виконувати оцінку технологічної безпеки ПЗ, якщо вважати безпеку ПЗ ймовірністю того, що програмні дефекти, які викликають критичну поведінку керованої комп'ютерної системи, будуть виявлені за певних умов зовнішнього середовища і протягом заданого періоду спостереження при випробуваннях.

Приклад. 15.1. Розрахувати ймовірність безвідмовного виконання програми за моделлю Коркорена, якщо відомо, що проведено 7 випробувань, з яких 4 випробування пройшли без відмов, виявлено 3 помилки логічних умов (ймовірність виявлення $a_1 = 0{,}29$), 2 помилки внутрішніх структур даних ($a_2 = 0{,}19$), 2 помилки порівняння ($a_3 = 0{,}22$), 2 помилки на граничних умовах ($a_4 = 0{,}15$) та 1 помилка незалежних маршрутів програми ($a_5 = 0{,}15$).

Для даного прикладу: $N_0 = 4$, $N = 5$, $K = 5$, $Y_1 = a_1 = 0{,}29$, $Y_2 = a_2 = 0{,}19$, $Y_3 = a_3 = 0{,}22$, $Y_4 = a_4 = 0{,}15$, $Y_5 = a_5 = 0{,}15$, $N_1 = 4$, $N_2 = 2$, $N_3 = 2$, $N_4 = 1$, $N_5 = 1$.

Тоді:

$$R = \frac{N_0}{N} + \sum_{i=1}^{K}(Y_i \cdot \frac{(N_i - 1)}{N}) = \frac{4}{7} + [(0.29 \cdot \frac{3-1}{7}) + (0.19 \cdot \frac{2-1}{7}) +$$

$$+ (0.22 \cdot \frac{2-1}{7}) + (0.15 \cdot \frac{2-1}{7}) + (0.15 \cdot \frac{1-1}{7})] = 0.57 +$$

$$+ [0.083 + 0.027 + 0.031 + 0.021 + 0] = 0.732$$

Приклад. 15.2. Розрахувати надійність ПЗ (ймовірність безвідмовної роботи) за моделлю Нельсона, якщо відомо, що область даних для виконання тестування ПЗ розділяється на 5 взаємовиключаючих підобластей з рівними ймовірностями вибору; виконано 7 прогонів програми на першій підобласті (3 з них завершились відмовою), 4 прогони - на другій підобласті (3 відмови), 2 прогони - на третій підобласті (1 відмова), 2 прогони - на четвертій підобласті (1 відмова), 4 прогони - на п'ятій підобласті (2 відмови).

Отже, для даного прикладу: $K = 5$, $N_1 = 7$, $N_2 = 4$, $N_3 = 2$, $N_4 = 2$, $N_5 = 4$, $n_1 = 3$, $n_2 = 3$, $n_3 = 1$, $n_4 = 1$, $n_5 = 2$, $P_1 = P_2 = P_3 = P_4 = P_5 = \frac{1}{5} = 0{,}2$.

Тоді:

$$R = 1 - \sum_{i=1}^{K} \frac{n_i}{N_i} \cdot P_i = 1 - [(\frac{3}{7} \cdot 0.2) + (\frac{3}{4} \cdot 0.2) + (\frac{1}{2} \cdot 0.2) + (\frac{1}{2} \cdot 0.2) +$$

$$+ (\frac{2}{4} \cdot 0.2)] = 1 - [0.086 + 0.15 + 0.1 + 0.1 + 0.1] = 1 - 0.536 = 0.464$$

3. Складність ПЗ. Моделі та методи складності

За особливостями і властивостями життєвого циклу програм їх доцільно ділити на ряд класів і категорій, з яких найбільш розрізняються два великих класи - малі й великі.

Клас малих програм складають відносно невеликі програми, які створюються одним або декількома (3-5) програмістами. Малі програми створюються переважно для одержання конкретних

результатів та аналізу відносно простих процесів. Вони не призначені для масового поширення, не мають конкретного незалежного замовника-споживача, не обмежуються замовником певною вартістю, трудомісткістю, вимогами заданої якості та термінами їх створення та не підлягають незалежному тестуванню, гарантуванню якості та сертифікації. До малих програм можна віднести програми, призначені для проміжних розрахунків, для потреб програміста і т.і.

Клас великих програм складають масштабні комплекси програм для складних систем управління та обробки інформації, які оформляються у вигляді програмних продуктів з гарантованою якістю. Великі програми мають великий розмір та високу вартість. Від замовника великої програми або програмної системи розробники повинні одержати конкретні вимоги до характеристик та функційності продукта. Від розробників проектів вимагаються гарантії високої якості, надійності функціювання та безпеки застосування компонентів і програмних систем. До великих програм відноситься системне програмне забезпечення, а також деяке прикладне програмне забезпечення, наприклад, офісні пакети, графічні редактори і т.і. Великі програми називають ще програмними системами (ПС) або інформаційними системами (ІС).

Програмна складність характеризується довжиною програми або обсягом пам'яті ЕОМ, необхідної для розміщення ПЗ.

Структурна складність програм визначається кількістю взаємодіючих компонентів, кількістю зв'язків між компонентами та складністю їх взаємодії.

Складність деякого міжмодульного зв'язку в процесі проектування можна характеризувати ймовірністю помилки при її формалізації та ступенем впливу цієї помилки на наступне функціювання модулів.

Складність програмних модулів характеризується конструктивною складністю створення оформленої компоненти програми і оцінюється з позиції складності внутрішньої структури та перетворення змінних в кожному модулі, а також інтегрально за деякими зовнішніми статичними характеристиками модулів.

Складність програми для систем реального часу переважно визначається допустимим часом відгуку, а для інформаційних систем - кількістю типів оброблюваних змінних.

Основні види складності ПЗ наведено на рис.15.6.

Рис.15.6 - Основні види складності ПЗ

Схема взаємодії показників складності показана на рис.15.7.

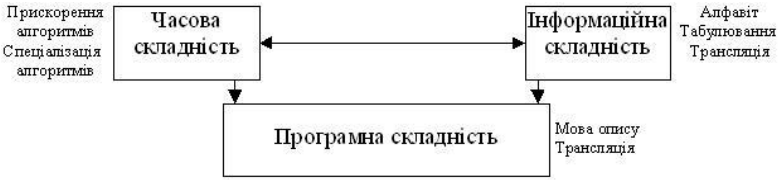


Рис.15.7 - Схема взаємодії показників складності

Як же виміряти складність різноманітних алгоритмів? Зрозуміло, що програма повинна надавати коректну відповідь на всі легальні вхідні дані, а також ефективно виконувати обчислення. Коректні відповіді, як правило, більш важливі, але іноді критичним є час виконання програми (наприклад, для програм виявлення колізій, керування світлофорами). Та й навіть, якщо коректність програми більш важлива, то часто оптимізації повинен підлягати і

час виконання програми (наприклад, для програмного забезпечення керування літаками).

Отже, *можна оцінити складність часом виконання програми для скільки завгодно великого вхідного аргумента*. Можна також *оцінити складність алгоритму обсягом пам'яті, потрібним для виконання програми*. Взагалі, потрібен баланс мінімізованої обчислювальної складності та концептуальної складності - розробляти та зберігати код, простий та легкий для розуміння, але по можливості оптимізувати продуктивність програми.

Рзін класи алгоритмів мають різний час виконання, а відтак і різну складність. Як же виміряти складність, розглядаючи її як час виконання програми? Є декілька шляхів:

1) запустити програму на деяких даних і засікти час виконання, але тут є певні проблеми: залежність від швикдості комп'ютера; залежність від реалізації певною мовою програмування; підбір вірних (за розміром) вхідних даних;

2) виконувати підрахунок кількості основних кроків (операцій), які виконуються для обчислення функції, з точки зору розміру вхідних даних. У такому разі час виконання програми вимірюватиметься як функція від розміру вхідних даних. Тоді й складність залежить від значень (розміру) вхідних даних.

Розглянемо простий приклад. Оцінимо складність алгоритму лінійного пошуку елемента у спписку, реалізованого мовою програмування Python:

```
def linearSearch(L,x):
      for e in L:
            if e==x:
                return True
      return False
```

У такому випадку, якщо шуканий елемент знаходиться на початку списку, то значення `True` повертається практично одразу після запуску підпрограми. Якщо ж шуканого елементу немає у списку, то потрібно виконати перегляд всього списку. Зрозуміло, що чим більший розмір має список, тим довше виконуватиметься операція його перегляду, відтак тим довше працюватиме підпрограма.

Як правило, для оцінки часу виконання з метою оцінки складності алгоритму розглядається найгірший (найдовший) шлях виконання алгоритму.

Розглянемо інший приклад. Оцінимо максимальну кількість кроків (операцій) нерекурсивного алгоритму пошуку факторіалу числа, реалізованого мовою програмування Python:

```
def fact(n):
    answer=1
    while n>1:
          answer*=n
          n-=1
    return answer
```

Підрахуємо кількість операцій:

- 1 операція - операція присвоєння (`answer=1`);

- 5 операцій для кожної ітерації циклу - перевірка умови виходу з циклу (`n>1`), множення та присвоєння (`answer*=n`), віднімання та присвоєння (`n-=1`);

- `n` ітерацій циклу.

Отже, кількість операцій в такому разі: *5*n+2*, але коли *n* - велике, то доданок 2 практично не має значення, тоді кількість кроків (операцій) становить *5*n*, тобто дорівнює добутку кількості ітерацій циклу на кількість операцій у циклі. Константа (в даному прикладі - число 5) називається мультиплікативною константою (мультиплікативним фактором).

В ході наших експериментальних досліджень було зроблено висновок, що складність алгоритму і час виконання алгоритму залежать від значень (розміру) вхідних даних. Для того, щоб говорити про такі залежності, потрібен формальний шлях.

Емпіричні правила для оцінки складності алгоритму:

1) асимптотична складність - описується час виконання алгоритму з точки зору кількості основних кроків; якщо час виконання є сумою мультиплікативних факторів, то враховується один з більшою швидкістю зростання; якщо останні терми є добутком, то будь-які мультиплікативні фактори не враховуються;

2) використовується нотація big O для верхньої границі асимптотичного зростання функції.

Класи складності алгоритмів (в порядку зростання складності):

1) `O(1)` - постійна складність (константний час виконання), тобто складність не залежить від розміру вхідних даних, і кількість кроків не залежить від розміру алгоритма. У цьому класі міститься мала кількість алгоритмів, але часто інші алгоритми можуть містити фргаменти, які відповідають цьому класу (наприклад, циклічні або рекурсивні алгоритми, в яких кількість ітерацій або викликів не залежить від вхідних даних)

2) `O(log n)` - логарфмічна складність (логарифмічний час виконання), тобто складність зростає як логарифмічна функція від розміру одного з його входів (наприклад, алгоритм бінарного пошуку);

3) `O(n)` - лінійна складність (лінійний час виконання), тобто складність зростає лінійно із зростанням розміру вхідних даних (наприклад, алгоритм лінійного пошуку, рекурсивний алгоритм, у якому складність залежить від кількості рекурсивних викликів);

4) `O(n log n)` - логарифмічно-лінійна складність (логарифмічно-лінійний час виконання) є складністю багатьох практичних алгоритмів (наприклад, алгоритм сортування злиттям);

5) $O(n^c)$, де `c-const` - поліноміальна складність (поліноміальний час виконання), тобто складність зростає як степінь розміру входу (наприклад, у випадку вкладених циклів або вкладених рекурсивних викликів);

6) $O(c^n)$, де `c-const`, яка зростає як степінь на основі розміру входів - експоненційна складність (експоненційний час виконання) є складністю багатьох важливих проблем (наприклад, рекурсивні функції, які мають більше одного рекурсивного виклику - Ханойські башти), але вартість алгоритмів такого класу складності досить висока, що й призводить до пошуку наближених розв'язків екпоненційних проблем.

Приклад 15.3. Визначити класи складності фрагментів програм мовою `Python`:

1)

```
# Нехай змінна n була раніше зв'язана з
деяким значенням
i = 0
while i < 5:
   n *= 2
   i += 1
print n
```

2)

```
def iterPower(a, b):
   result = 1
   while b > 0:
      result *= a
      b -= 1
   return result
```

3)

```
def recurPower(a, b):
   print a, b
   if b == 0:
      return 1
   else:
      return a * recurPower(a, b-1)
```

4)

```
def recurPowerNew(a, b):
   print a, b
   if b == 0:
      return 1
   elif b%2 == 0:
      return recurPowerNew(a*a, b/2)
   else:
      return a * recurPowerNew(a, b-1)
```

Виходячи з вищенаведених міркувань стосовно класів складності алгоритмів та програм, можна зробити наступні висновки щодо класу складності заданих фрагментів програм:

1) `O(1)` - постійна складність;

2) `O(b)` - лінійна складність;

3) `O(b)` - лінійна складність;

4) `O(log(b))` - логарифмічна складність.

Моделі складності ПЗ засновані на гіпотезі, що рівень безпомилковості ПЗ може бути спрогнозований за допомогою показників (метрик) складності ПЗ. Це вірно для ненавмисних дефектів, оскільки, чим складніша та більша програма, тим вища ймовірність того, що програміст припуститься помилки при її написанні та модифікації. В якості аргументів моделей, як правило, використовуються метрики складності ПЗ, а самі моделі складності можна розділити на апріорні, статистичні та емпіричні.

Найбільш відомою *апріорною моделлю* складності ПЗ є модель *Холстеда* (Halstead). В основу розроблення моделі покладено дві базові характеристики ПЗ: словник операторів та операндів мови програмування η та кількість використань операторів та операндів у програмних реалізаціях N, а також гіпотеза, що частота використання операторів та операндів у програмі пропорційна двійковому логарифму кількості їх типів (за аналогією з теорією інформації). На основі описаних характеристик та гіпотези, а також фізіологічних характеристик людини при реалізації програми, описаної даними характеристиками, можна одержати низку емпіричних моделей оцінки показників якості ПЗ. Складність ПЗ запропоновано розглядати як сукупність інтелектуальних зусиль (розв'язок елементарних задач людиною до виникнення помилки) при кодуванні текста певною мовою програмування:

$$E = \tilde{N} \cdot \log_2(\frac{\eta}{L}) = \frac{\eta_1 \cdot N_2 \cdot N \cdot \log_2 \eta}{2 \cdot \eta_2},$$

де $\tilde{N} = \eta_1 \cdot \log_2 \eta_1 + \eta_2 \cdot \log_2 \eta_2$ - теоретична довжина програми, $\eta = \eta_1 + \eta_2$ - кількість унікальних операторів та операндів мови програмування, $L = \frac{2 \cdot \eta_2}{\eta_1 \cdot N_2}$ - якість програмування, $N = N_1 + N_2$ - кількість звертань до операторів та операндів у ПЗ.

Для розрахунку D_0 початкової кількості помилок у ПЗ запроновано використовувати вираз: $D_0 = \frac{V}{3000}$, де $V = N \cdot \log_2(\eta_1 + \eta_2)$ - обсяг програми (в бітах інформації).

Відома також *статистична модель Холстеда* для розрахунку кількості помилок на етапах налагодження та випробувань у ПЗ: $D_m = k_m \cdot E^{\frac{2}{3}}$, де k_m - коефіцієнт пропорційності (наприклад, для мейнфреймів $k_m = 0{,}0003125$).

До простих *статистичних моделей* складності належить *феноменологічна модель фірми TRW*. Феноменологічна модель представляє собою лінійну модель оцінювання показника складності ПЗ за 5-ма емпіричними характеристиками програми, а саме: логічної складності L_{tot}, складності взаємозв'язків C_{in}, складності обчислень C_c, складності введення-виведення C_{io} та зрозумілості U_{read}:

$$C = L_{tot} + 0.1 \cdot C_{in} + 0.2 \cdot C_c + 0.4 \cdot C_{io} + (-0.1) \cdot U_{read}.$$

Для розрахунку кількості помилок N використовується наступна багатофакторна модель:

$$N = L_{tot} \cdot k_1 + 0.1 \cdot C_{in} \cdot k_2 + 0.2 \cdot C_c \cdot k_3 + 0.4 \cdot C_{io} \cdot k_4 + \\ + (-0.1) \cdot U_{read} \cdot k_5,$$

де k_i - коефіцієнт кореляції кількості помилок з i-м показником складності. Значення k_i легко знайти за методом найменших квадратів. До недоліків багатофакторних моделей належить можливість одержання ефекту "прокляття розмірності" при великій кількості факторів або нелінійному вигляді апроксимуючого полінома.

Емпіричні (імітаційні) моделі базуються на аналізі структурних особливостей програмного забезпечення. Такі моделі часто не дають кінцевих результатів показників складності, якості та надійності, але їх використання на етапі проектування ПЗ корисне для прогнозування потрібних ресурсів тестування,

уточнення планових термінів заверешння проекту і т.і. Якщо дотримуватись спрощеного розуміння складності ПЗ, то вона може описуватись такими характеристиками, як розмір ПЗ (кількість програмних модулів - програмних одиниць, які виконують певні функції та пов'язані з іншими модулями ПЗ), кількість та складність міжмодульних інтерфейсів.

Для складних модулів та для великих багатомодульних програм складається *імітаційна модель*, програма якої "засівається" помилками і тестується за випадковими входами. Оцінювання надійності (а відтак і складності) здійснюється за моделлю Міллса.

Переваги оцінювання показників складності та надійності за імітаційною моделлю, створеною на основі аналізу структури майбутнього реального ПЗ:

1) модель дозволяє на етапі проектування ПЗ приймати оптимальні проектні рішення, спираючись на характеристики помилок, які оцінюються за допомогою імітаційної моделі;

2) модель дозволяє прогнозувати потрібні ресурси тестування;

3) модель дає можливість визначити міру складності програм та спрогнозувати можливу кількість помилок і т.і.

До недоліків такої моделі можна віднести високу вартість методу, оскільки він вимагає додаткових витрат на складання імітаційної моделі, і наближений характер одержаних показників.

Динамічна (або обчислювальна) складність характеризує процес виконання програми і має три складових:

1) часова - визначається часом виконання програми або часом її реакції на запит користувача;

2) програмна - визначається складом та способом взаємодії процедур або модулів, які утворюють програму, а також можливістю їх розташування в кеш-пам'яті, основній пам'яті або на диску, а у випадку розподілених додатків - розташуванням програми на комп'ютерах мережі;

3) інформаційна - визначається складністю організації даних та доступу до них, а також особливостями їх розташування в кеш-пам'яті, основній пам'яті, на диску або на мережевому сервері.

Крім розглянутих моделей існують також структурні моделі, нечіткі моделі, інтервальні моделі складності ПЗ, а також нейронні мережі, які застосовуються для розв'язку задачі оцінювання складності ПЗ в окремих випадках.

Отже, класифікацію моделей визначення складності ПЗ можна представити у наступному вигляді (рис.15.8):

Рис.15.8 - Класифікація моделей надійності програмного забезпечення

Для вибору потрібної моделі використовуються якісні та кількісні критерії.

Якісні критерії:

1) простота використання - вхідні дані для моделі повинні легко одержуватись та бути зрозумілими експертам;

2) достовірність - модель повинна мати точність, необхідну і достатню для розв'язання задач аналізу та синтезу в галузі безпеки ПЗ;

3) застосовність для розв'язку різних задач - деякі моделі дозволяють одержати оцінки широкого спектру показників, необхідних експертам на різних етапах життєвого циклу ПЗ;

4) простота реалізації - можливість автоматизації процесу оцінювання, перенавчання моделі після доопрацювань, врахування випадків неповних або некоректних вхідних статистик, врахування інших обмежень моделей.

Кількісні критерії:

1) показники точності оцінювання;

2) показники якості прогнозуючих моделей (збіжність, стійкість, точність прогнозу, узгодженість);

3) інформаційні критерії якості прогнозуючих моделей (розмірність, критерії BIC/AIC);

4) комбіновані та інтегральні показники.

Дослідження показало, що існує велика кількість математичних моделей складності ПЗ, які дозволяють одержати оцінки показників технологічної безпеки ПЗ на різних етапах життєвого циклу, що важливо при плануванні витрат на інформаційну безпеку. Розглянута класифікація моделей дозволяє на практиці зорієнтуватись при виборі моделей в залежності від одержаної статистики.

Розглянемо тепер методи визначення складності програмного забезпечення. На сьогодні найбільш поширеним та добре документованим методом оцінювання складності програмного забезпечення є *метод функціональних точок*. Метод функціональних точок - це стандартний метод вимірювання розміру ПЗ з точки зору його користувачів. Метод розроблено Аланом Альбрехтом в 1979 році. Розрахунок функціональних точок - це метод вимірювання, при якому шляхом підрахунку кількості введень, виведень, запитів, баз даних, а також системних інтерфейсів програмного забезпечення визначається його міра складності та обсягу. В результаті досліджень була запропонована схема послідовності кроків при розрахунку кількості функціональних точок (рис.15.9). Метод призначений для оцінювання на основі логічної моделі обсягу програмного забезпечення кількістю функціоналу, який вимагає замовник та постачає розробник. Вимірювання для методу функціональних точок не залежать від технологічної платформи, на якій розроблятиметься ПЗ, і він забезпечує єдиний підхід до оцінювання всіх проектів софтверної компанії. Метод функціональних точок дозволяє визначити розмір ПЗ в абстрактних одиницях вимірювання, які відображають його функціональність - функціональних балах. Це дозволяє усунути залежність оцінки від

середовища розроблення. Крім того, подібні абстрактні одиниці вимірювання добре підходять для оцінок розміру проекту на ранніх етапах життєвого циклу ПЗ. Однак сама методика застосовна лише для певного типу ПЗ - систем оброблення даних - і не є універсальною. Розрахунок кількості функціональних балів вимагає значного часу на збирання даних про систему. Точність сильно залежить від кваліфікації фахівця, який проводив оцінку.

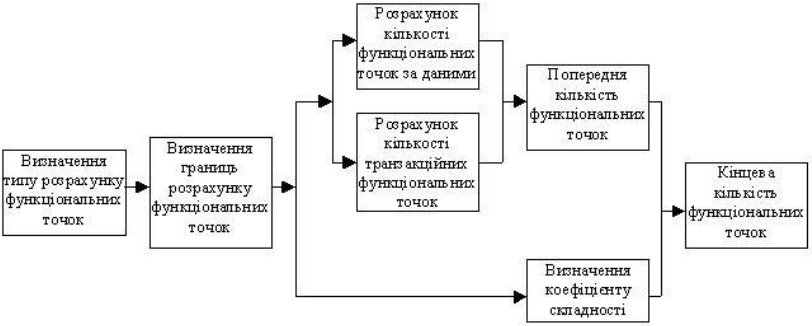


Рис.15.9 - Високорівнева схема розрахунку кількості функціональних точок

Метод функціональних точок може бути успішно застосований для:

1) визначення трудомісткості та вартості планованих проектів розроблення програмного забезпечення;

2) проведення порівняльного аналізу якості та продуктивності розроблення різнотипних проектів, або однотипних проектів, при виконанні яких використовувались різні технології;

3) проведення аналізу планової та реальної оцінки складності та величини розробленого ПЗ та трудомісткості виконання проекту, одержання стандартної метрики порівняння програмних продуктів.

Незважаючи на переваги методу оцінювання складності ПЗ за функціональними точками, він має низку недоліків, які істотно обмежують можливості його застосування:

1) для роботи за методом функціональних точок потрібні висококваліфіковані сертифіковані фахівці;

2) розрахунок кількості функціональних точок потребує значного часу на збір даних про ПЗ та одержання повного розуміння вимог до ПЗ;

3) при застосуванні методу на ранніх етапах розроблення в багато разів знижується точність оцінки;

4) метод "інтуїтивно" не зрозумілий, оскільки оперує багатьма поняттями, використовуваними у сучасному об'єктно-орієнтованому підході.

Характер проектованої системи та структурна складність її програмного забезпечення є визначальними факторами, від яких залежать витрати ресурсів на розроблення ПЗ. Для оцінювання структурної складності існує достатня кількість *метрик*: Холстеда, Маккейба, Майєрса, Джилба, Хансена, Вудворда. Однак всі вони передбачають обов'язкову наявність для аналізу або тексту самої програми, або детального алгоритму її роботи. Цей факт робить неприйнятним використання даних підходів з метою досягнення високої точності на ранніх етапах життєвого циклу ПЗ, коли у розробника є лише концептуальні моделі архітектури майбутнього ПЗ та окремі ескізні проектні рішення. *Розмір ПЗ є найпоширенішою характеристикою для оцінювання структурної складності на ранніх етапах процесу розроблення.* Розмір ПЗ в сучасній практиці серед всіх факторів, які впливають на параметри проекту, змінюється на три-чотири порядки - від 104 до 107 рядків тексту програми. Тому методам його оцінювання приділяється велика увага при аналізі методів визначення складності.

Найпростішою та найбільш широко використовуваною метрикою вимірювання розміру ПЗ є *розмір програми в рядках коду* (*LOC* - Line of code). В якості методу оцінювання розміру ПЗ за допомогою даної метрики використовується *метод аналогій*. При його використанні виникають труднощі з визначенням параметрів, за якими визначається подібність проектів. Звичайно ці параметри визначаються експертним методом, що вносить певну частку суб'єктивності в оцінку.

Метод об'єктних точок (OP - object point) є розвитком ідеї, закладеної в методі функціональних точок. При підрахунку використовуються більш великі програмні об'єкти, ніж у методі

функціональних точок. Це дозволяє виконувати оцінку швидше і простіше. Метод ще менш універсальний, ніж метод функціональних точок, оскільки використовує для підрахунку обмежений набір програмних об'єктів - лише вікна, звіти та 3GL-компоненти. При цьому кожний програмний об'єкт повинен бути віднесений до певної групи складності з використанням процедури експертної оцінки.

В результаті огляду формальних методів прогнозування розміру ПЗ, використовуваних на ранніх етапах життєвого циклу, можна навести зведену таблицю із вказанням переваг та недоліків існуючих підходів (таблиця 15.2).

Таблиця 15.2 - Порівняння методів оцінювання складності програмного забезпечення

Методи визначення розміру ПЗ	Одиниця вимірювання	Незалежність від середовища розроблення	Легкість застосування	Відсутність впливу суб'єктиності на результат	Калібрування за даними	Універсальність для різних типів проектів
Метод аналогії	LOC	-	±	-	+	±
Метод функціональних точок (Function Points)	FP	+	-	±	-	-
Метод об'єктних точок (Object Points)	OP	+	+	±	-	-

Альтернативним методом визначення складності ПЗ є *структурний метод оцінювання складності ПЗ*. Структурний метод складається з наступних основних частин (рис.15.10).

На сьогодні вирішення задач оцінювання техніко-економічних параметрів проектів розроблення ПЗ відбувається також і за допомогою експертних методів. Такі оцінки мають сильну залежність від компетенції та об'єктивності експерта. Подібні оцінки можуть легко та швидко одержуватись, але водночас і важко перевірятись, і слабо обгрунтовуватись. Також

для оцінювання складності ПЗ можна використовувати й нейромережні методи, один з яких і буде розглянуто пізніше.

Рис.15.10 - Структурний метод оцінювання складності ПЗ

Отже, класифікацію методів визначення складності ПЗ на етапі проектування можна представити наступним чином (рис.15.11).

Результати аналізу існуючих методів оцінювання складності ПЗ показали, що деякі з них хоч і задовольняють основні вимоги до технології оцінки складності, але в той же час вони недостатньо гнучко враховують різноманітні фактори, які впливають на терміни та тривалість проекту, а їх застосування на практиці виявляється вельми трудомістким та дорогим.

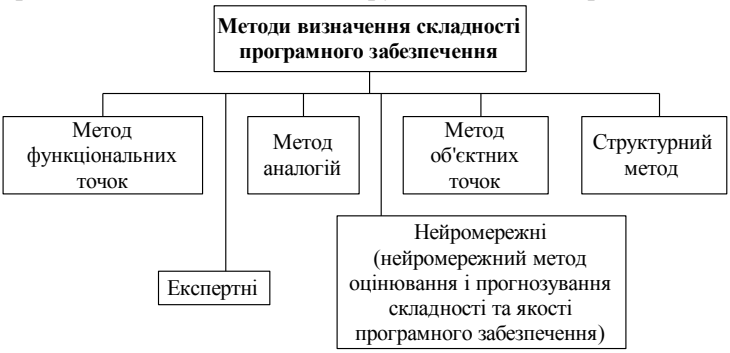


Рис.15.11 - Методи визначення складності ПЗ

4. Основні принципи метричного аналізу

Згідно IEEE Std 610.12-1990, *метрика* визначається як міра ступеня володіння властивістю, яка має числове значення. Взагалі, метрика ПЗ - це міра, яка дозволяє одержати числове значення деякої властивості ПЗ або його специфікацій. Сучасна програмна індустрія накопичила велику кількість метрик, які оцінюють окремі виробничі та експлуатаційні властивості ПЗ. Однак прагнення їх універсальності, неврахування типу та області застосування розроблюваного ПЗ, ігнорування етапів життєвого циклу ПЗ та необгрунтоване їх використання в процедурах прийняття виробничих рішень істотно підірвало довіру розробників та користувачів ПЗ до метрик. Ці обставини вимагають ретельного відбору метрик для певного типу та області застосування розроблюваного ПЗ, врахування їхніх обмежень на різних етапах життєвого циклу ПЗ, встановлення порядку їх сумісного використання, накопичення та інтеграції різнорідної метричної інформації для прийняття своєчасних виробничих рішень.

Незважаючи на численні дослідження програмних метрик, в галузі дослідження метрик якості ПЗ залишається *ряд невирішених питань:*

1) технологія вимірювання якості ще не досягла зрілості - лише 1.5% софтверних організацій намагаються оцінити якість процесів і готового продукту кількісно, за допомогою метрик, і лише 0.5% софтверних організацій намагаються покращити роботу, керуючись кількісними критеріями якості з метою випуску бездефектних продуктів;

2) відсутні єдині стандарти на метрики - створено більше тисячі метрик, кожен постачальник "вимірювальної" системи пропонує власні способи оцінки якості і відповідно метрики;

3) існує проблема складності інтерпретації величин метрик - значення метрик, одержані за допомогою "вимірювальних систем", неінформативні або малоінформативні для користувача, замовника, а часто і для програміста;

4) метрики розраховуються лише для готового ПЗ - всі "вимірювальні" системи орієнтовані на розрахунок метрик для програмного коду, але часто є необхідність у розрахунку метрик

вже на етапі проектування - метрик якості з точними значеннями для проекту ПЗ і метрик якості з прогнозованими значеннями для розроблюваного за проектом програмного забезпечення;

5) низький рівень автоматизації аналізу та опрацювання метрик якості програмного забезпечення - на сьогодні автоматизовано лише процеси збирання, реєстрації та обчислення метричної інформації;

6) відсутня можливість порівняння принципово нового проекту з попередніми, що призводить до неможливості інтерпретації одержаних метрик для нового проекту;

7) основними параметрами при виборі варіанту реалізації ПЗ є вартість та тривалість розроблення і репутація фірми-проектувальника, але рішення, прийняті на основі цих параметрів, не завжди гарантують належну якість ПЗ.

Саме через невирішеність цих питань не дозволяє підвищувати якість ПЗ та зменшувати його складність, саме через неї поки що неможливо створити бездефектне високоякісне ПЗ. Складність обгрунтування вибору та інтерпретації метрик в процедурах прийняття виробничих рішень та ігнорування етапів життєвого циклу ПЗ не дозволяють повноцінно використовувати метрики для підвищення якості ПЗ.

Розглянемо основні *метрики складності* програмного забезпечення.

Метрика Чепіна - аналізує характер використання змінних зі списку введення, тобто оброблювану інформацію. Множина змінних з врахуванням їх значущості обчислюється як:

$$Q = P + 2\mathrm{M} + 3\mathrm{C} + 0{,}5T,$$

де P - змінні для розрахунків і виведення; M - модифіковані або створені в програмі змінні; C - керуючі змінні; T - не використовувані в програмі ("паразитні") змінні. Метрика Чепіна дає оцінку інформаційної міцності модуля.

Метрика Джилба (складова метрики - модульна складність програми) - на етапі проектування можна підрахувати кількість міжмодульних зв'язків N_{zv} (абсолютна модульна складність) та відношення кількості міжмодульних зв'язків до

кількості модулів L_{mod}: $f = \frac{N_{zv}^4}{L_{\text{mod}}}$ - відносна модульна складність. Чим вище значення відносної модульної складності проекту, тим вищий ступінь має зчеплення модулів проекту, а, отже, й складнішим є програмний проект.

За іншою методикою кількість умовних операторів (L_{IF}) та операторів циклу (L_{LOOP}) складає абсолютну логічну складність програми

$$CL = L_{IF} + L_{LOOP},$$

а відношення *абсолютної логічної складності* програми до загальної кількості операторів L складає *відносну логічну складність* програми

$$cl = \frac{CL}{L}.$$

Метрика Мак-Клура - метрика, спрямована на оцінювання архітектури системи; міра складності, заснована на кількості можливих шляхів виконання програми, кількості керуючих конструкцій і змінних.

Обчислюється в 3 етапи:

1) для кожної керуючої змінної i обчислюється значення її функції складності

$$C(i) = (D(i) * J(i))/n,$$

де $D(i)$ - величина, яка вимірює сферу дії змінної (для локальної змінної $D(i) = 0$, а для глобальної змінної - $D(i) = 1$), $J(i)$ - міра складності взаємодії модулів через цю змінну (скільки разів модулі взаємодіяли з використанням цієї змінної), n - кількість модулів;

2) для всіх модулів визначаються функції складності

$$M(P) = fp * X(P) + gp * Y(P),$$

де fp, gp - відповідно кількість модулів, безпосередньо передуючих і безпосередньо слідуючих за модулем P, $X(P)$ - складність звертання до модуля P (скільки разів відбувається

звертання до модуля P), $Y(P)$ - складність управління викликом з модулю P інших модулів (скільки разів модуль P викликає інші модулі);

3) загальна складність MP програмної системи:

$$MP = \sum_{P} M(P).$$

Дана метрика орієнтована на добре структуроване програмне забезпечення, для якого можна побудувати схему розбиття ПЗ на модулі. В кожному модулі передбачена одна точка входу і одна точка виходу, модуль виконує одну функцію, виклик модуля здійснюється відповідно до ієрархічної системи керування. Метрика Мак-Клура дозволяє обрати схему розбиття з меншою складністю ще до написання програми.

Метрика Кафура - метрика, заснована на врахуванні потоків даних. Для розрахунку цієї метрики вводяться поняття локального і глобального потоків даних. Якщо модуль А викликає модуль В, то це прямий локальний потік з А в В; якщо модуль В викликає модуль А, і модуль А повертає значення в модуль В - це непрямий локальний потік з А в В; якщо модуль С викликає модулі А і В та передає результат з модуля А в модуль В - це локальний потік з А в В. Глобальний потік з модуля А в модуль В крізь глобальну структуру даних D існує, якщо модуль А поміщає інформацію в структуру D, а модуль В використовує інформацію зі структури D. Тоді:

1) інформаційна складність процедури обчислюється за формулою:

$$I = length \times (fan_{in} \times fan_{out})^2,$$

де $length$ - складність тексту процедури (вимірюється однією з метрик складності), fan_{in} - кількість локальних потоків всередину процедури плюс кількість структур даних, fan_{out} - кількість локальних потоків з процедури плюс кількість структур даних, які оновлюються процедурою;

2) інформаційна складність модуля відносно деякої структури даних обчислюється за формулою:

$$I = W \times R + W \times WrRd + WrRd \times R + WrRd \times (WrRd - 1),$$

де W - кількість процедур, які оновлюють структуру даних, R - кількість процедур, які читають інформацію зі структури даних, $WrRd$ - кількість процедур, які читають і оновлюють структуру даних.

Очікувана LOC-оцінка (експертна) - за кожною функцією експерти надають краще, гірше та імовірне значення, тоді очікувана LOC-оцінка (LOC - кількість рядків вихідного коду програми) обчислюється за формулою:

$$LOC_{оч_i} = (LOC_{кращ_i} + LOC_{гірш_i} + 4 \times LOC_{імов_i}) / 6.$$

Показник LOC залежить від мови програмування. Це оціночна і необов'язкова метрика. Вона може призвести до оптимізації кількості рядків коду, а не проекту в цілому. Дозволяє спрогнозувати трудовитрати, час і вартість розробки, а також необхідну кількість програмістів для виконання проекту. Є вільно поширювані інструменти очікуваної LOC-оцінки. LOC-оцінка впливає на оцінку величини змін обсягу коду в часі.

Метрики Холстеда:

1) міра довжини модуля

$$N = n_1 \log_2(n_1) + n_2 \log_2(n_2),$$

де n_1 - кількість різних операторів, n_2 - кількість різних операндів;

2) обсяг модуля як кількість символів для запису всіх операторів і операндів тексту програми:

$$V = N \times \log_2(n_1 + n_2).$$

Метрики Холстеда обчислюються на основі аналізу кількості рядків та синтаксичних елементів вихідного коду програми.

Основу метрик Холстеда складають 4 вимірювані характеристики програми: *NUOprtr* - кількість унікальних операторів програми включно з іменами підпрограм (словник операторів), *NUOprnd* - кількість унікальних операндів програми

(словник операндів), $NOprtr$ - загальна кількість операторів в програмі, $NOprnd$ - загальна кількість операндів програми.

На основі цих характеристик обчсилюються наступні оцінки:

1) словник програми:

$$HPVoc = NUOprtr + NUOprnd;$$

2) довжина програми (впливає на оцінку величини змін обсягу коду в часі):

$$HPLen = NOprtr + NOprnd;$$

3) обсяг програми (впливає на оцінку величини змін обсягу коду в часі):

$$HPVol = HPLen \times \log_2 HPVoc;$$

4) складність програми:

$$HDiff = (NUOprtr / 2) \times (NOprnd / NUOprnd);$$

5) оцінка зусиль програміста при розробці програми (вказує, наскільки ефективна праця розробника):

$$HEff = HDiff \times HPVol$$

Використовується для визначення складності реалізації певної вимоги, функції, модуля, частини проекту. Впливає на точність прогнозів оцінки трудомісткості та на розуміння того, наскільки інтелектуально-витратною для розробника буде та чи інша функція.

Метрика Маккейба - цикломатична складність. Один з найбільш розповсюджених показників оцінки складності програмних проектів. Цикломатичне число Маккейба показує необхідну кількість проходів для покриття всіх контурів сильнозв'язаного графу керування програми, тобто кількість тестових прогонів програми, необхідних для проведення вичерпного тестування. Дозволяє провести оцінку трудомісткості реалізації окремих елементів програмного проекту, оцінку трудомісткості тестування програмного проекту, скоригувати загальні показники оцінки тривалості та вартості проекту, оцінити зв'язані ризики і прийняти необхідні управлінські рішення.

Метрика Маккейба обчислюється для оцінки складності ПС, виходячи з топології внутрішніх зв'язків. Обчислення метрики

проводяться на основі графу керування програми (оргграф, в якому обчислювальні оператори або вирази представляються вершинами, а передачі керування між вершинами представляються дугами). Спрощена оцінка цикломатичної складності $V(G) = E - N + 2$, де E - кількість дуг, а N - кількість вершин в керуючому графі ПЗ (логічні оператори не враховуються).

Автоматизоване обчислення метрики Маккейба зводиться до підрахунку кількості логічних операторів та можливої кількості шляхів виконання програми.

Метрика Маккейба може бути обчислена для модуля, методу та інших структурних одиниць програмного проекту. Вона впливає на оцінку змін складності в часі.

Прогнозована кількість операторів програми:

$$N_{прогн} = NF \times N_{од},$$

де NF - кількість функцій або вимог в специфікації програми, $N_{од}$ - одиничне значення кількості операторів (середня кількість операторів які доводяться на одну функцію або вимогу).

Прогнозована оцінка складності інтерфейсів компонентів ПЗ:

$$C = \frac{NI}{NF \times NI_{од} \times K_{скл}},$$

де NI - загальна кількість змінних, які передаються по інтерфейсах між компонентами програми, $NI_{од}$ - одиничне значення кількості змінних, які передаються по інтерфейсах між компонентами (середня кількість змінних, які передаються по інтерфейсах, що доводяться на одну функцію або вимогу); обирають на основі аналізу статистичних даних; $K_{скл}$ - коефіцієнт складності розроблюваної програми, який враховує ріст одиничної складності програми (складності, що доводиться на одну функцію або вимогу специфікації) в порівнянні з середнім показником складності; його слід обирати з врахуванням статистичних даних та характеристик розробленої ПЗ.

Для оцінки складності програми використовуються й інші метрики: метрика Шнадевида - кількість шляхів в графі керування; метрика Майєрса - інтервальна міра; метрика Хансена - пара (цикломатична кількість, кількість операторів); метрика Чена - топологічна міра; метрика Вудворда - кількість вузлів передач управління; метрика Кулика - кількість найпростіших циклів; метрика Хура -цикломатична кількість мереж Петрі; метрики Вітворфа, Зулевського - міра складності потоку керування, міра складності потоку даних; метрика Петерсона - кількість багатовходових циклів; метрики Харрісона, Мейджела - функційна кількість, функційне відношення, регулярні вирази; метрика Пивоварського - модифікована цкломатична міра складності; метрика Пратта - тестуюча міра; метрика Кантоне - характеристичні числа поліномів графу програми; метрика Схуттса, Моханті - ентропійні міри; метрика Коллофело - міра логічної стабільності програм; метрика Зольновського, Сімонса, Тейєра - зважена сума різних індикаторів; метрика Берлінгера - інформаційна міра; метрика Шумана - складність з позиції статистичної теорії мови; метрика Янгера - логічна складність з врахуванням історії обчислень; метрика Тая - покращення метрики Маккейба; метрика Кокола - комплексна метрика, заснована на більш простих; метрики зв'язності (зчеплення) - ступінь залежності кожного модуля від кожного з інших модулів за даними, за зразком, за управлінням, за зовнішніми посиланнями, за загальною областю, за змістом (кількісний показник - ступінь зчеплення).

Розглянемо тепер основні *метрики якості* ПЗ.

Метрики зв'язності (зв'язність (cohesion) - внутрішня характеристика програмного модуля) - залежать від типу модуля або проекту. Правила визначення рівня зв'язності модуля та сили зв'язності (СЗ):

- якщо модуль - одинична проблемно-орієнтована функція, то рівень зв'язності - функційний (частини модуля разом реалізують одну функцію) і СЗ=10;
- якщо дії всередині модуля зв'язані даними, і порядок дій всередині модуля важливий, то рівень зв'язності - інформаційний

(вихідні дані однієї частини використовуються як вхідні дані в іншій частині модуля) і СЗ=9;

- якщо дії всередині модуля зв'язані даними, і порядок дій всередині модуля не має жодного значення, то рівень зв'язності - комунікативний (частини модуля зв'язані даними - працюють з однією структурою даних) і СЗ=7;
- якщо дії всередині модуля зв'язані потоком керування, і порядок дій всередині модуля важливий, то рівень зв'язності - процедурний (частини модуля зв'язані порядком виконуваних ним дій, які реалізують деякий сценарій поведінки) і СЗ=5;
- якщо дії всередині модуля зв'язані потоком керування, і порядок дій всередині модуля не має жодного значення, то рівень зв'язності - часовий (частини модуля не зв'язані, але необхідні в один і той же момент роботи системи) і СЗ=3; недоліком такого типу зв'язності є сильний взаємний зв'язок з іншими модулями, через що проект стає дуже чутливим до внесення змін;
- якщо дії всередині модуля жодним чином не зв'язані, але належать до однієї категорії, то рівень зв'язності - логічний (частини модуля об'єднані за принципом функційної подібності) і СЗ=1; недоліками логічної зв'язності є складне спряження та велика ймовірність внесення помилок при зміні спряження заради однієї з функцій;
- якщо дії всередині модуля жодним чином не зв'язані, а також не належать до однієї категорії, то рівень зв'язності - за співпаданням (в модулі відсутні явно виражені внутрішні зв'язки) і СЗ=0.

Можливі більш складні випадки, коли з модулем асоціюються декілька рівнів зв'язності. В такому випадку, враховуючи статистичну та експертну інформацію про тип та область застосування розроблюваного ПЗ, модулю присвоюють найсильніший рівень зв'язності або найслабший рівень зв'язності.

Чим вище зв'язність модуля, тим кращий результат проектування, тобто чим "чорніша" скринька проекту (захисна оболонка модуля), тим менше "важелів керування" на ній знаходиться і тим вона простіша. Такі типи зв'язності, як зв'язність за співпаданням, логічна зв'язність та часова зв'язність - результат

невірного планування архітектури, а процедурний тип зв'язності - результат недбалого планування архітектури додатку.

Метрики зчеплення (зчеплення (coupling) - зовнішня характеристика модуля, яку бажано і слід зменшувати) - міра взаємозалежності модулів за даними. Кількісно зчеплення вимірюється ступенем зчеплення (СЗч). Виділимо наступні типи зчеплення:

- зчеплення за даними (СЗч=1) - один модуль викликає інший модуль, всі вхідні і вихідні параметри модуля, що викликається, - прості елементи даних;
- зчеплення за зразком (СЗч=3) - в якості вхідних і вихідних параметрів використовуються структури даних;
- зчеплення за керуванням (СЗч=4) - один модуль явно керує функціюванням іншого модуля за допомогою прапорців або перемикачів і надсилає йому керуючі дані;
- зчеплення за зовнішніми посиланнями (СЗч=5) - модулі посилаються на один і той же глобальний елемент даних;
- зчеплення за спільною областю (СЗч=7) - модулі поділяють одну й ту ж глобальну структуру даних;
- зчеплення за вмістом (СЗч=9) - один модуль прямо посилається на вміст іншого модуля не через його точку входу.

Метрика звертання до глобальних змінних - пара (модуль, глобальна змінна) – пара (p,r), де p - модуль, який має доступ до глобальної змінної r. В залежності від наявності реального звертання до змінної r формуються 2 типи пар (p,r) - фактичні і можливі. Характеристика Aup показує, скільки разів модуль дійсно одержить доступ до глобальної змінної, а характеристика Pup - скільки разів він міг би одержати такий доступ. Тоді наближену ймовірність посилання довільного модуля на довільну глобальну змінну можна обчислити як відношення: $Rup = \frac{Aup}{Pup}$.

Чим вище така ймовірність, тим вище ймовірність "несанкціонованої" зміни певної глобальної змінної, що призводить до ускладнення модифікації програми.

Час модифікації моделей - метрика процесу розробки ПЗ.

Кількість знайдених помилок при інспектуванні моделей та прототипів підсистем, модулів, функцій, вимог та густота помилок (кількість помилок на одну підсистему, модуль, функцію, вимогу) - вказує на проблемну підсистему, модуль, функцію, вимогу. Це метрика процесу розробки ПЗ.

Загальний час розроблення ПЗ - метрика процесу розробки ПЗ, вимірюється у робочих днях.

Час виконання робіт процесу проектування - метрика процесу розробки ПЗ, вимірюється у робочих днях.

Очікувана вартість розроблення ПЗ:

$$ВАРТІСТЬ_i = LOC_{оч_i} \times ВАРТІСТЬ_{рядка_i}.$$

Вартість рядка є константою і не змінюється від реалізації до реалізації; вимірюється у гривнях або доларах США.

Прогнозована вартість перевірки якості - метрика процесу розробки ПЗ; вимірюється у гривнях або доларах США.

Прогнозована продуктивність розроблення ПЗ - вимірюється у хвилинах на один рядок коду.

Прогнозовані витрати на реалізацію програмного коду - вимірюється у гривнях або доларах США.

Прогнозований функційний розмір FP - вимірює суть можливостей майбутньої програми. Метод визначення функційного розміру опишемо покроково наступним чином:

- виділити функції додатку;
- для кожної потенційної виділеної функції слід порахувати кількість зовнішніх входів EI, які по-різному впливають на виконувану функцію; кількість зовнішніх виходів EO, для істотно різних алгоритмів і нетривіальної функційності; кількість зовнішніх запитів EIN; кількість внутрішніх логічних файлів або унікальних логічних груп даних користувача ILF; кількість зовнішніх логічних файлів або унікальних логічних груп користувацьких даних ELF;
- кожен з визначених на попередньому кроці факторів множиться на коефіцієнт, який визначається складністю даного фактору в програмному проекті (EI - 3 або 4 або 6 (від простого до

складного), *EO* - 4 або 5 або 7, *EIN* - 3 або 4 або 6, *ILF* - 7 або 10 або 15, *ELF* - 5 або 7 або 10). Ці добутки додаються за кожною функцією;

- визначити ваги для 14 загальних характеристик проекту (від 0 до 5); ваги вказуються в формі діапазонів, що відображає невпевненість відносно функцій; призначення ваг вимагає певного досвіду у використанні методу функційного розміру;
- обчислити уточнений функційний розмір за формулою:

$$Уточн.функц.розмір = Наближений\_функц\_розмір \times [0.65 + 0.01 \cdot (Сума\_загальних\_характеристик]$$

Якщо до проекту не висувається жодних спеціальних вимог (всі загальні характеристики дорівнюють 0), то неуточнений функційний розмір слід зменшити на 35%, інакше слід збільшити на 1% на кожну одиницю значень загальних характеристик.

Функційний розмір використовується як відносна метрика для порівняння з попередніми проектами, за його допомогою можна обчислити кількість рядків коду, що дозволяє визначити загальну трудомісткість та терміни проекту. Є вільно поширювані інструменти для обчислення функційного розміру.

Прогнозована оцінка трудовитрат за моделлю Боема - трудовитрати на розробку програмних додатків зростають швидше, ніж розмір додатків. Для представлення даного співвідношення використовується експоненційна функція зі значенням показника, близьким до *1.12*: $Трудовитрати = a \times KLOC^b$, де *KLOC* - кількість тисяч рядків коду, a, b - коефіцієнти COCOMO. Для органічного (самостійного) програмного проекту: $a = 2{,}4;\ b = 1{,}05$. Для вбудованих програмних проектів (інтеграція апаратного та програмного забезпечення): $a = 3{,}6;\ b = 1{,}20$. Для проміжних програмних проектів (не органічні, але й не жорстко вбудовані): $a = 3{,}0;\ b = 1{,}12$.

Прогнозована оцінка тривалості проекту за моделлю Боема - тривалість проекту за моделлю Боема зростає експоненційно разом з докладеними до проекту зусиллями. Однак в

даному випадку значення експоненти менше 1 і складає близько *0.35*:

$$Тривалість = c \times Трудовитрати^d = c \times (a \times KLOC^b)^d = \\ = c \times a^d \times KLOC^{b \times d},$$

де c, d - коефіцієнти COCOMO. Для органічного (самостійного) програмного проекту: $c = 2{,}5;\ d = 0{,}38$. Для вбудованих програмних проектів (інтеграція апаратного та програмного забезпечення): $c = 2{,}5;\ d = 0{,}32$. Для проміжних програмних проектів (не органічні, але й не жорстко вбудовані): $c = 2{,}5;\ d = 0{,}35$.

5. Вибір та розрахунок метрик, придатних до використання на етапі проектування ПЗ

На етапі проектування важливо закласти цілу низку вимог по складності та якості: вимоги до структури програмного забезпечення (ПЗ); вимоги до навігації по ПЗ; вимоги до дизайну інтерфейсів користувача; вимоги до мультимедіа-компонентів ПЗ; вимоги по зручності (usability); технічні вимоги.

На етапі проектування формується відповідь на питання "Яким чином програмна система буде реалізовувати висунуті до неї вимоги?".

Інформаційні потоки етапу проектування ПЗ: вимоги до ПЗ, представлені інформаційною, функційною та поведінковою моделями аналізу. Інформаційна модель описує інформацію, які повинна обробляти ПЗ на думку замовника. Функційна модель визначає перелік функцій обробки інформації та перелік модулів програмної системи. Поведінкова модель фіксує бажану динаміку системи (режими її роботи). На виході етапу проектування - розробка даних, розробка архітектури та процедурна розробка ПЗ.

Приймаючи до уваги результати аналізу моделей, методів та метрик складності ПЗ, а також інформаційних потоків етапу проектування ПЗ, можна зробити висновок, що на етапі проектування можна використати наступні метрики складності, які матимуть точні або прогнозовані значення (таблиця 16.1).

Таблиця 16.1 - Метрики складності етапу проектування ПЗ з точними та прогнозованими значеннями

№	Метрики складності етапу проектування з точними значеннями	Метрики складності етапу проектування з прогнозованими значеннями
1	Метрика Чепіна	Очікувана LOC-оцінка
2	Метрика Джилба (абсолютна модульна складність програми)	Метрика Холстеда (складність програми)
3	Метрика Мак-Клура (загальна складність програмної системи)	Метрика Маккейба (цикломатична складність ПЗ в цілому)
4	Метрика Кафура (інформаційна складність модуля)	Метрика Джилба (відносна логічна складність програми)
5		Прогнозована кількість операторів програми
6		Прогнозована оцінка складності інтерфейсів ПЗ

Отже, *на етапі проектування можна використати 10 основних метрик складності, причому 4 з них матимуть точні значення, а інші 6 метрик - прогнозовані значення*.

Приймаючи до уваги результати вище виконаного аналізу моделей, методів та метрик якості ПЗ, а також інформаційних потоків етапу проектування ПЗ можна зробити висновок, що на етапі проектування можна використати наступні метрики якості, які матимуть точні або прогнозовані значення (таблиця 16.2).

Таблиця 16.2 - Метрики якості етапу проектування ПЗ з точними та прогнозованими значеннями

№	Метрики якості етапу проектування з точними значеннями	Метрики якості етапу проектування з прогнозованими значеннями
1	Метрика зв'язності	Загальний час розроблення ПЗ
2	Метрика зчеплення	Час етапу проектування
3	Метрика звертання до	Продуктивність розроблення ПЗ

	глобальних змінних	
4	Час модифікації моделей	Вартість перевірки якості ПЗ
5	Кількість виявлених помилок при інспектуванні моделей та прототипів	Вартість розроблення ПЗ
6		Вартість реалізації коду ПЗ
7		Функційний розмір (FP)
8		Оцінка трудовитрат за моделлю Боема
9		Оцінка тривалості проекту за моделлю Боема

Отже, *на етапі проектування можна використати 14 основних метрик якості ПЗ, причому 5 з них матимуть точні значення, а інші 9 метрик - прогнозовані значення*.

Приклад 16.1. Розрахувати максимальні значення метрик складності ПЗ при використанні наступних обмежень: 1) у кожному модулі ПЗ є максимум 100 змінних для розрахунків і виведення, 100 модифікованих або створених в модулі змінних, 100 керуючих змінних, 100 невикористовуваних в модулі ("паразитних") змінних - всього 400 змінних; 2) ПЗ має не більше 50 модулів; 3) кожен модуль ПЗ пов'язаний з іншими 49 модулями одним зв'язком; 4) в кожному модулі є не більше 50 процедур, які оновлюють структуру даних, не більше 50 процедур, які читають інформацію зі структури даних, та не більше 50 процедур, які читають і оновлюють структуру даних; 5) ПЗ містить не більше 50000 рядків вихідного коду; 6) кількість унікальних операторів програми включно з іменами підпрограм (словник операторів) не перевищує 25000 (кожен другий рядок коду - це унікальний оператор), загальна кількість операндів програми не перевищує 50000 (в кожному рядку коду є один операнд), кількість унікальних операндів програми (словник операндів) не перевищує 400; 7) кожен рядок програми - це логічний оператор або оператор циклу; 8) кожна змінна модуля передається по його інтерфейсу.

Метрика Чепіна для одного модуля матиме максимальне значення $Q_m = 100 + 2 \cdot 100 + 3 \cdot 100 + 0{,}5 \cdot 100 = 650$ згідно формули, наведеної вище. Максимальне значення метрики Чепіна для всього ПЗ становить $Q = 50 \cdot 650 = 32500.$

Максимальна кількість міжмодульних зв'язків (абсолютна модульна складність програми) за метрикою Джилба становитиме 50*49=2450. Максимальна абсолютна логічна складність програми становить 50000, а максимальна *відносна логічна складність програми* в такому разі становить $CL = \frac{50000}{50000} = 1.$

Враховуючи вищевикладені припущення щодо кількості модулів та кількості міжмодульних зв'язків, функція складності кожного з модулів для *метрики Мак-Клура* матиме значення:

$$M(P_1) = 0 \cdot 49 + 49 \cdot 49 = 2401$$

$$M(P_2) = 1 \cdot 49 + 48 \cdot 49 = 2401$$

$$M(P_3) = 2 \cdot 49 + 47 \cdot 49 = 2401$$

.....

$$M(P_{50}) = 49 \cdot 49 + 0 \cdot 49 = 2401$$

Тоді максимальне значення загальної складності програмної системи (метрика Мак-Клура) становить $M(P) = 50 * 2401 = 120050.$

За *метрикою Кафура* одержимо максимальне значення інформаційної складності одного модуля $I_m = 50 \cdot 50 + 50 \cdot 50 + 50 \cdot 50 + 50 \cdot (50 - 1) = 9950.$ Максимальне значення інформаційної складності проекту становить $I = 50 * 9950 = 497500.$

Максимальне значення *очікуваної LOC-оцінки* становить 50000 рядків коду, враховуючи вищевикладені припущення.

Максимальна складність програми за метрикою Холстеда згідно формули, наведеної вище, становить $HDiff = \frac{25000}{2} \cdot \frac{50000}{400} = 1562500.$

Максимальна *цикломатична складність Маккейба* за формулою, наведеною вище: $V(G) = 2450 - 50 + 2 = 2402.$

Якщо кожен рядок коду - це один оператор, тоді максимальна *прогнозована кількість операторів програми* становить 50000.

Враховуючи припущення щодо кількості змінних в модулі, обчислимо *максимальну прогнозовану оцінку складності інтерфейсів ПЗ* за формулою, наведеною вище: $C = \frac{20000}{50 \cdot 400 \cdot K_{скл}} = \frac{1}{K_{скл}}$. Оскільки $K_{скл} = 1 + \sum_{i=1}^{n} K_i$, де K_i - приріст складності розроблюваної програми за *i*-ю характеристикою, n - кількість врахованих характеристик, то максимальна прогнозована оцінка складності інтерфейсів ПЗ становить 1.

Приклад 16.2. Розрахувати максимальні значення метрик якості ПЗ при використанні наступних обмежень: 1) кожен модуль реально одержує доступ до глобальної змінної стільки разів, скільки може одержати такий доступ; 2) ПЗ містить не більше 50000 рядків вихідного коду; 3) тип проекту не задано; 4) постановка задачі займає 10% загального часу розроблення ПЗ і проектної вартості; проектування - 35% загального часу і проектної вартості; програмування - 35% загального часу і проектної вартості; тестування, налагодження і перевірка якості - 20% загального часу і проектної вартості; 5) 25% часу проектування займає побудова та модифікація моделей; 6) максимальна кількість помилок моделей та прототипів одного модуля не повинна перевищувати 100; 7) ПЗ має не більше 50 модулів; 8) 50% часу та вартості, відведених на тестування, налагодження і перевірку якості, займає перевірка якості ПЗ; 9) кількість зовнішніх входів кожного модуля - 49, кількість зовнішніх виходів кожного модуля - 49, кількість зовнішніх запитів до кожного модуля - 49; 10) кожен модуль має максимум 50 внутрішніх логічних файлів та використовує 50 зовнішніх логічних файлів.

Максимальне значення *метрики зв'язності* $СЗ = 10$.

Максимальне значення *метрики зчеплення* $СЗ_ч = 9$.

Для визначення максимального значення *метрики звертання до глобальних змінних* використаємо формулу, наведену вище, тоді максимальне значення імовірності посилання довільного модуля на довільну глобальну змінну $Rup = 1$.

Для обчислення максимального часу модифікації моделей потрібно спочатку прорахувати максимальну прогнозовану тривалість проекту за моделлю Боема. Для визначення коефіцієнтів COCOMO слід визначити, з яким типом проекту маємо справу (самостійним; жорстко вбудованим; не самостійним, але й не жорстко вбудованим). Щоб не накладати обмеження на тип проекту, для розрахунку максимальних значень оцінки трудовитрат та тривалості проекту оберемо максимальні значення коефіцієнтів COCOMO. Тоді *максимальна прогнозована оцінка трудовитрат за моделлю Боема*: $Трудовитрати = 3{,}6 \cdot 50^{1{,}20} = 394$ (людиномісяців), а *максимальна прогнозована оцінка тривалості проекту за моделлю Боема*: $Тривалість = 2{,}5 \cdot 394^{0{,}38} = 24$ (місяці) = 520 (робочих днів).

Якщо прийняти, що в цілому на розроблення програмного забезпечення на 50000 рядків коду - від постановки задачі до налагодження - потрібен максимальний час 520 днів, тоді на постановку задачі потрібно 52 дні (10% часу), на проектування - 182 дні (35% часу), на програмування - 182 дні (35% часу) та на тестування, налагодження та перевірку якості - 104 дні (20% часу). Якщо взяти до уваги, що під час проектування приблизно 25% часу забирають побудова та модифікація моделей, то максимальне значення *часу модифікації моделей* - 46 днів.

Максимальна *загальна кількість знайдених помилок при інспектуванні моделей та прототипів 50 модулів* складає 5000.

Максимальним значенням *прогнозованого загального часу розроблення ПЗ* вважатимемо 520 робочих днів, а максимальним значенням *часу виконання робіт в процесі проектування ПЗ* вважатимемо 182 робочих дні.

За статистикою *максимальною очікуваною вартістю розроблення ПЗ* в доларах США вважають кількість рядків вихідного коду, поділену навпіл. З врахуванням вищенакладених обмежень маємо 25000 доларів США, що приблизно становить 200000 грн.

На тестування, налагодження та перевірку якості відводиться відповідно 40000 грн. Максимальне значення *прогнозованої вартості перевірки якості ПЗ* - 20000 грн.

Максимальне значення *прогнозованих витрат на реалізацію програмного коду* становить 70000 грн.

Продуктивність праці може вимірюватись кількістю часу, витраченого на одиницю продукції. Так, для розроблення ПЗ продуктивністю вважатимемо кількість часу, витрачену на 1 рядок коду. Тоді, враховуючи вищенаведені розрахунки тривалості проекту та обмеження щодо кількості рядків коду, максимальне значення *прогнозованої продуктивності розроблення ПЗ* при 8-годинному робочому дні становить $P = \frac{520 \cdot 8 \cdot 60}{50000} \approx 5$ хвилин.

Оскільки розраховуємо максимальне значення функційного розміру, то значення коефіцієнтів складності кожного фактору в програмному проекті обираємо максимальні. Тоді максимальне значення наближеного функційного розміру згідно формули складає: $FP_{набл} = 49 \cdot 6 + 49 \cdot 7 + 49 \cdot 6 + 50 \cdot 15 + 50 \cdot 10 = 2181.$ Для одержання максимального значення уточненого *функційного розміру* за формулою, наведеною вище, оберемо ваги для 14 загальних характеристик проекту рівними 5 (максимальне значення), тоді: $FP = 2181 \cdot (0{,}65 + 0{,}01 \cdot (14 \cdot 5)) = 2945.$

6. Дослідження результатів метричного аналізу

На основі аналізу метрик можна зробити висновки про рівень якості сирцевого коду: обчислити ймовірності виникнення помилок в окремих модулях, ступені зв'язності модулів та інші параметри. Як зазначалось раніше, однією з проблем аналізу метрик є складність інтерпретації обчислених величин.

Середні значення метрик можуть значно відрізнятись для проектів різного роду. Наприклад, значення цикломатичної складності, нормальне для бібліотеки обчислювальних алгоритмів, невиправдано високе для клієнтського додатку.

Статистичний аналіз метричної інформації передбачає:

1) аналіз метрик за релізами: накопичення статистичної інформації метрик складності і якості ПЗ служить основою для управління складністю і якістю ПЗ в наступних проектах. Так, метрики довжини і обсягу програми дають інформацію про збільшення чи зменшення обсягу програми в часі. Метрика цикломатичної складності показує, чи зростає складність від релізу до релізу. Метрика кількості рядків на реалізацію вимоги попереджає про виявлення збільшення кількості рядків під час виконання типового запиту чи під час реалізації типової вимоги. До того ж ця метрика дозволяє програмісту або менеджеру проекту спрогнозувати кількість рядків коду при використанні типових вимог і функцій. Вони використовуються для прогнозу складності на ранніх етапах на основі статистики. Глибокий аналіз змін по релізах інших кількісних метрик (кількість функцій, класів, файлів) дає можливість виявити вузьке місце в програмі - блок коду, що інтенисвно змінюється - як потенційне місце виникнення помилок.

2) аналіз метрик за замовниками: якщо виділяються окремі гілки або функції програми для конкретного замовника. Потрібно зберігіати інформацію про всі зміни коду для всіх замовників.

3) вибірка проектних даних за певними критеріями - проектами, програмами, замовниками і т.і.

Для визначення задовільності чи незадовільності досягнутого рівня складності ПЗ використовується відношення розрахункового значення метрики до базового (статистичного) значення метрики, взятого зі статистичних даних попередніх проектів. Якщо таке відношення близьке до 1 або менше 1, то можна зробити висновок про задовільний рівень складності ПЗ за аналізованою метрикою. При значенні відношення, яке значно перевищує 1, можливо, слід виконати попередні роботи заново. При цьому слід враховувати змістовний вміст незадовільної

метрики, щоб коригувати відповідні аспекти програмного продукту.

Для визначення впливу конкретних значень окремих метрик на загальну складність ПЗ виконується *інтегральна оцінка складності*. Можливі 2 варіанти інтегральної оцінки: 1) інтегральну відносну складність ПЗ можна визначити як середнє арифметичне відносних результатів (відношень розрахункового значення метрики до базового) всіх метрик, якщо відсутня апріорна інформація про вплив результатів конкретної метрики на загальну складність проміжного або кінцевого продукту; 2) інтегральна складність ПЗ визначається як середньозважена сума одержаних значень метрик (сума ваг всіх метрик дорівнює 1), якщо наявна статистична інформація про сутпінь впливу значень конкретних метрик на складність.

При статистичному аналізі метричної інформації основною проблемою є неможливість опрацювання метрик для принципово нового проекту.

Важливість оцінювання якості та складності ПЗ збільшує інтерес до нових методів, які використовуються в побудові моделей складності та якості ПЗ для прогнозування атрибутів складності та якості. Одним з таких методів є нейромережний метод опрацювання метричної інформації. У одній з галузевих публікацій показано використання ШНМ для прогнозу складності та якості ПЗ на основі об'єктно-орієнтованих метрик. Залежною змінною в такому досліді були роботи по технічному обслуговуванню. Незалежними змінними були головні компоненти восьми об'єктно-орієнтованих метрик. Результат показав, що середня абсолютна величина відносної похибки був 0.265 для моделі з ШНМ. Тому можна зробити висновок, що ШНМ можуть бути корисні в побудові моделі якості ПЗ.

В іншій публікації доведено застосовність байєсових мереж в якості інструменту підтримки для числової оцінки ПЗ в реальному масштабі часу, особливо для додатків з особливими вимогами щодо безпеки. Байєсові мережі можуть також відігравати важливу роль при прийнятті рішення про сертифікацію ПЗ.

Приймаючи до уваги результати аналізу методів оцінки ПЗ, можна зробити висновок, що *перспективним напрямком досліджень є розроблення інтелектуальних систем, які аналізуватимуть і опрацьовуватимуть результати метричного аналізу етапу проектування та надаватимуть оцінку проекту та прогноз про характеристики ПЗ, що розробляється*.

В якості прикладу розглянемо *інтелектуальні метод та систему оцінювання результатів проектування і прогнозування характеристик складності та якості ПЗ на основі опрацювання метричної інформації етапу проектування*.

У попередньому питанні лекції було зроблено висновки, що на етапі проектування можна використати 10 основних метрик складності (4 з них матимуть точні значення, а інші 6 метрик - прогнозовані значення) та 14 основних метрик якості програмного забезпечення (5 метрик якості з точними значеннями, 9 метрик якості з прогнозованими значеннями). Інші метрики є похідними від обраних базових метрик.

В результаті опрацювання вищенаведених 24 метрик складності та якості ПЗ з точними та прогнозованими значеннями потрібно одержати оцінку результатів проектування і прогноз характеристик складності та якості ПЗ, що розробляється на основі того чи іншого проекту.

Нейромережний метод оцінювання результатів проектування і прогнозування характеристик складності та якості програмного забезпечення (НМОП) дозволяє оцінити проект та спрогнозувати характеристики складності та якості розроблюваного ПЗ на основі точних або прогнозованих значень метрик складності та якості ПЗ етапу проектування.

НМОП базується на опрацюванні наступних множин:

1) множина метрик складності етапу проектування з точними значеннями $CMEV = \{cmev_i \mid i = 1..4\}$;

2) множина метрик якості етапу проектування з точними значеннями $QMEV = \{qmev_j \mid j = 1..5\}$;

3) множина метрик складності етапу проектування з прогнозованими значеннями $CMPV = \{cmpv_k \mid k = 1..6\}$;

4) множина метрик якості етапу проектування з прогнозованими значеннями $QMPV = \{qmpv_n \mid n = 1..9\}$.

Результатами опрацювання цих множин є:

1) оцінка складності проекту PCE;

2) оцінка якості проекту PQE;

3) прогноз складності розроблюваного за проектом програмного забезпечення SCP;

4) прогноз якості розроблюваного програмного забезпечення SQP.

Основою для одержання оцінки складності проекту є елементи множини $CMEV$. Основою для одержання оцінки якості проекту є елементи множин $CMEV$ і $QMEV$. Основою для одержання прогнозу складності розроблюваного ПЗ є елементи множини $CMPV$, але враховуються й елементи множин $CMEV$ і $QMEV$. Основою для одержання прогнозу якості розроблюваного ПЗ є елементи множин $CMPV$ і $QMPV$, але враховуються й елементи множин $CMEV$ і $QMEV$.

НМОП складається з наступних *етапів*:

1) підготовка метрик етапу проектування з точними та прогнозованими значеннями для подання їх на вхід ШНМ;

2) перевірка, чи не виходять одержані значення метрик, за межі діапазонів значень входів ШНМ (діапазони вхідних значень ШНМ були визначені у прикладах 16.1 та 16.2);

3) опрацювання значень метрик штучною нейронною мережею;

4) аналіз результатів функціонування ШНМ;

5) формування висновку про складність та якість проекту і розроблюваного ПЗ на основі результатів ШНМ.

Для оцінювання та прогнозування складності та якості ПЗ на основі метричного аналізу слід вирішити задачу визначення взаємозв'язків між значеннями метрик та якістю і складністю проекту та ПЗ. Одним із засобів, який дозволяє узагальнити інформацію та виявити залежності між вхідними і результуючими даними, є штучні нейронні мережі (ШНМ).

Результати метричного аналізу опрацьовуватимемо з використанням ШНМ, яка здійснює апрокисмацію метрик ПЗ етапу проектування та надає оцінку складності та якості проекту та прогноз характеристик складності та якості розроблюваного за проектом ПЗ.

Вхідними даними для ШНМ є множина метрик складності етапу проектування з точними значеннями, множина метрик якості етапу проектування з точними значеннями, множина метрик складності етапу проектування з прогнозованими значеннями, множина метрик якості етапу проектування з прогнозованими значеннями.

Результатом роботи ШНМ є 4 показники: оцінка складності проекту; оцінка якості проекту; прогноз складності розроблюваного за проектом ПЗ; оцінка якості розроблюваного за проектом ПЗ.

Концепція НМОП представлена на рис.16.1.

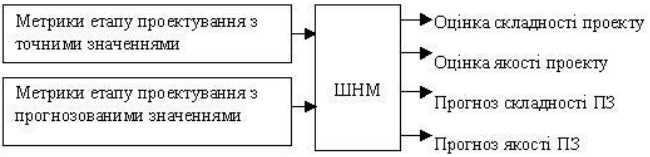


Рис.16.1 - Концепція НМОП

У результаті аналізу відомих архітектур штучних нейронних мереж було обрано архітектуру ШНМ для аналізу метрик етапу проектування ПЗ та прогнозу характеристик якості ПЗ - багатошаровий персептрон. При використанні нейромережі іншого типу для розв’язання цієї задачі її природа буде штучно спотворюватись, в результаті чого результати роботи ШНМ будуть невідповідними.

ШНМ має 9 входів для кількісних значень метрик етапу проектування з точними значеннями та 15 входів для кількісних значень метрик етапу проектування з прогнозованими значеннями. Якщо певна метрика не визначалась, то на відповідний вхід подається -1.

На основі аналізу 4-х одержаних результатів робиться висновок про якість і складність проекту та очікувану якість і складність розроблюваного за проектом програмного забезпечення. Оцінка складності проекту, оцінка якості проекту, прогноз складності розроблюваного ПЗ, прогноз якості розроблюваного ПЗ є значеннями з діапазону [0, 1], де 0 - свідчить про недостатність інформації щодо результатів метричного аналізу, близько 0 - проект або розроблюване програмне забезпечення має високу складність або низьку якість відповідно та 1 - проект або розроблюване програмне забезпечення є відповідно простим або високоякісним.

Вищезазначені залежності результуючих оцінок від вхідних множин метрик враховуються при навчанні нейромережі.

ШНМ архітектури багатошаровий персептрон має 24 нейрони вхідного шару, 14 нейронів апроксимуючого шару, 10 нейронів коригуючого шару та 4 нейрони вихідного шару. Описану ШНМ реалізовано у пакеті Matlab. Структурну схему шарів ШНМ у пакеті Simulink показано на рис.16.2.

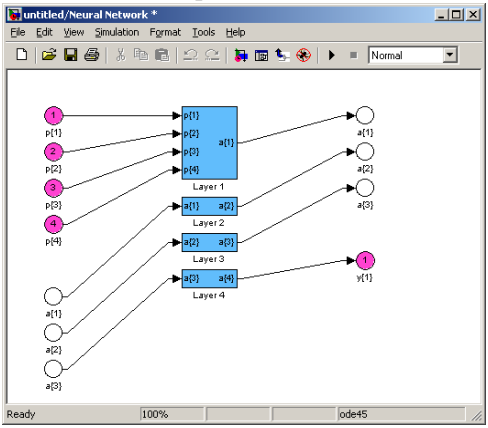


Рис.16.2 - Структурна схема шарів ШНМ у пакеті Simulink

Для навчання одержаної ШНМ послідовність навчальних векторів (навчальна вибірка) задано у вигляді:

`c={[32500;0;0;0] [30875;0;0;0] […];` - навчальні вектори для входу Input1;

[0;0;0.15;0;0] [0;0;0.2;0;0] […]; - навчальні вектори для входу Input2;

[3450;0;0;0;0;0] [5900;0;0;0;0;0] […]; - навчальні вектори для входу Input3;

[0;0;0;0;1.7;0;0;0;0] [0;0;0;0;1.8;0;0;0;0] […]} - навчальні вектори для входу Input4.

Цільовий вектор визначено як: m={[0.05;0.02;0.01;0.01] [0.1;0.04;0.02;0.02] […]}.

Для вибору алгоритму навчання було проведено навчання ШНМ з навчальною вибіркою з 1935 векторів за різними алгоритмами. Дослідження показали, що найменшу похибку навчання одержано при комбінованому критерії якості навчання (msereg); оптимальною кількістю нейронів другого, прихованого, шару є 14 нейронів. В ході дослідження виявлено, що похибка навчання суттєво не відрізняється для різних алгоритмів навчання при використанні однакового критерію якості навчання, тому вибір алгоритму навчання обумовлювався показниками "кількість епох" та "час навчання", за якими кращими є алгоритми навчання OSS, CGB, SCG. Результати дослідження показали, що ШНМ навчається різними алгоритмами за різну кількість епох, але похибка навчання є приблизно однаковою для всіх алгоритмів навчання і у середньому складає приблизно 0,102197. Для тестування ШНМ використовувалась тестова вибірка з 324 векторів.

Для автоматизації оцінювання результатів проектування і прогнозування характеристик складності та якості ПЗ на основі опрацювання метрик етапу проектування з точними та прогнозованими значеннями розроблено *інтелектуальну систему оцінювання і прогнозування складності та якості програмного забезпечення (ІСОП)*.

На вхід ІСОП подаються кількісні значення метрик етапу проектування з точними та прогнозованими значеннями, а результатом роботи є висновки про складність та якість проекту та розроблюваного ПЗ. Структурна схема ІСОП представлена на рис.16.3.

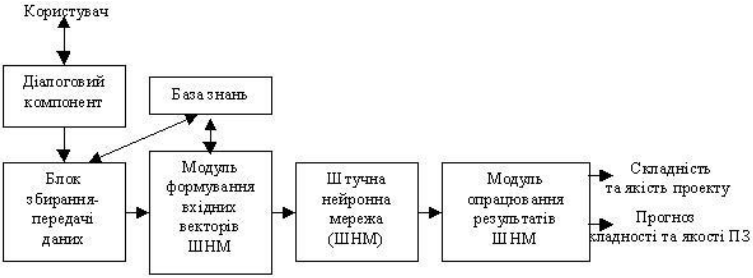


Рис.16.3 – Структурна схема ІСОП

ІСОП складається з наступних *компонентів*:

1) діалоговий компонент;
2) блок збирання-передачі даних;
3) база знань;
4) модуль формування вхідних векторів для ШНМ;
5) штучна нейронна мережа;
6) модуль опрацювання результатів ШНМ.

Діалоговий компонент візуалізує роботу блоку збирання-передачі даних, відображає роботу системи та видає користувачу повідомлення в зрозумілій для нього формі.

Блок збирання-передачі даних зчитує інформацію користувача щодо кількісних значень точних та прогнозованих метрик етапу проектування ПЗ, зберігає одержану інформацію в базі знань та передає її у модуль формування вхідних векторів ШНМ.

База знань містить кількісні значення точних та прогнозованих метрик етапу проектування ПЗ, вхідні вектори ШНМ та правила опрацювання результатів роботи ШНМ.

Штучна нейронна мережа здійснює апрокисмацію метрик ПЗ етапу проектування та надає кількісну оцінку складності та якості проекту та значення прогнозу характеристик складності та якості розроблюваного ПЗ згідно нейромережного методу оцінювання і прогнозування складності та якості ПЗ.

Модуль формування вхідних векторів ШНМ готує значення метрик з бази знань до подачі на входи ШНМ.

На основі 4-х одержаних від ШНМ результатів модуль опрацювання результатів роботи ШНМ робить висновки про якість і складність проекту та очікувану якість і складність розроблюваного програмного забезпечення за наступними правилами:

1) якщо $PCE = 0$, то метрики складності з точними значеннями не визначено;

2) якщо $PCE \to 0$, то проект складний для реалізації та передбачає високу вартість реалізації;

3) якщо $PCE \to 1$, то проект простий для реалізації;

4) якщо $PQE = 0$, то метрики якості з точними значеннями не визначено;

5) якщо $PQE \to 0$, то проект неякісний;

6) якщо $PQE \to 1$, то проект задовольняє вимоги замовника щодо якості;

7) якщо $SCP = 0$, то метрики складності з прогнозованими значеннями не визначено;

8) якщо $SCP \to 0$, то майбутнє ПЗ буде мати суттєву складність;

9) якщо $SCP \to 1$, то майбутнє ПЗ очікується відносно простим;

10) якщо $SQP = 0$, то метрики якості з прогнозованими значеннями не визначено;

11) якщо $SQP \to 0$, то майбутнє ПЗ буде неякісним;

12) якщо $SQP \to 1$, то майбутнє ПЗ очікується високої якості.

Використовуючі такі правила, ІСОП надає оцінку складності та якості проекту і прогноз складності та якості розроблюваного ПЗ, які допомагають замовнику прийняти правильні рішення щодо вибору проекту.

Система ІСОП розроблена мовою Delphi 7.

Перше інтерфейсне вікно системи виглядає наступним чином (рис.16.4). Воно має довідковий характер, оскільки покаузє користувачу всі метрики, які опрацьовуються системою ІСОП.

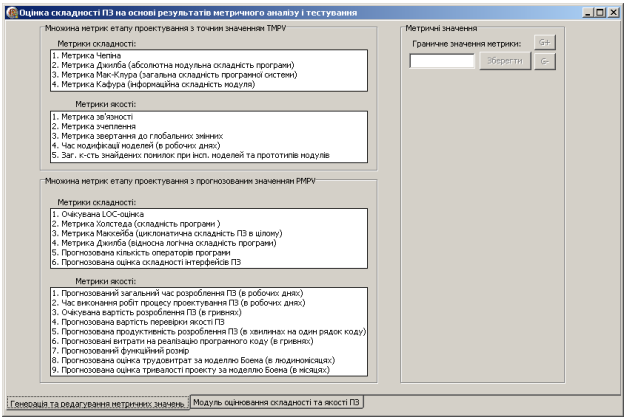


Рис.16.4 - Перше інтерфейсне вікно системи ІСОП (довідкове)

При виборі певної метрики у правій частині вікна відображатимуться граничні значення метрики та попередні значення даної метрики, які були введені при користуванні системою, та значення відповідного вихідного значення системи як реакція саме на цю метрику (рис.16.5). Якщо користувачем вводяться декілька метрик певного типу або ж певна кількість метрик різних типів, то вони корелюють між собою, і вихідне значення системи враховує всі введені значення метрик.

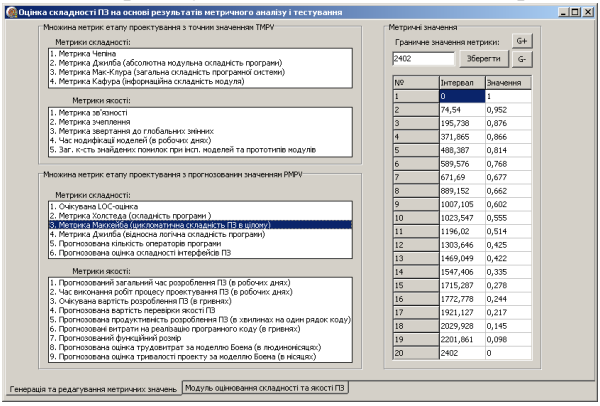


Рис.16.5 - Інтерфейсне вікно системи з інформацією про діапазон обраної метрики та з попередніми введеними значеннями обраної метрики

Для введення значень метрик та одержання висновків системи слід перейти на вкладку "Редагування метричних значень", яка на початку роботи із системою має вигляд, показаний на рис.16.6. В лівій частині вікна потрібно вводити значення метрик, які визначалсиь для проекту, або -1, якщо дана метрика не визначалась. Після введення всіх метрик слід натиснути кнопку "Отримати оцінку" для одержання висновків системи ІСОП.

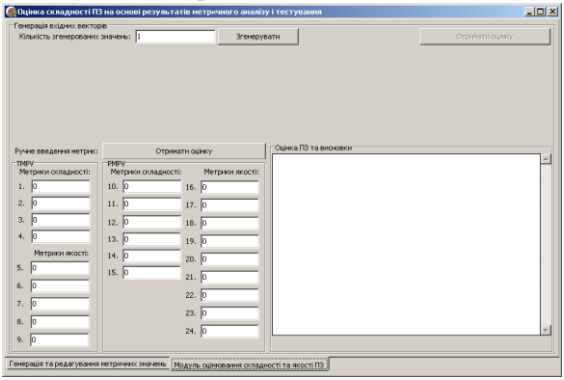


Рис.16.6 - Інтерфейсне вікно системи ІСОП "Редагування метричних значень" до введення значень метрик

У якості вхідних даних ІСОП будемо використовувати результати метричного аналізу 3-х проектів, розроблених софтверною компанією "СТУ-Електронікс" м.Хмельницького (таблиця 16.3).

Таблиця 16.3 - Результати метричного аналізу етапу проектування

№ пр	Метрики складності та якості з точними значеннями на етапі проектування	Метрики складності та якості з прогнозованими значеннями на етапі проектування
1	Метрика Чепіна – 22750 Метрика Джилба (абсолютна) – 1730 Метрика Мак-Клура – 84050 Метрика Кафура – не	Очікувана LOC-оцінка – 35300 Метрика Холстеда – не визначалась Метрика Маккейба – 1680

	визначалась Метрика зв'язності – 3 Метрика зчеплення – 7 Метрика виклику глобальних змінних – не визначалась Час модифікації моделей – 33 Кількість виявлених помилок при інспектуванні моделей та прототипів – не визначалась	Метрика Джилба (відносна) - 0,7 Очікувана кількість операторів програми – 35000 Очікувана оцінка складності інтерфейсів – не визначалась Загальний час розроблення ПЗ – 364 робочих дні Час виконання робіт етапу проектування – 127 робочих дні Очікувана вартість розроблення ПЗ – 140000 гривень Очікувана вартість перевірки якості ПЗ – не визначалась Прогнозована продуктивність розроблення ПЗ – не визначалась Прогнозовані витрати на реалізацію програмного коду – 49000 гривень Очікуваний функційний розмір FP - 2100 Прогнозована оцінка трудовитрат за моделлю Боема – 283 людиномісяців Прогнозована оцінка тривалості проекту за моделлю Боема – не визначалась
2	Метрика Чепіна – 16250 Метрика Джилба (абсолютна) – 1250 Метрика Мак-Клура – 60050 Метрика Кафура – 257000 Метрика зв'язності – 5 Метрика зчеплення – 4 Метрика виклику глобальних	Не визначались

	змінних – 0,5 Час модифікації моделей – 24 Кількість виявлених помилок при інспектуванні моделей та прототипів – 2500	
3	Не визначались	Очікувана LOC-оцінка – 10800 Метрика Холстеда – 312501 Метрика Маккейба – 480 Метрика Джилба (відносна) - 0,2 Кількість операторів програми – 10000 Очікувана оцінка складності інтерфейсів – 0,2 Загальний час розроблення ПЗ – 104 робочих дні Час виконання робіт етапу проектування – 37 робочих днів Очікувана вартість розроблення ПЗ – 40000 гривень Очікувана вартість перевірки якості ПЗ – 4000 гривень Прогнозована продуктивність розроблення ПЗ – 1,1 хвилина на рядок Прогнозовані витрати на реалізацію програмного коду – 14000 гривень Очікуваний функційний розмір FP - 560 Прогнозована оцінка трудовитрат за моделлю Боема – 80 людиномісяців Прогнозована оцінка тривалості проекту за моделлю Боема – 5 місяців

Після одержання даних метричного аналізу блок збирання-передачі даних передає інформацію у модуль формування вхідних векторів ШНМ та зберігає у розділі даних бази знань. Модуль формування вхідних векторів ШНМ створює наступні вектори для входів ШНМ (таблиця 16.4).

Таблиця 16.4 - Приклад вхідних векторів ШНМ

№	Input1	Input2	Input3	Input4
1	{[22750;1730; 84050;-1];	[3;7; -1;33; -1];	[35300;-1; 1680;0.7; 35000;-1];	[364;127;140000; -1;-1;9000;2100; 283;-1]}
2	{[16250;1250; 60050;257000];	[5;4;0.5; 24; 2500];	[-1;-1;-1; -1; -1; -1];	[-1;-1;-1;-1;-1;-1; -1;-1;-1]}
3	{[-1;-1;-1;-1];	[-1;-1;-1; -1;-1];	[10800; 312501; 480; 0.2;10000;0.2];	[104;37;40000; 4000;1.1;14000; 560;80;5]}

ШНМ опрацьовує вхідні вектори та видає результати, на основі яких, згідно правил з бази знань, модуль опрацювання результатів роботи ШНМ робить певні висновки (таблиця 16.5).

Таблиця 16.5 - Результати ШНМ та висновки щодо проаналізованих проектів

№	Значення $Y_1(OSP)$ та висновок ІСОП	Значення $Y_2(OQP)$ та висновок ІСОП	Значення $Y_3(PSPZ)$ та висновок ІСОП	Значення $Y_4(PQPZ)$ та висновок ІСОП
1	0.31	0.30	0.34	0.29
	Проект складний для реалізації	Проект має низьку якість	Майбутнє ПЗ матиме суттєву складність	Майбутнє ПЗ матиме низьку якість
2	0.50	0.51	0	0
	Проект має середню складність	Проект має середню якість	Метрики складності з прогнозованими значеннями не визначались	Метрики якості з прогнозованими значеннями не визначались

3	0	0	0.83	0.80
	Метрики складності з точними значеннями не визначались	Метрики якості з точними значеннями не визначались	Майбутнє ПЗ очікується доволі простим	Майбутнє ПЗ очікується високої якості

Як видно з таблиці 16.3, для проекту №2 не визначались метрики складності та якості з прогнозованими значеннями, тому система не змогла зробити прогноз складності та якості розроблюваного за проектом програмного забезпечення. Для проекту №3 не визначались метрики складності та якості з точними значеннями, тому система не змогла оцінити складність та якість проекту. Дослідження проектів з визначеними усіма групами метрик показують, що оцінки складності та якості проекту і оцінки складності та якості майбутнього ПЗ практично однакові. Згідно отриманих результатів, найпростішим та найякіснішим є проект №3. Програмне забезпечення, розроблене за проектом №3, теж буде достатньо простим та високоякісним.

Приклад опрацювання метрик реалізованою системою ІСОП показано на рис.16.7. В лівій частині вікна введено значення метрик для проекту 1 (таблиця 16.4), у правій частині вікна - сформовано висновки системи (таблиця 16.5).

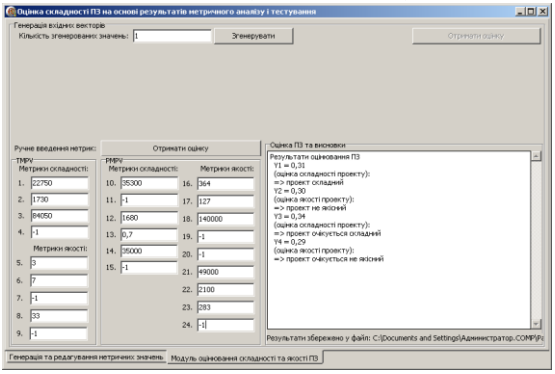


Рис.16.7 - Інтерфейсне вікно з введеними значеннями метрик для проекту 1 та згенерованими системою висновками

Висновки ІСОП дозволяють порівняти між собою різні версії проектів у ситуації, коли вартість і тривалість приблизно однакові.

Наприклад, розглянемо дані щодо 2-х проектів, розроблених софтверною компанією "СТУ-Електронікс" м.Хмельницького (таблиця 16.6).

Таблиця 16.6 - Характеристики варіантів реалізації проекту

№	Характеристики Y_1, Y_2	Характеристики Y_3, Y_4	Вартість	Час розроблення
1	Y_1=0,85 Y_2=0,86	Y_3=0,81 Y_4=0,87	100000 грн	250 робочих днів
2	Y_1=0,22 Y_2=0,25	Y_3=0,28 Y_4=0,27	105000 грн	260 робочих днів

Очевидно, що на основі лише вартості та часу розроблення софтверна організація може прийняти хибний висновок щодо вибору варіанту реалізації проекту, і саме висновки ІСОП допоможуть замовнику зробити вірний вибір проекту.

Запропонована інтелектуальна система оцінювання і прогнозування складності та якості програмного забезпечення дає дані замовнику для вибору проекту необхідного ПЗ та дозволяє порівняти між собою різні версії проекту, тобто дає змогу прийняти мотивоване та обгрунтоване рішення щодо вибору проекту та його реалізації на основі не лише вартісних та часових характеристик, але й з врахуванням характеристик складності та якості проекту і розроблюваного програмного забезпечення.

Проблеми реалізації НМОП та ІСОП:

1) відсутність метричної інформації для нарощування розміру навчальної та тестової вибірок;

2) необхідність деяких диверсних утиліт для порівняння результатів опрацювання метричної інформації для певного проекту;

3) необхідність розроблення метрик оцінки складності розроблюваного ПЗ з точки зору складності чи простоти

супроводу, зручності використання та ефективності методів, обраних для вирішення задачі;

4) відсутність критеріїв вибору з множин метрик етапу проектування з точним та прогнозованим значенням саме тих метрик, які реагують на проект певного типу і відображають особливості такого проекту;

5) відсутність критеріїв, за якими б можна було довести достовірність вибору архітектури ШНМ, достовірність навчання ШНМ та достовірність її функціювання.

7. Поняття та методи статичного аналізу програмного коду

В літературних джерелах та стандартах найчастіше розглядаються такі складові діагностування ПЗ, як верифікація, валідація, атестація, тестування і налагоджування. Інколи частина спеціалістів їх плутає або намагається операціями однієї складової перекривати інші складові, тобто переносить частину операцій складових у процеси, які їм не притаманні. Тому варто уточнити зміст кожної з основних складових процесу діагностування як єдиного цілого і вказати конкретні технологічні операції та зв'язки між цими складовими.

Чітке відокремлення понять "тестування", "верифікація" і "валідація" надає рис.17.1.

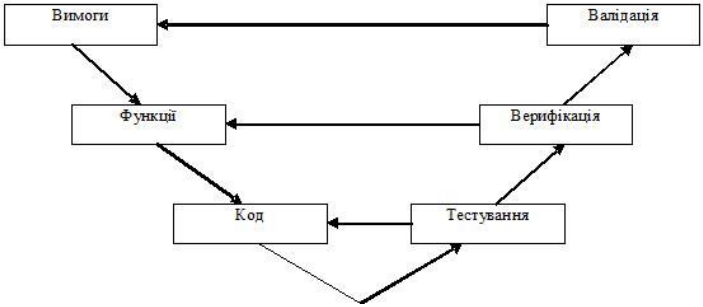


Рис.17.1 - Тестування, верифікація і валідація ПЗ

Найбільш вагомими операціями, що охоплюють всі етапи життєвого циклу ПЗ, є верифікація і валідація. Щодо верифікації і валідації, то ними називають процеси перевірки та аналізу ПЗ, під час яких перевіряється програмне забезпечення на відповідність специфікації та вимогам замовників. Починаються верифікація і валідація ще на етапі аналізу вимог і закінчується на етапі перевірки програмного коду готової програми шляхом тестування.

Верифікація ПЗ - це процес визначення того, чи виконує ПЗ та його компоненти вимоги, висунуті до нього послідовно на різних етапах життєвого циклу розроблюваного ПЗ. Суть верифікації полягає у доведенні того, що поведінка програми відповідає

специфікації на цю програму. Додатковою метою верифікації є виявлення та реєстрація помилок, які внесені під час розроблення або модифікації програми. Верифікація - це процес, спрямований на демонстрацію наявності помилок та умов їх виникнення. Верифікація включає аналіз причин виникнення помилок та наслідків, які викличе їх виправлення, планування процесів пошуку помилок та їх виправлення, оцінювання одержаних результатів.

Верифікація і валідація являють собою різні технологічні операції діагностування. Результатом першої є висновок про те, чи правильно розроблене ПЗ, а другої – чи ПЗ працює правильно. Якщо під час верифікації перевіряється відповідність ПЗ системній специфікації, зокрема функційним та нефункційним вимогам, то під час валідації необхідно впевнитись, що ПЗ відповідає вимогам і очікуванням замовника. Метою валідації ПЗ є доказ того, що в результаті розроблення ПЗ були досягнуті цілі, які планувались.

Співвідношення верифікації та валідації ПЗ представлено на рис.17.2.

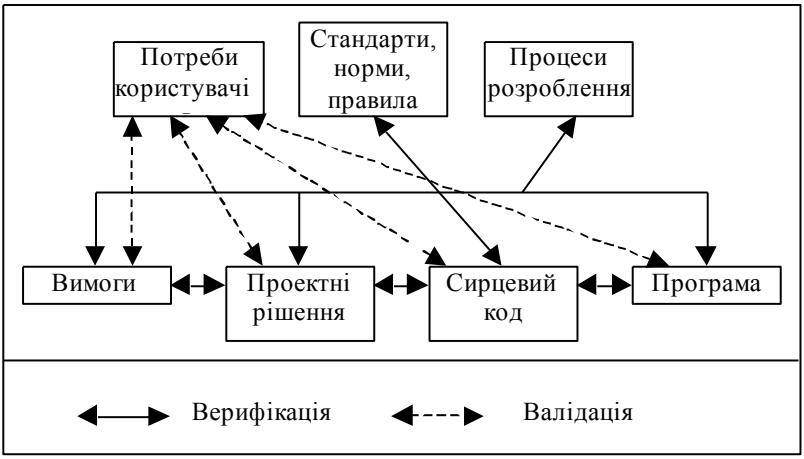


Рис.17.2 - Співвідношення верифікації та валідації ПЗ

Верифікація відповідає на питання: "Чи робимо ми продукт вірно?", а *валідація* - на питання "Чи робимо ми вірний продукт?"

Вимоги повинні бути верифіковані за наступними критеріями:

- відсутність протиріч, виконуваність та тестованість вимог до системи;
- розподіл вимог до системи між об'єктами технічних та програмних засобів та ручних операцій відповідно до проекту;
- відсутність протиріч, виконуваність, тестованість та точність відображення вимог до системи у вимогах до програмних засобів;
- правильність, підтверджена відповідними методами, вимог до програмних засобів по безпеці, захисту та критичності.

Методи верифікації ПЗ поділяються на наступні групи:

1) експертиза різних артефактів життєвого цилку ПЗ;

2) статичний аналіз властивостей артефактів життєвого циклу ПЗ;

3) формальні методи верифікації - дедуктивний аналіз, перевірка моделей, перевірка узгодженості;

4) динамічні методи верифікації - моніторинг, тестування;

5) синтетичні методи - тестування на основі моделей, моніторинг формальних властивостей.

Статичний аналіз (СА) коду - це методологія виявлення помилок в програмному коді, заснована на перегляді коду програмістом (code review), позначеного статичним аналізатором там, де потенційно може знаходитись помилка. Статичний аналіз вихідного коду є одним з методів оцінки якості програмного забезпечення, що застосовується на ранніх етапах розробки і не вимагає повної завершеності циклу розробки.

Для визначення задач статичного аналізу розглянемо статистику виникнення дефектів, надану компанією Coverity, якою було проаналізовано 280 C++ проектів загальним обсягом 60 млн. рядків коду. В результаті такого аналізу було виявлено 38453 дефекти із щільністю 0.25-1 дефект на 1000 рядків коду (рис.17.3).

Задачами статичного аналізу є: оптимізація програм (обчислення константних виразів, виявлення "мертвого" коду, розпаралелювання програми), перетворення програм (переведення на інші платформи, мови та бібліотеки), обфускація/деобфускація

(обфускація - приведення тексту або виконуваного коду програми до вигляду, який зберігає її функційність, але ускладнює аналіз, розуміння алгоритмів роботи та модифікацію при декомпіляції), виявлення помилок.

Тип дефекту	Частота
Розіменовування нульових вказівників	27,81%
Витоки пам'яті	23,34%
«Мертвий» код	9,71%
Небезпечне використання значення, що повертається	9,2%
Використання значень до перевірки	8,35%
Використання об'єкта після вивільнення	5,91%
Переповнення буферу	6,0%
Читання не ініціалізованої змінної	8,41%
Інші	1,27%

Рис.17.3 - Статистика виникнення дефектів

СА включає наступні *типи перевірок*:

- семантичну;
- виділення пам'яті;
- логічних операторів;
- проблем включень;
- безпеки;
- жорстку перевірку типів;
- аналіз метрик.

Сфери застосування статичного аналізу:

- виявлення функціональних помилок у вихідному коді програм;
- рекомендації щодо оформлення вихідного коду;
- перевірка відповідності вихідного коду прийнятим стандартам;
- підрахунок метрик (числових значень деякої властивості ПЗ).

Приклади метрик, які можна отримати на етапі статичного аналізу:

1) кількісні метрики:

- кількість порожніх рядків,

- кількість коментарів,

- відсоток коментарів (відношення числа рядків, що містять коментарі до загальної кількості рядків, виражене у відсотках),

- середнє число рядків для функцій (класів, файлів),

- середнє число рядків, що містять вихідний код для функцій (класів, файлів)

- середнє число рядків для модулів.

2) метрика Холстеда:

- *n1* - число унікальних операторів програми, включаючи символи-роздільники, імена процедур і знаки операцій (словник операторів);

- *n2* - число унікальних операндів програми (словник операндів);

- *N1* - загальне число операторів у програмі;

- *N2* - загальне число операндів в програмі.

Візуалізація метрик представлена на рис.17.4.

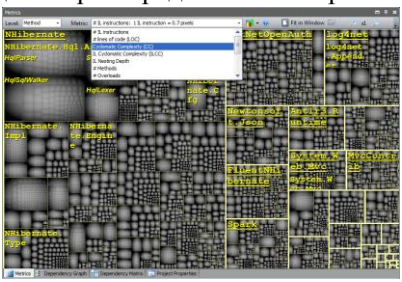


Рис.17.4 - Візуалізація метрик

Переваги статичного аналізу:

1) теоретично дозволяє проаналізувати усі можливі траси виконання;

2) дозволяє проаналізувати всі набори вхідних даних;

3) може бути повністю автоматизований;

4) виявлення «мінливих» помилок (undefined behavior);

5) перевірка фрагментів програмного коду для яких важко розробити перевірочний тест (розрахункові алгоритми, інтерфейс);

6) виявлення помилок набору та однакових конструкцій.

Недоліки статичного аналізу:

1) невисокі результати в діагностиці витоків пам'яті (memory leak) і помилок паралельного виконання програм;

2) хибні спрацювання.

Види статичного аналізу:

1) синтаксичний аналіз (пошук по шаблонах);

2) аналіз потоку даних;

3) аналіз станів об'єктів програми;

4) аналіз потоку керування;

5) аналіз паралельного виконання програми.

Ефективність статичного аналізу залежить від вигляду інформації, що видобувається; властивостей мови програмування; особливостей алгоритму програми та стилю кодування; ступені впливу оточення та зовнішніх впливів; використовуваних алгоритмів аналізу. Ефективність СА - комплексна властивість, що відображає якість одержуваних результатів, ступінь автоматизації процесу аналізу та складність його організації, ресурсомісткість, застосованості для програм різного розміру та класу

Показники ефективності СА:

- повнота результатів;
- точність результатів;
- ресурсоємність;
- можливість автоматизації.

Ефективність СА залежить від:

1) типу отриманої інформації;

2) властивостей мови програмування;

3) особливостей алгоритму програми та стилю кодування;

4) ступеня впливу оточення і зовнішніх впливів;

5) застосованих алгоритмів аналізу.

СА вихідного коду є ефективним засобом підвищення якості ПЗ. Кількість помилок, про які сигналізує СА, не вказує на якість результатів його функціонування.

Згідно наведених визначень, статичний аналіз коду програм відповідає верифікації ПЗ як перевірка відповідності програмного коду різним стандартам кодування. Статичний аналіз перевіряє

також відповідність результатів етапу конструювання програмної системи вимогам та обмеженням, сформульованим раніше.

Статичний аналіз може використовуватися:

1) у складі середовищ розробки для оперативної перевірки програм;

2) у системах Continuous integration для постійного контролю якості коду і своєчасного виявлення дефектів.

Статичний аналіз не замінює тестування, а доповнює його.

8. Засоби статичного аналізу ПЗ

Інструмент для статичного аналізу визначає в тексті програми місця, які містять або потенційно можуть містити помилки, а також ділянки коду, які мають погане форматування. Такі ділянки коду надаються програмісту для вивчення, і він може прийняти рішення про модифікацію даної ділянки програми.

Статичні аналізатори можуть бути як загального призначення, так і спеціалізованими для пошуку певних класів помилок. Статичні аналізатори коду дозволяють виявити велику кількість помилок на самих ранніх етапах розроблення програмного коду. *Основна перевага використання статичних аналізаторів* коду полягає у можливості істотного зниження вартості усунення дефектів у програмі. Для прикладу: виправлення помилки на етапі тестування вартує вдесятеро дорожче, ніж на етапі конструювання (кодування) - рис.17.5.

	Час виявлення дефекту				
Час внесення дефекту	Формулювання вимог	Проектування архітектури	Конструювання (кодування)	Тестування	Після випуску ПЗ
Формулювання вимог	1	3	5-10	10	10-100
Проектування архітектури	-	1	10	15	25-100
Конструювання (кодування)	-	-	1	10	10-25

Статичний аналіз

Рис.17.5 - Середня вартість виправлення дефектів в залежності від часу їх внесення та виявлення

Одним із популярних статичних аналізаторів є аналізатор *PVS-Studio*, який дозволяє перевірити сучасні та ресурсомісткі додатки. Аналізатор дозволяє діагностувати помилки, специфічні для 64-бітних та паралельних рішень, допомогти в оптимізації додатків та підвищити їх безпечність. PVS-Studio - це сучасний аналізатор коду, призначений для розробників Windows-додатків мовою C/C++/C++0x як інструмент виявлення широкого спектру дефектів в коді. Аналізатор "вміє" шукати помилки, пов'язані з використанням WinAPI, STL, OpenMP і т.д., здійснює найпотужніший аналіз для виявлення 64-бітних помилок та проблем міграції. Аналізатор PVS-Studio інтегрується в середовище розроблення Microsoft Visual Studio 2005/2008/2010. Даний аналізатор має досить високу вартість - 3500 євро.

Іншими достатньо відомими та використовуваними статичними аналізаторами є: *Red Lizard Goanna Studio* (від $999 в залежності від компонування); *Gimpel Software PC-Lint* в комплексі з Visual Lint для інтеграції з IDE ($389); *CppCheck* (Freeware). Приклад інтерфейсного вікна CppCheck представлений на рис.17.6.

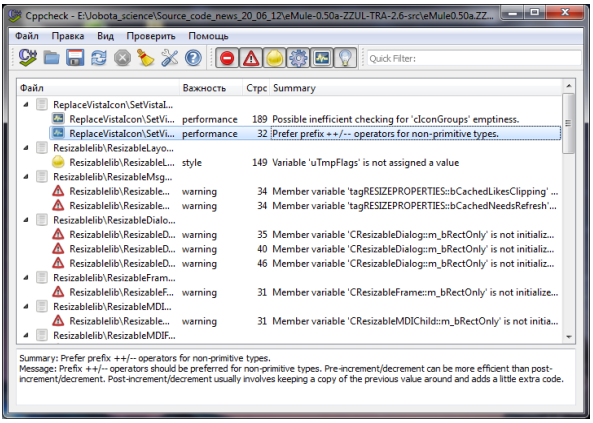


Рис.17.6 - Статичний аналізатор CppCheck

9. Дослідження результатів статичного аналізу

Тестоване за допомогою CppCheck ПЗ представлено на рис.17.7.

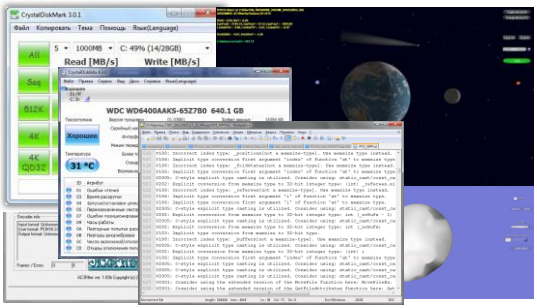


Рис.17.7 - Приклади тестованого ПЗ

Приклади помилок, віднайдених у ПЗ статичним аналізатором CppCheck:

1) if(m_GamePad[iUserIndex].wButtons ||
m_GamePad[iUserIndex].sThumbLX ||
m_GamePad[iUserIndex].sThumbLX ||
m_GamePad[iUserIndex].sThumbRX ||
m_GamePad[iUserIndex].sThumbRY)

В складній умові оператора IF двічі використовується той самий параметр m_GamePad[iUserIndex].sThumbLX, хоча з контексту вихідного коду слідує, що у другому випадку повинен був вказуватись параметр m_GamePad[iUserIndex].sThumbLY

2) else if(DeviceType ==
D3D10_DRIVER_TYPE_SOFTWARE)
wcscpy_s(pstrDeviceStats, 256, L"WARP");
else if(DeviceType ==
D3D10_DRIVER_TYPE_HARDWARE)
wcscpy_s(pstrDeviceStats, 256, L"HARDWARE");
else if(DeviceType ==
D3D10_DRIVER_TYPE_SOFTWARE)
wcscpy_s(pstrDeviceStats, 256, L"SOFTWARE");

Як слідує з фрагмента вихідного коду, розробники припустились помилки набору, відповідно, перша умова в операторі IF ідентична третій, яка в свою чергу ніколи не зможе бути виконаною.

3)

```
uint  t;
…
if(t>=0 ...)
{.}
else  … ;
```

Оскільки змінна t була оголошена, як беззнакова і не може приймати значення, менше нуля, дана умова не має сенсу, а відповідно ELSE-частина виразу ніколи не може бути виконана.

4) CDXUTControl::CDXUTControl(CDXUTDialog* pDialog)

```
{
   m_pDialog = pDialog;
   …
/*дії зm_pDialog в пропущеном коді відсутні/*
   m_pDialog = NULL;
```

Оскільки змінна t була оголошена, як беззнакові і не може приймати значення, менше нуля, дана умова не має сенсу, а відповідно ELSE-частина виразу ніколи не може бути виконаною.

5) DWORD WINAPI Notepad_plus::threadTextPlayer(void *params)

```
{
 ...
 const char *text2display = ...;
 ...
 if (text2display[i] == ' ' && text2display[i] == '.')
 ...
}
```

Умова (text2display [i] == '' && text2display [i] == '.') Ніколи не виконується. Символ не може одночасно бути і пробілом і крапкою.

Результати роботи статичного аналізатору CppCheck представлені на рис.17.8.

Результати статичного аналізу, проведеного різними статичними аналізаторами для ПЗ різних типів представлено у таблиці 17.1.

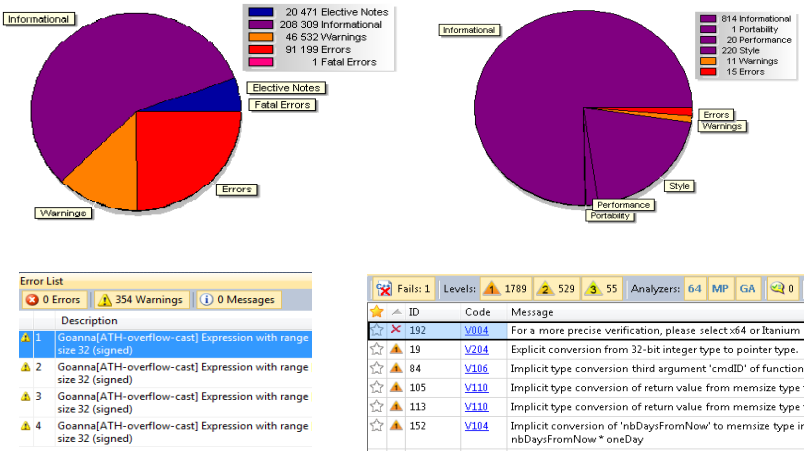


Рис.17.8 - Результати роботи статичного аналізатору CppCheck

Таблиця 17.1 - Результати статичного аналізу різного ПЗ різними аналізаторами

	Asteroid (14357)			Solar System (16789)			Notepad++ (31174)		
	Errors	Warnings	Other	Errors	Warnings	Other	Errors	Warnings	Other
Visual Studio	0	0	0	0	0	0	0	0	0
PVS-Studio	5	7	107	6	7	125	1789	529	55
Goanna Studio	0	60	0	0	38	0	0	85	0
CppCheck	16	31	41	21	39	65	4	155	132
PC-Lint	206	2881	14003	207	3247	15493	65040	6400	39242

	Ac3 Filter (14609)			Crystal DiskInfo (9007)			Crystal DiskMark (1785)		
	Errors	Warnings	Other	Errors	Warnings	Other	Errors	Warnings	Other
Visual Studio	0	0	0	0	0	0	0	0	0
PVS-Studio	288	1725	766	244	1238	544	13	38	294
Goanna Studio	0	31	0	0	0	350	0	11	0
CppCheck	79	162	86	11	36	82	1	25	139
PC-Lint	9	4784	10049	2582	3044	5878	902	533	1235

10. Основи тестування програмного забезпечення

Тестування ПЗ полягає у перевірці правильності результатів роботи програми з спеціальними наборами вхідних даних, які називають *тестами*.

Суть тестування полягає у перевірці роботи програм з даними, що подібні реальним і мають оброблятися у процесі експлуатації ПЗ. Під час тестування виявляються відмови та дефекти програм і невідповідність вимогам. Це здійснюється шляхом дослідження даних на виході системи і виявлення серед них таких, яких не повинно бути на виході.

Під *помилкою ПЗ* взагалі розуміють неправильність його функціювання, що призводить до хибного чи спотвореного результату обчислювального процесу. Під *збоєм*, у тому числі і ПЗ, розуміють відмову, що самоусувається.

Розглянемо механізм виникнення відмов ПЗ, враховуючи те, що відмова ПЗ зумовлена невідповідністю програмного забезпечення поставленим задачам. Як правило, вона виникає внаслідок порушення специфікації, тобто технічних вимог до програми, або неточної чи неповної специфікації. Порушення специфікації частіше зустрічається в складних програмних системах, де окремі помилки програміста важко спостережувані і тому залишаються невиявленими. Неточність чи неповність специфікації виникає за причини численних факторів при складанні специфікації. На початковому етапі розроблення програми вони невідомі і з'ясовуються поступово у процесі її експлуатації. Відмови ПЗ може зумовлювати також *прихованість помилок*. У цьому випадку помилка проявляється тільки в окремих комбінаціях, що рідко зустрічаються. Тому вони виявляються тільки у процесі тривалої експлуатації ПЗ. Такі помилки є найбільш небезпечними.

Під час тестування виявляються відмови та дефекти програм і невідповідність вимогам. Це здійснюється шляхом дослідження даних на виході системи і виявлення серед них таких, яких не повинно бути на виході. Тестування здійснюється у цьому випадку як на етапі реалізації програмної системи з метою перевірки її відповідності очікуванням замовника, так і після завершення її реалізації.

Тестування програмного забезпечення (ПЗ) забезпечує виявлення (констатацію наявності) фактів розбіжностей з вимогами (помилок). Ці помилки знижують рівень якості ПЗ відносно

очікувань. Різниця між очікуваною та одержаною якістю проявляється у ризиках для софтверної компанії-розробника, а також для замовників, користувачів та інших зацікавлених осіб.

Є багато трактувань і розумінь тестування програм та програмного забезпечення. Одним з найбільш узагальнених є таке трактування, коли під тестуванням розуміють процес виявлення відмов за причини наявності помилок у програмах.

Варте уваги трактування тестування як запуску виконуваного коду з тестовими даними і дослідження даних на виході програмної системи та робочих характеристик програмного продукта для перевірки правильності роботи системи.

Найбільш поширеним трактуванням тестування є таке, коли *суть тестування* (testing) полягає у перевірці роботи програм за результатами їх виконання з використанням спеціально підібраних вихідних даних, які називають тестами. Це, фактично, метод виявлення наявності помилок у програмах шляхом опрацювання тестових даних і порівняння одержаних при цьому результатів з розрахунковими.

У багатьох випадках спеціалістам, які виконують тестування ПЗ, додають інші завдання, зокрема, виконання налагодження ПЗ. Найчастіше це робиться, коли тестувальник здійснює тестування програмних модулів та перевірку їх взаємодії, а також функціювання компонентів програмної системи. Тому інколи в процес тестування програм (рис.17.9) помилково включають операції, що відносяться до процесу налагодження, суть якого полягає у виявленні, локалізації та усуненні помилок у програмі.

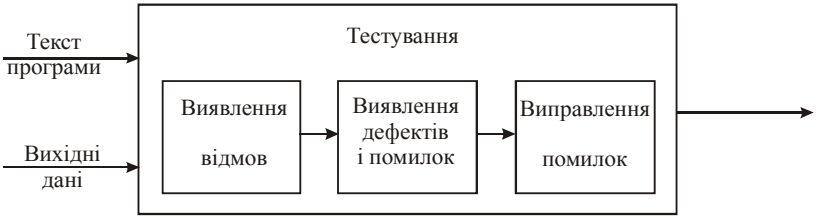


Рис.17.9 - Процес тестування ПЗ

Взагалі, тестування і налагодження є двома відокремленими процесами. Під час тестування виявляється тільки наявність чи ненаявність відмов за причин наявності помилок. При цьому тестування не передбачає виявлення і усунення помилок. Це здійснюється тільки у процесі налагодження. Тому під налагодженням розуміють процес виявлення джерел відмов чи помилок і внесення в програму відповідних виправлень.

Взаємозв'язок тестування і налагодження ПЗ та інформаційні потоки цих процесів показано на рис.17.10.

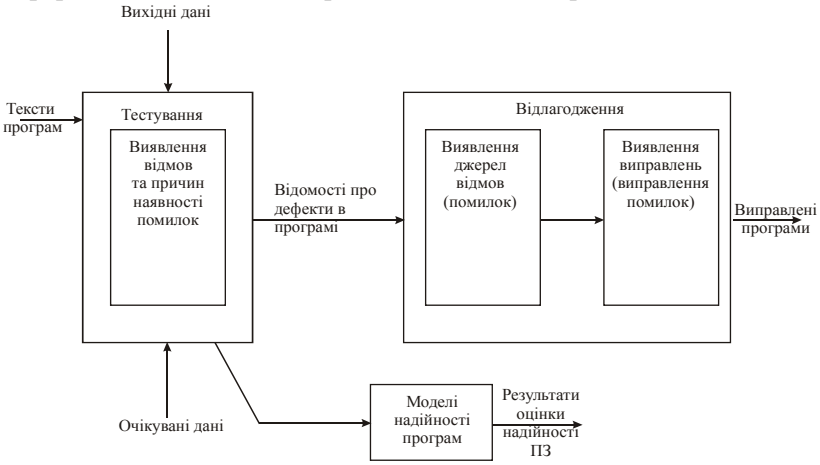


Рис.17.10 - Взаємозв'язок тестування та налагодження ПЗ

Тестування ПЗ може полягати у:

- виявленні помилок, які будуть усунені, або навіть запобіганні їх внесення;

- виявленні помилок, які не будуть усунені, але про них стає відомо;

- проведенні тестів, які знижують ризики;

- забезпеченні проекту своєчасною, точною інформацію, якій можна довіряти.

З такої позиції *функційно тестування ПЗ* - це процес оцінювання якості ПЗ, надання послуг та інформації, який дозволяє софтверній компанії керувати ризиками якості ПЗ, а *організаційно* -

це група фахівців, яка надає дані послуги та інформаційні продукти для проекту, в якому вона задіяна.

Перш ніж приступати до тестування ПЗ, слід впевнитись, що код працює (усунути синтаксичні помилки, усунути статичні семантичні помилки - це дозволяє виконати компілятор або інтерпретатор), а також створити множину очікуваних результатів (пари "вхід-вихід") - *тестові набори* (test case), які можуть проявити помилку та локалізувати її. Тестові набори можна створити на основі документації до ПЗ, обмежень на модулі, очікувань на входах-виходах, припущень у коді.

Тестовий набір повинен бути ефективним, тобто давати високу ймовірність прояву помилок. Для побудови такого тестового набору слід:

1) розбити множину входів на підмножини, які забезпечують еквівалентну інформацію про коректність роботи програми - групи підмножин організовувати таким чином, щоб кожен елемент множини знаходився лише в одній підмножині ;

2) створити тестовий випадок, який містить один вхід для кожного елементу підмножини.

Тестування програмного забезпечення здійснюють з метою впевнитися, що ПЗ відповідає певним критеріям і вимогам кінцевого користувача, в першу чергу замовника. За допомогою тестування необхідно розпізнати дефекти ПЗ. Основна мета тестування полягає у підвищенні ймовірності того, що ПЗ за будь-яких умов буде функціювати належним чином і відповідатиме встановленим вимогам. Тестування ПЗ повинно забезпечувати виявлення помилок в ньому, демонстрацію відповідності функцій ПЗ його призначенню, реалізації вимог до його характеристик, відображення якості та надійності ПЗ.

Типи і методи тестування ПЗ представлені на рис.17.11.

Функційне тестування призначене для перевірки відповідності ПЗ його специфікації (завданню) або еталону, тобто перевірки, чи виконує програмна система або її модулі відповідні функції та поставлені вимоги. При цьому тестувальник перевіряє виконувані функції, а не реалізацію ПЗ шляхом аналізу вхідних і вихідних даних. В літературі цей метод називають ще *методом*

тестування "чорної скриньки". Для реалізації цього методу повинні бути відомі функції програм ПЗ. Функційне тестування використовує евристичні методи, засновані на вивченні шляхів специфікацій.

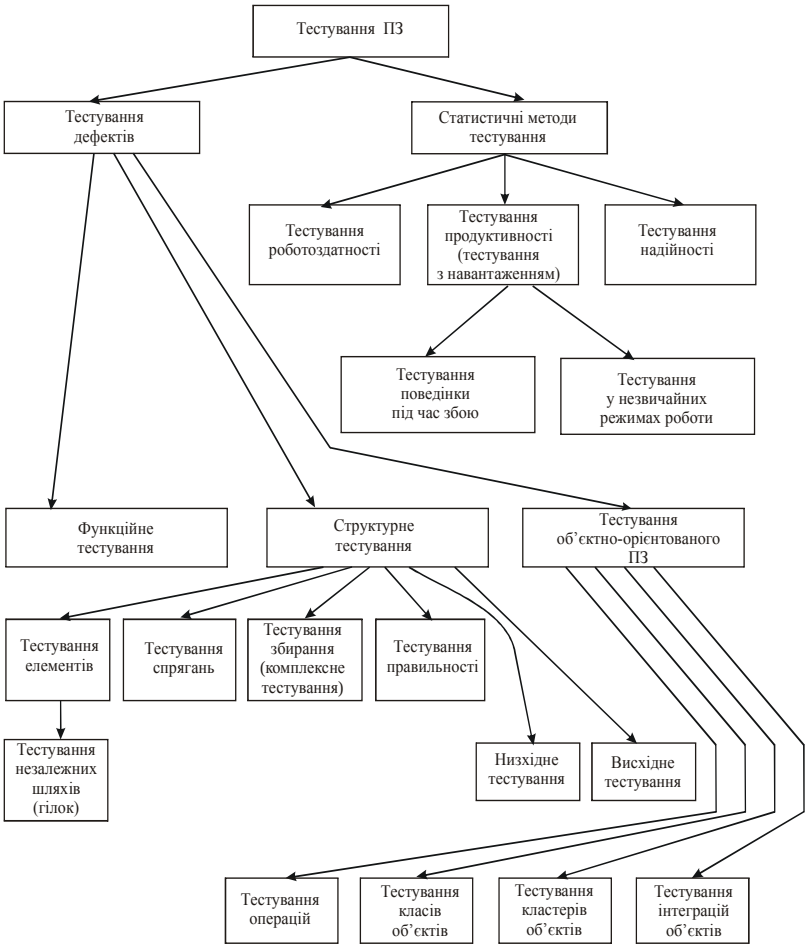


Рис.17.11 - Типи і методи тестування ПЗ

Результати функційного тестування дають відповіді на такі питання:

1) як виконуються функції програм ПЗ?

2) як сприймаються вхідні дані?

3) як виробляються результати на виходах “чорної скриньки”?

4) як зберігається цілісність зовнішньої інформації?

При *структурному тестуванні* перевіряється коректність побудови всіх елементів програм і правильність їх взаємодії між собою. Цей підхід по-іншому називають *методом тестування “прозорої скриньки”*. Інколи використовуються такі назви цього методу, як метод “білої скриньки” або “скляної скриньки”. Для реалізації цього методу повинна бути відома детальна внутрішня структура програм. Об’єктом тестування у такому випадку є внутрішня, а не зовнішня поведінка програм. Тому розглядаються внутрішні елементи програм і зв’язки між ними. Досліджується, по суті, логіка програм. Тестування методом “прозорої скриньки” грунтується на керуючій структурі програм. Програми вважаються повністю перевіреними, якщо проведено вичерпне тестування всіх гілок їх графів керування. Для цього формуються такі тестові варіанти, які б гарантували перевірку всіх незалежних маршрутів кожної програми, перевіряли гілки True і False для всіх логічних рішень, виконували всі цикли та аналізували правильність внутрішніх структур даних.

Для створення тестових наборів при структурному тестуванні використовується код. Структурне тестування є вичерпним, якщо кожен потенційний шлях через код тестується принаймні один раз, що не завжди є можливим, оскільки цикл може бути виконаний багато разів, а рекурсія може бути довільної глибини. Вичерпний тестовий набір може пропускати певні помилки. Так, наприклад, розглянемо функцію `abs` мовою `Python`:

```
def abs(x):
    """x - int;
    if x>=0 return x, else return -x """
    if x<-1:    #Помилка! Потрібно: x<=-1
       return -x
    else:
       return x
```

Тестовий набір {-2; 2} буде вичерпним для цієї функції, але він не проявлятиме помилки, оскільки `abs(-1)` некоректно повертатиме -1. Вірний тестовий набір повинен бути таким: {-2; 1; 2}.

Емпіричні правила побудови тестових наборів для структурного тестування ПЗ:

1) виконувати всі гілки умовних операторів;
2) забезпечувати виконання кожного пункту виключення;
3) для кожного циклу створити тести: в цикл немає входу; тіло циклу виконується лише один раз; тіло циклу виконується більше одного разу; простежуються всі шляхи виходу з циклу;
4) для рекурсій: тестування з нерекурсивними викликами; з одним рекурсивним викликом; більше одного рекурсивного виклику.

Процес тестування ПЗ складається з наступних *етапів*:

1) інтегральне тестування - перевіряє кожне збирання інтеграції та кожну ітерацію;
2) тестування артефактів - моделі варіантів використання, тестових варіантів, процедур тестування, оцінок тестів, плану тестування, компонентів тесту, дефектів;
3) перевірка співробітників, які приймають участь у тестуванні - тестових інженерів, розробників компонентів тестів, тестувальників інтеграції та системних тестувальників;
4) тестування інтерфейсів між компонентами;
5) системне тестування - складається з тестів "чорної скриньки" (функціональних), які стверджують узгодженість всього ПЗ з вимогами;
6) тестування зручності та простоти використання - стверджує прийнятність ПЗ для користувачів; полягає у вимірюванні ступеня задоволеності користувачів від застосування ПЗ;
7) регресійне тестування - перевірка того, що ПЗ після внесення змін продовжує проходити той же визначений набір системних тестів, що й до змін;

8) приймально-здавальне тестування - застосовується для переконання клієнта в тому, що розроблене саме ПЗ, яке було замовлене;

9) тестування інсталяції - тестування ПЗ в цільовій апаратно-програмній конфігурації.

При реалізації *тестування критичного ПЗ* повинні бути виконані наступні *дії* [42]:

1) планування тестування;

2) розроблення плану тестування;

3) розроблення вимог до тестів;

4) визначення складових якості ПЗ, які будуть оцінюватись в процесі тестування;

5) визначення складових ПЗ, які піддаються тестуванню;

6) оцінювання потрібних для тестування ресурсів (бюджет, час, персонал);

7) проектування тестів;

8) проектування специфікації тестів;

9) проектування тестових процедур;

10) проектування детальних тестів;

11) проектування інструментального середовища тестування;

12) розроблення тестів;

13) розроблення детальних тестів;

14) розроблення тестових даних;

15) розроблення інструментального середовища тестування;

16) виконання тестування та оцінювання його результатів;

17) виконання тестових прикладів;

18) аналіз результатів тестування;

19) розроблення звіту про тестування.

Всі роботи по тестуванню критичного ПЗ повинні бути ретельно задокументовані.

Документація по тестуванню ПЗ (STD - Software Test Documentation) згідно складається з наступних розділів:

1) вступ - пояснює вміст тестів та їх загальні принципи. Наприклад, якщо ПЗ керує обладнанням приміщень швидкої

допомоги, саме в цьому розділі потрібно пояснити загальний підхід до тестування моделей, який зводиться до тестування в умовах даних приміщень;

2) *план тестування* - описує тестовані елемети, межі, підхід, ресурси, розклад та персонал та пояснює, як слід організувати персонал, програми і обладнання, щоб виконати тестування. Наприклад, "часовий модуль тестуватиме Тестувальник1 протягом тижнів 30-33", "модуль моніторингу серця тестуватиме Тестувальник2 протягом тижнів 26-30", "інтеграцію модулю моніторингу серця та часового модулю тестуватиме Тестувальник3 протягом тижнів 31-33";

3) проект тестування - описує тестовані елементи, підхід, план в подробицях, відображає наступний рівень деталізації після плану тестування. Проект тестування розкриває значення відповідних програмних елементів, описує порядок, в якому їх слід застосувати. Наприклад, "часовий модуль окремо тестуватиме Тестувальник1 протягом тижнів 30-33, використовуючи тестову процедуру 1 і драйвер 2";

4) тестові варіанти - описує набори вхідних даних, подій та точних вхідних сигналів, які повинні використовуватись для виконання тесту. Наприклад, "модуль моніторингу серця повинен працювати за тестовим файлом 25, в якому містяться конкретні дані по конкретному пацієнту в конкретний час";

5) тестові процедури - описує кроки налагоджень та виконання тестових варіантів. До цього розділу входять процедури налагодження, імена необхідних файлів з початковими даними та об'єктним кодом, сирцеві файли, log-файли, файли з тестовими варіантами та звіти;

6) звіт про проведення тестування елементів - описує тестований елемент, використовувані версій ПЗ та фізичне місцерозташування результатів, надає список відповідальних осіб за проведення тестів;

7) журнал випробувань - оформляється у вигляді хронологічних записів "фізичне місцерозташування тесту - назва тесту - результат". Це детальний поточний звіт про одержану під

час тестів інформацію. Він може виявитись корисним при спробі відновити ситуації, в яких тест завершився невдало;

8) звіт про події під час тестування - містить документування будь-якої події, яка мала місце під час тестування і вимагає подальших досліджень - наприклад, відхилення від нормальної роботи ПЗ або припущені під час тестування помилки;

9) підсумковий звіт про тестування - підсумок всього вищезазначеного.

На етапі планування тестування ПЗ потрібно дослідити місце робіт по тестуванню у проекті та вивчити ризики, які тестування дозволить знизити. В ідеалі процес тестування повинен починатись одночасно з виникненням задумки ПЗ та продовжується до кінця життєвого циклу ПЗ, тобто до завершення його існування.

Ризики, які виникають внаслідок різниці між очікуваною та одержаною якістю ПЗ, можуть мотивувати софтверну компанію на інвестиції у якість ПЗ. Звісно, дані попередніх проектів та моделі можуть допомогти у прогнозуванні кількості дефектів розроблюваного ПЗ, але точна кількість, а також серйозність та локалізація проблем якості не відомі наперед. Таким чином, ці потенційні проблеми є ризиками, які піддають небезпеці якість ПЗ. Ефективно організоване тестування надає послуги та інформацію, які сприяють керуванню ризиками якості розроблюваного ПЗ.

У плануванні тестування ПЗ можна виділити наступні *етапи*:

1. визначення функційного (система, проект і процес) та організаційного контекстів, в рамках яких виконується тестування;

2. визначення та пріоритетність ризиків якості системи, а також досягнення консенсусу зацікавлених сторін проекту відносно можливостей тестування для зменшення цих ризиків;

3. оцінка часових, ресурсних та фінансових витрат, необхідних для виконання тестування та узгоджених з попереднім етапом, отримання підтримки оцінки у керівництва;

4. планування задач, залежностей та складу учасників, необхідних для зменшення ризиків якості системи, та одержання підтримки плану у зацікавлених сторін проекту.

В свою чергу, *процес дослідження контекстів* можна розбити на наступні етапи:

1) зрозуміти процеси життєвого циклу, застосовувані у софтверній компанії для розроблення, супроводу або поставки ПЗ, включаючи будь-які плановані або здійснювані ініціативи по вдосконаленню процесів;

2) вивчити всю документацію, яка стосується тестування та якості, даних, метрик та вимірювань щодо попередніх проектів, в тому числі плани тестування, бази виявлених дефектів, дані про помилки, знайдених при експлуатації ПЗ, системи тестів і т.і. Зрозуміти, хто розробив всі ці артефакти, чому це було зроблено саме таким чином і що призвело до таких рішень і таких результатів у компанії;

3) обговорити з іншими учасниками проекту, в яких діях тестування вони прийматимуть участь і що вони планують робити далі;

4) визначити зацікавлених осіб у процесі тестування, провести з ними переговори для розуміння взаємин з тими, хто зараз проводить тестування, їхніх очікувань від процесу тестування, попередніх розчарувань від процесу тестування, конфліктів з тими, хто зараз бере участь у тестуванні;

5) уточнити з безпосередніми керівниками їхні очікування від процесу тестування та оцінки якості.

Процес аналізу ризиків якості ПЗ складається з наступних етапів:

1) визначити ключових осіб, зацікавлених у тестуванні та якості, та одержати їхню згоду на участь у роботах з аналізу ризиків якості;

2) зробити для ключових зацікавлених осіб огляд існуючих методик та методів аналізу ризиків якості та, по можливості, запропонувати методику для практичного застосування у проекті;

3) зібрати ідеї ключових зацікавлених осіб про ризики якості, про види помилок, пов'язаних з цими ризиками, про вплив на якість цих проблем та про пріоритетність ризиків, а також визначити рекомендовані дії по зниженню кожного ризика;

4) повідомляти про будь-які помилки, виявлені в інших проектних документах в ході аналізу, в тому числі про невдалі та пропущені вимоги, проблеми, пов'язані з архітектурою і т.і.;

5) документувати ризики якості у відповідності до обраної методики;

6) розташувати документацію з аналізом ризиків якості у проектну бібліотеку або систему керування конфігурацією та організувати контроль над змінами документації.

Оцінювання ресурсів - це об'ємний складний процес, який вимагає значних зусиль. Процес оцінювання витрат складається з 5 етапів:

1) виконання декомпозиції робіт і складання плану робіт;

2) підготовка бюджету, який відповідає фінансовим обмеженням проекту, на основі декомпозиції робіт та план-графіку;

3) затвердження план-графіку та бюджету керівництвом;

4) уточнення план-графіку та бюджету при необхідності;

5) розташування прийнятих план-графіка та бюджету у проектну бібліотеку.

Процес планування задач, залежностей та складу учасників, необхідних для зменшення ризиків якості системи можна розбити на наступні етапи:

1) дослідити, продумати, зібрати та документувати стратегію, тактику та складові роботи підпроекту тестування;

2) обговорити та документувати спільні роботи підпроекту тестування та загального проекту розроблення;

3) завершити підготовку та документування неврахованих логічних деталей та подробиць плану, в тому числі ризиків самого підпроекту тестування;

4) розіслати план для закритого рецензування, зібрати відгуки та переглянути план;

5) розіслати план для відкритого рецензування, провести обговорення для затвердження плану;

6) переглянути оцінку графіку та бюджету на основі нової інформації;

7) розташувати план в проектну бібліотеку.

Отже, для покращення процесу тестування потрібно виконувати планування процесу з метою: зрозуміти мету процесу тестування і його планування; зрозуміти контекст тестування, плану та процесу планування; документувати процес тестування; встановити консенсус між тест-менеджерами, тестувальниками, ключовими учасниками процесу проектування та замовниками.

Незалежно від моделі розроблення ПЗ *при плануванні тестування необіхдно відповісти на п'ять питань*, які визначають цей процес:

1) хто буде тестувати і на яких етапах? - розробники, незалежні тестувальники чи сумісно;

2) які компоненти потрібно тестувати? - всі або лише ті, які загрожують найбільшими втратами для всього проекту;

3) коли слід тестувати? - постійно, в спеціальних контрольних точках або на завершуючій стадії розроблення;

4) як слід тестувати? - тільки перевірка функцій ПЗ або також і перевірка, як функції реалізовані;

5) в якому обсязі слід тестувати? - в достатньому обсязі або в межах виділених на тестування ресурсів.

Тестовий план може містити наступну інформацію:

1) перелік тестових ресурсів;

2) перелік функцій та підсистем, які підлягають тестуванню;

3) тестова стратегія:

3.1) аналіз функцій та підсистем з метою визначення слабких місць, які потребують вичерпного тестування, тобто ділянок функціональності, де поява дефектів найбільш ймовірна;

3.2) визначення стратегії вибору вхідних данихдля тестування. Для вирішення цієї задачі можуть бути застосовані методи покриття класів вхідних та вихідних даних, аналіз граничних значень, покриття випадків використання і т.і.;

3.3) визначення потреби автоматизації процесу тестування;

4) графік (розклад) тестових циклів;

5) вказання конкретних параметрів апаратури та програмного оточення;

6) визначення тестових метрик, які необхідно збирати і аналізувати, таких як покриття набору вимог, покриття коду, кількість та рівень серйозності дефектів, обсяг тестового коду і т.і.

Джерела та вміст плану тестування представлено на рис. 17.12.

Рис.17.12 - Джерела та вміст плану тестування

Існують різноманітні *засоби по управлінню тестуванням*:

- TET (Test Environment Toolkit), TETware, Test Manager, RTH - відкриті ресурси;

- HP Quality Center/ALM, QA Complete, T-Plan Professional, Automated Test Designer (ATD), Testuff, SMARTS, QAS.TCS (Test Case Studio), PractiTest, Test Manager Adaptors, SpiraTest, TestLog, ApTest Manager, DevTest - комерційні засоби.

Існують різноманітні *засоби для функційного тестування*:

- Selenium, Soapui, Watir, HTTP::Recorder, WatiN, Canoo WebTest, Webcorder, Solex, Imprimatur, SAMIE, Swete, ITP, WET, WebInject - відкриті ресурси;

- QuickTest Pro, Rational Robot, Sahi, SoapTest, Badboy, Test Complete, QA Wizard, Netvantage Functional Tester, PesterCat, AppsWatch, Squish, actiWATE, liSA, vTest, Internet Macros, Ranorex

Висновки

На думку Д.Паттерсона, світова гонитва за продуктивністю призвела до залежності людини від технологій, і людству пора створювати такі інформаційні технології, на які світ дійсно може покластися, повністю довіряючи їм

Звісно, хаотичний період розвитку ПЗ, коли значно більше уваги приділялось саме програмному коду, а не його якості, став відходити у минуле. За останні роки програмна індустрія досягла такого рівня розвитку, при якому вимоги до забезпечення якості стали обов'язковим пунктом договорів на предмет розроблення програмних систем, оскільки саме якість ПЗ є його найважливішою характеристикою з точки зору користувача.

Гарантування якості ПЗ – проблема, вирішення якої потребує комплексного дослідження за наступними напрямками: 1) розроблення засобів аналізу і оцінювання якості ПЗ на різних етапах його життєвого циклу (ЖЦ); 2) визначення і управління параметрами, які впливають на якість ПЗ на всіх етапах його ЖЦ.

Наразі в більшості програмних проектів основна частина зусиль з виявлення та виправлення помилок все ще виконується на етапі тестування, а то й після випуску ПЗ, отже, на відлагодження та переробку йде близько 50% часу типового циклу розроблення ПЗ. В десятках компаній було виявлено, що політика раннього виправлення дефектів може в кілька разів знизити фінансові та часові витрати на розроблення ПЗ, що є вагомим аргументом на користь якомога більш раннього виявлення та усунення дефектів.

Рекомендовані друковані та Інтернет-джерела:

1. Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 1136 с.

2. Эрик Дж. Брауде. Технология разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.

3. С.Зыль. Проектирование, разработка и анализ програмного обеспечения систем реального времени – СПб: БХВ-Петербург, 2010. – 336 с.

4. И.Соммервилл. Инженерия программного обеспечения. 6-е издание. - Издательский дом “Вильямс”, 2002

5. ISO/IEC 12207:2008. Systems and software engineering — Software life cycle processes

6. IEEE Standard 1074-1995 - IEEE Standard for Developing Software Life Cycle Processes

7. Why the Waterfall Model Doesn’t Work // [Електронний ресурс] - Режим доступу: http://www.infoq.com/resource/articles/scaling-software-agility/en/resources/ch02.pdf

8. Boehm B. A Spiral Model of Software Development and Enhancement - ACM SIGSOFT Software Engineering Notes, ACM, 11(4):14-24, August 1986 // [Електронний ресурс] - Режим доступу: http://weblog.erenkrantz.com/~jerenk/phase-ii/Boe88.pdf

9. Guide to the Software Engineering Body of Knowledge (SWEBOK) - A project of the IEEE Computer Society Professional Practices Committee, 2004 // [Електронний ресурс] - Режим доступу: http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/otros-recursos-1/SWEBOK_Guide_2004.pdf

10. IEEE Standard 1517-99. IEEE Standard for Information Technology – Software Lifecycle Process – Reuse Processes

11. Мацяшек, Лешек. А. Анализ и проектирование систем. Разработка инфорамционных систем с использованием UML. –М.: Издательский дом “Вильямс”.

12. Вендров А.М. Проектирование программного обеспечения экономических информационных систем / А.М. Вендров. - М.: Финансы и статистика, 2002.

13. С.Макконнелл. Совершенный код. Мастер-класс - М.: Издательство "Русская редакция", 2013 - 896 с.

14. Мищенко В.О., Поморова О.В., Говорущенко Т.А. CASE-оценка критических программных систем. В 3-х томах. Том 1. Качество / Под ред. Харченко В.С. - Харьков: Нац.аэрокосмический университет "ХАИ", 2012. - 201 с.

15. IEEE Standard 610.12-90. Standard Glossary of Software Engineering Terminology // [Електронний ресурс] - Режим доступу: http://web.ecs.baylor.edu/faculty/grabow/Fall2011/csi3374/secure/Standards/IEEE610.12.pdf

16. IEEE Standard 1219-98. IEEE Standard for Software Maintenance // [Електронний ресурс] - Режим доступу: http://www.cs.uah.edu/~rcoleman/CS499/CourseTopics/IEEE_Std_1219-1998.pdf

17. IEEE Standard 14764-2006 (ISO/IEC 14764). Standard for Software Engineering - Software Maintenance // [Електронний ресурс] - Режим доступу: http://webstore.iec.ch/preview/info_isoiec14764%7Bed2.0%7Den.pdf

18. ISO 9001: 2008-12. Quality management systems - Requirements // [Електронний ресурс] - Режим доступу: http://nads.gov.ua/sub/data/upload/publication/cherkaska/ua/6331/iso-9001_2008_dvuyazychnyj.pdf?s398224032=63e9aac82dc234cb077145d99b22b9aa

19. Липаев В.В.Сертификация программных средств. Учебник–М.:СИНТЕГ,2010– 348 с.

20. Документирование программного обеспечения и эффективный процесс разработки // [Електронний ресурс] – Режим доступу: http://creograf.ru/?messPress_ShowR_161=1

21. Скляр В.В. Оценка качества и экспертиза программного обеспечения. Лекционный материал. - Харьков: НАУ "ХАИ", 2008. - 204 с.

22. Харченко В.С., Скляр В.В., Гордеев А.А. Верификация программного обеспечения. - Харьков: НАУ "ХАИ", 2006. - 132 с.

23. Крайер Э. Успешная сертификация на соответствие нормам ИСО серии 9000. Пер. с нем. – М.: ИЗДАТ, 1999

24. А.М. Вендров. Современные технологии создания программного обеспечения // [Електронний ресурс] – Режим доступу: http://citforum.ru/programming/application/program/

25. Ю. Ю. Якунин. Технологии разработки программного обеспечения - Красноярск: ИПК СФУ, 2008

26. Д.А.Чернев. Технология разработки програмного обеспечения –Ташкент,2004– 224с.

27. IEEE 1209-1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools.

28. IEEE 1348-1995. IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools.

29. ISO/IEC 14102:1995(E). Information technology - Guideline for the evaluation and selection of CASE Tools

30. Карл И. Вигерс. Разработка требований к программному обеспечению — Русская редакция, 2004.

31. Кобёрн А. Современные методы описания функциональных требований к системам — М.: Лори, 2002.

32. Леффингуелл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению — М.: Вильямс, 2002.

33. IEEE 830-1998. Recommended Practice for Software Requirements Specifications - New York: IEEE, 1998

34. IEEE 1233-1998. Guide for Developing System Requirements Specifications - New York: IEEE, 1998

35. В.Кулямин. Компонентный подход в программировании // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/64/64/lecture/965

36. Подробнее о стадии проектирования // [Електронний ресурс] - Режим доступу: http://testingworld.ru/podrobnee-o-stadii-proektirovania/

37. Ю.Химонин. Сбор и анализ требований к програмному продукту // [Електронний ресурс] - Режим доступу: http://pmi.ru/profes/Software_Requirements_Khimonin.pdf

38. Ф.А.Новиков, Э.А.Опалева, Е.О.Степанов. Учебно-методическое пособие по дисциплине «Управление проектами и

разработкой ПО» // [Електронний ресурс] - Режим доступу: http://window.edu.ru/resource/366/60366/files/itmo305.pdf

39. Технологии командной разработки программного обеспечения информационных систем // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/4806/1054/lecture/9998

40. ISO/IEC 25010:2011. Systems and software engineering - Systems and software Quality Require-ments and Evaluation (SQuaRE) - System and software quality models / ISO/IEC, 2011

41. Д.Андреев. Организация процессов разработки программного обеспечения с использованием Team Foundation Server 2010 // [Електронний ресурс] - Режим доступу: http://www.intuit.ru/studies/courses/649/505/info

42. Бьерк А., ДелаМазаМ., Резник С. ScrumTeamFoundationServer 2010. Профессиональный подход. - М. ЭКОМ Паблишерз, 2012.

43. Левинсон Дж. Тестирование ПО с помощью Visual Studio 2010. - М.: ЭКОМ-Паблишерз, 2012.

44. Шаблоны и средства Microsoft — Microsoft Visual Studio 2010 // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/vstudio//aa718795.aspx

45. Шаблон процесса гибкой разработки для Visual Studio ALM // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/dd380647.aspx

46. Шаблон процесса Scrum для Visual Studio ALM // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/ff731587.aspx

47. Управления жизненным циклом приложений с помощью Visual Studio и Team Foundation Server // [Електронний ресурс] - Режим доступу: http://msdn.microsoft.com/ru-ru/library/fda2bad5.aspx

48. Мейер Дж. Д. Командная разработка с использованием Visual Studio Team Foundation Server / Дж. Д.Мейер, Дж. Тейлор, А. Макман, П. Бансод, К. Джонс - Изд. Корпорация Microsoft, 2007.

Підсумкові контрольні питання з навчальної дисципліни «Технологія проектування програмних систем»

1. Огляд та порівняльний аналіз моделей життєвого циклу ПЗ.
2. Вибір прийнятної моделі життєвого циклу ПЗ.
3. Сучасний стан сфери виробництва програмних засобів.
4. Розповсюджені процеси та етапи розроблення програмних систем.
5. Сертифікація процесів створення ПЗ.
6. Оцінювання процесів створення ПЗ.
7. Визначення технології проектування програмного забезпечення (ТППЗ).
8. Загальні вимоги, пропоновані до ТППЗ.
9. Приклади ТППЗ.
10. Моделі програмних систем.
11. Прототипування програмних систем.
12. Основні фази розроблення ПЗ: формулювання вимог, формулювання цілей проекту, аналіз прикладної галузі, створення функційної специфікації, проектування, реалізація.
13. Стандарти в галузі розроблення ПЗ.
14. Програмні засоби підтримки життєвого циклу.
15. Методи визначення вимог.
16. Планування етапу визначення вимог.
17. Формалізація вимог: виділення вимог за допомогою прецедентів.
18. Формалізація вимог: псевдокод, кінцеві автомати, графічні дерева рішень, візуальне подання вимог за допомогою діаграм UML.
19. Завдання та результати етапу аналізу вимог.
20. Планування архітектури.
21. Проектування архітектури.
22. Документування архітектури.
23. Аналіз архітектури.
24. Поняття ризику.
25. Управління ризиками.

26. Формальні специфікації як засіб підвищення якості ПЗ.

27. Специфікування інтерфейсів.

28. Специфікація поведінки систем.

29. Мови розроблення формальних специфікацій.

30. Методологія розроблення ПЗ Microsoft Solutions Framework (MSF).

31. Принципи створення бібліотеки MSF.

32. Модель команди в MSF, ролеві кластери, масштабованість команд та керування компромісами у MSF.

33. Гнучкий підхід до створення ПЗ, основні принципи гнучкого розроблення.

34. Реалізація концепції керування програмним проектом на всіх етапах життєвого циклу у Visual Studio 2012.

35. Функціональні можливості та архітектура TeamFoundationServer 2012 (TFS).

36. Способи розгортання TFS на одному або декількох серверах, в одному домені, робочі групі або в декількох доменах.

37. Шаблони командних проектів TFS, області керування командними проектами.

38. Питання створення командного проекту, зміст програмної інфраструктури проекту, склад і призначення робочих елементів.

39. Аналіз методології Scrum, робочі елементи шаблону MicrosoftVisualStudioScrum. Організація колективу у методології Scrum.

40. Методика оцінки трудомісткості розробки ПЗ на основі функціональних точок.

41. Алгоритмічне моделювання трудомісткості розробки ПЗ.

42. Поняття та модель якості ПЗ.

43. Основні принципи метричного аналізу.

44. Вибір метрик, придатних до використання на етапі проектування ПЗ.

45. Дослідження результатів метричного аналізу.

46. Засоби статичного аналізу програмного коду.

47. Дослідження результатів статичного аналізу.

Перелік використовуваних скорочень
у конспекті лекцій з навчальної дисципліни
«Технологія проектування програмних систем»

CMM - модель Capability Maturity Model
DET - нерекурсивний елемент даних
EI - вхідний елемент додатку
EIF - зовнішній інтерфейсний файл
EO - вихідний елемент додатку
EQ - зовнішній запит
FP - функційна точка
FTR - файл типу посилань
ILF - внутрішній логічний файл
IT-рішення - рішення в галузі інформаційних технологій
LOC, SLOC - кількість рядків коду
MSF - методологія Microsoft Solutions Framework
OP - об'єктна точка
RET - підгрупа елементів даних ILF
RUP - технологія Rational Unified Process
TFS - методологія Team Foundation Server

ВВ - варіант використання
ЖЦ ПЗ - життєвий цикл програмного забезпечення
ІС - інформаційна система
ІСОП - інтелектуальна система оцінювання і прогнозування складності та якості програмного забезпечення
МНПЗ - моделі надійності програмного забезпечення
НМОП - нейромережний метод оцінювання результатів проектування і прогнозування характеристик складності та якості програмного забезпечення
ПЗ - програмне забезпечення
ПС - програмна система
СА - статичний аналіз коду
СУБД - система управління базами даних
ТППЗ - технологія проектування програмного забезпечення
ШНМ - штучна нейронна мережа

ЗМІСТ