МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

С.В. Цюцюра, Є.В. Бородавка

СУЧАСНІ МЕТОДОЛОГІЇ ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій

Київ, 2020

Зміст

С.

ТЕМА 1. Вступ у технології програмування. Моделі життєвого циклу програмного забезпечення

Базові поняття, види програмного забезпечення

Технологія програмування - дисципліна, що вивчає технологічні процеси програмування та порядок їх проходження.

Споріднені поняття до технології програмування як дисципліни:

- **Комп'ютерні науки (Computer science)** – сукупність знань та практичних навичок, які використовують спеціалісти з інформаційних систем, обчислювальної техніки, інформатики.
- **Системна інженерія (System engineering)** – розділ науки, що вивчає питання розроблення комп'ютерних систем (архітектура, дизайн, інтеграція, ПЗ та ін.).
- **Програмна інженерія (Software engineering)** – дисципліна, спрямована на розроблення й супроводження програмного забезпечення систем, що функціонують надійно та ефективно, можуть вдосконалюватися й еволюціонувати та відповідають вимогам, визначеним замовником [1].

Програмування (Programming) - процес підготовки задач для їх розв'язання за допомогою комп'ютера; ітераційний процес складання програм.

Програма – дані, призначені для управління конкретними компонентами системи обробки інформації з метою реалізації певного алгоритму [2], послідовність машинних команд, призначена для досягнення конкретного результату.

Програмне забезпечення (ПЗ/Software) – комп’ютерні програми, процедури, а також документація й дані, що з ними асоційовані, які стосуються функціонування комп’ютерної системи [3].

Уперше термін software увів відомий статистик Джон Т'юкей (John Tukey) у 1958 р. для позначення різниці апаратного забезпечення ЕОМ (hardware) від засобів обробки даних.

Б'ярне Страуструп (Bjarne Stroustrup) зазначив, що добре ПЗ не можна побачити, але можна відчути, коли воно працює із помилками [4].

За видами виконуваних функцій програмне забезпечення поділяється на системне, прикладне та інструментальне. Такий поділ є умовним, оскільки широке запровадження комп'ютеризації привело до того, що майже кожна програма має ознаки кількох видів ПЗ.

Види програмного забезпечення

Системне ПЗ (System software) призначене для управління роботою комп'ютера, розподілу його ресурсів, підтримки діалогу з користувачами, а також для часткової автоматизації розроблення нових програм. Як правило, системні програми забезпечують взаємодію інших програм з апаратними складовими, організацію інтерфейсу користувача. Виділяють три типи системного ПЗ:

- **операційна система (ОС)** - програмне забезпечення, що забезпечує інфраструктуру, на якій можуть працювати прикладні програми. Найпоширеніші ОС - Microsoft Windows, Mac OS X та Linux;
- **системи програмування** - призначені для полегшення та часткової автоматизації процесу розроблення та відлагодження програм;
- **сервісні програми (утиліти)** розширюють можливості ОС. До утиліт відносять архіватори, антивіруси, драйвери та ін.

Прикладне ПЗ (application, application software) - комп'ютерна програма, що вирішує конкретні задачі фахової діяльності користувача.

Інструментальне ПЗ призначене для розроблення всіх видів інформаційно-програмного забезпечення. При цьому під інформаційним забезпеченням розуміють сукупність

попередньо підготовлених даних, необхідних для роботи програмного забезпечення. До інструментального ПЗ відносять текстові редактори, системи керування базами даних, транслятори мов програмування.

Розроблення програмного забезпечення

Інтегроване середовище розроблення програмного забезпечення (integrated development environment, IDE) – це система програмних засобів, що використовується програмістами для розроблення програмного забезпечення.

Як правило, середовище розроблення включає текстовий редактор, компілятор і/або інтерпретатор, засоби автоматизації складання, налагоджував та різноманітні інструменти для конструювання графічного інтерфейсу користувача. Значне поширення об'єктно-орієнтованого програмування (ООП) привело до того, що сучасні інструменти розроблення включають браузер класів та інспектор об'єктів. На сьогодні до середовищ розроблення підключають систему керування версіями, засоби тестування та ін. Раніше середовища розроблення переважно призначалися для однієї мови (**Delphi, Turbo Pascal**, **Borland C++**,**Visual Basic**), але на сьогодні широко застосовувані такі середовища, як **Eclipse** або **Microsoft Visual Studio**, призначені для мультимовного розроблення ПЗ.

Розвиток мов, стилів та технологій програмування

Ранні мови програмування

Перші електронні обчислювальні машини (ЕОМ) виникли відносно недавно – у 40-ві роки XX століття. Слідом за цим виникли й перші мови програмування, які були досить примітивні і орієнтовані на числові розрахунки. Це були і суто теоретичні наукові розрахунки (математичні й фізичні), і прикладні завдання, в першу чергу у галузі військової справи.

Програми, написані на ранніх мовах програмування, були лінійними послідовностями елементарних операцій з регістрами, в яких зберігалися дані. Тому будь-що технологія

програмування була відсутня. Перші кроки в розробленні технології полягали у *представленні програми у вигляді послідовності операторів* – так званий **операторний підхід**. Написанню послідовності машинних команд передувало складання операторної схеми, що відбивала послідовність операторів і переходи між ними. У цей період почало зароджуватися поняття алгоритму.

Потрібно відзначити, що ранні мови програмування були оптимізовані під ту апаратну архітектуру конкретного комп'ютера, для якого вони призначалися, і хоча вони забезпечували високу ефективність обчислень, ніякої стандартизації ще не було. Програма, що була цілком працездатною на одній обчислювальній машині, часто не могла бути виконана на іншій. Таким чином, ранні мови програмування істотно залежали від середовища обчислень і приблизно відповідали сучасним машинним кодам або мові асемблера. Перші мови програмування часто називають мовами низького рівня.

Час появи: *1940 роки.*

Стисла характеристика: *лінійна послідовність елементарних інструкцій «низького рівня».*

Переваги: *висока обчислювальна ефективність.*

Недоліки: *істотна залежність від середовища обчислень.*

Приклади*: машинні коди, асемблери.*

Імперативне програмування (Imperative programming)

Приблизно у 50-ті роки XX століття з’явилися мови програмування так званого «високого рівня» порівняно з раніше розглянутими попередниками.

Різниця від мов низького рівня полягає у підвищенні ефективності праці розробників за рахунок **абстрагування від конкретного апаратного забезпечення**. Один оператор мови високого рівня відповідав послідовності з кількох низькорівневих команд. Виходячи з того, що *программа фактично являла собою набір директив, звернених до*

комп'ютера, такий підхід до програмування назвали **імперативним**.

Можна вважати що в цей період з'явились стилі, або парадигми, програмування. На рис.1 наведена схема їх розвитку.

Наступним кроком розвитку програм стало підвищення їх структурованості – **структурний підхід**, при якому *виділяли канонічні структури*: лінійні ділянки, цикли та розгалуження. Завдяки цьому з'явилася можливість читати і перевіряти програму як текст, а це підвищило ефективність праці програмістів під час розроблення та відлагодження програм.

Рисунок 1 – Стилі програмування

Процедурний підхід (Procedural programming) та імперативні мови програмування

Розмір програм постійно збільшувався, тому програмісти почали об'єднувати окремі їх частини у підпрограми, які

групували у бібліотеки. Бібліотеки підключалися до основної робочої програми, яка за необхідності викликала потрібну підпрограму. Фактично такий підхід збільшив структурність програм – велика *програма стала сукупністю процедур-підпрограм.* Одна підпрограма, головна, розпочинала роботу всієї програми. Тобто відбувся перехід до наступного етапу розвитку технологій – **процедурного програмування.**

При розробленні окремої процедури для використання інших процедур потрібно було знати лише їх призначення та спосіб. Процедурний підхід дозволив скоротити затрати праці і машинного часу на створення і модернізацію програм завдяки можливості зміни окремої процедури без втручання в інші. З'явилася можливість повторного використання раніше написаних програмних блоків завдяки їх ідентифікації і наступного звернення за цим ідентифікатором.

Також завдяки структурності програм була збільшена їх надійність – підпрограми стали зв'язуватись одна із одною тільки шляхом передачі їм аргументів, змінні розподілилися на локальні і глобальні.

Окрім того, поява мов високого рівня значно зменшила залежність реалізації від апаратного забезпечення. Щоб це реалізувати, були створені спеціалізовані програми – **транслятори,** призначені для перетворення інструкції мови програмування у коди певної машини. Використання трансляторів привело до певної втрати швидкості обчислень, але цей недолік компенсувався значним виграшем у швидкості розроблення і модифікації програм.

Також у цей період розпочали розроблення спеціалізованих мов програмування для розв'язання конкретних класів задач: для систем керування базами даних, імітаційного моделювання та ін.

У той час програми все частіше розроблялися для виконання завдань військового призначення, для космічної галузі, енергетики – і від надійності програмного забезпечення залежали людські життя. Одним із напрямків удосконалення мов програмування стало підвищення рівня типізації даних.

Використання жорстко типізованої мови при розробленні програми дозволяє ще під час її трансляції у машинні коди виявити більшість помилок використання даних і цим підвищити якість програми. Але типізація обмежує свободу програміста і не дозволяє виконувати частину перетворень даних, що часто потрібно у системному програмуванні.

Практика програмування показала, що велика частина високорівневих мов, створених у період процедурного підходу, дуже вдало реалізована, тому вони або їх варіації використовуються до цього часу. Наприклад, до цього часу використовується мова Fortran для реалізації обчислювальних алгоритмів, мова COBOL для опису бізнес-процесів або мова APL, що поетапно трансформувалась у мову С (Сі).

Потреба підвищення рівня типізації мов програмування привела до появи мови Pascal (Паскаль). Одночасно з Паскалем була розроблена мова С, що здебільшого орієнтована на системне програмування і є слабко типізованою мовою.

Час появи: *1950 роки.*

Стисла характеристика: *програма - послідовність інструкцій-операторів, у яку включені блоки типових дій – процедури або функції.*

Переваги:

- *підвищення рівня абстракції;*
- *менша машинна залежність;*
- *більша сумісність;*
- *змістовна значущість текстів програм;*
- *уніфікація програмного коду;*
- *підвищення ефективності праці програмістів.*

Недоліки:

- *більші витрати на вивчення мов;*
- *менша обчислювальна ефективність.*

Приклади: *Fortran, ALGOL, PL/1, APL, BPL, COBOL, Pascal, C, Basic.*

Декларативне програмування (Declarative programming)

У 60-х рр. XX століття виникає новий підхід до програмування, що до цього часу успішно конкурує з імперативним, а саме – **декларативний підхід**. При декларативному підході до програмування *програма є не набором команд, а описом дій*, які потрібно виконати.

Цей підхід легко формалізується математичними засобами. У результаті програми легше перевіряти на помилки і відповідність технічним специфікаціям (верифікувати).

Також даний підхід має високий ступінь абстракції. Фактично програміст оперує не набором інструкцій, а абстрактними поняттями, часто досить узагальненими.

На початку декларативним мовам програмування було важко конкурувати з імперативними у зв'язку із проблемами при створенні ефективних трансляторів. Програми працювали повільніше, однак вони вирішували більш абстрактні завдання із меншити затратами.

Наприклад, мова SML була розроблена як засіб доведення теорем. Діалекти мови LISP виникли завдяки тому, що ця мова є ефективною під час обробки символьної інформації.

Функціональне програмування (Functional programming)

Одним із шляхів розвитку декларативного стилю програмування став **функціональний підхід**, що виник із появою і розвитком мови LISP.

Відмінною особливістю даного підходу є та обставина, що будь-що *програма*, написана такою мовою, *може інтерпретуватися як функція з одним або кількома аргументами,* деякі з яких також можна розглядати як функції. Складні програми при такому підході будуються за допомогою поєднання функцій. Такий підхід дає можливість прозорого моделювання тексту програм математичними засобами.

При функціональному програмуванні повторне використання коду зводиться до виклику раніше описаної функції, структура якої, на відміну від процедури імперативної

мови, прозора математично. Більш того, типи окремих функцій, що використовуються у функціональних мовах, можуть бути змінними. Тобто забезпечується можливість обробки різнорідних даних (наприклад, упорядкування елементів списку за зростанням для цілих чисел, окремих символів і рядків), або поліморфізм. Завдяки реалізації механізму зіставлення із зразком, такі мови, як ML та Haskell, дуже добре застосовні для символьної обробки.

Ще однією важливою перевагою реалізації мов функціонального програмування є автоматизований динамічний розподіл пам'яті комп'ютера для зберігання даних. При цьому програміст позбавляється від рутинного обов'язку контролювати дані.

До недоліків мов функціонального програмування відносять нелінійну структуру програми і відносно невисоку ефективність реалізації. Однак перший недолік досить суб'єктивний, а другий успішно подоланий сучасними реалізаціями, зокрема низкою останніх трансляторів мови SML, включаючи і компілятор для середовища Microsoft. NET.

Час появи: *1960 роки.*

Стисла характеристика: *програма - функція, аргументи якої, можливо, також є функціями.*

Переваги:

• повністю автоматичне управління пам'яттю комп'ютера («збирання сміття»);

• простота повторного використання фрагментів коду;

• розширена підтримка функцій з параметричними аргументами;

• простота верифікації і тестування програм;

• строгість математичної формалізації;

• високий ступінь абстракції, абстрагування від машинного представлення даних;

• прозорість реалізації рекурсивних функцій;

• зручність символьної обробки даних (списки, дерева).

Недоліки:

• складність ефективної реалізації;

• необхідність фундаментальних математичних знань;
• нелінійна структура програми;
• відносно низька ефективність.

Приклади: *LISP (Interlisp, Common Lisp, Scheme), SML, Haskell, Prolog, Miranda, Hope.*

Логічне програмування (Logic programming)

У 70-х рр. XX століття виникла ще одна гілка мов декларативного програмування, пов'язана з проектами у галузі штучного інтелекту – мови **логічного програмування**.

Згідно з логічним підходом до програмування *програма є сукупністю правил або логічних висловлювань*. Мови логічного програмування базуються на класичній логіці і застосовні для систем логічного висновку, зокрема для так званих експертних систем. На мовах логічного програмування, природно, формалізується логіка поведінки, їх можна застосовувати для описів правил прийняття рішень, наприклад, у системах, орієнтованих на підтримку бізнесу.

Важливою перевагою підходу є досить високий рівень машинної незалежності, а також можливість відкотів – повернення до попередньої мети при негативному результаті аналізу одного з варіантів у процесі пошуку рішення (скажімо, чергового ходу при грі в шахи), що позбавляє від необхідності пошуку рішення повним перебором варіантів і збільшує ефективність реалізації.

Одним з недоліків логічного підходу в концептуальному плані є специфічність класу вирішуваних завдань.

Інший недолік практичного характеру полягає у складності ефективної реалізації для прийняття рішень у реальному часі, скажімо, для систем життєзабезпечення.

Як приклади мов логічного програмування можна навести Prolog (назва виникла від слів PROgramming in LOGic) і Mercury.

Час появи: *1970 роки.*

Стисла характеристика: *програма - сукупність правил або логічних висловлювань з причиною і наслідком.*

Переваги:

• високий рівень абстракції;

• зручність програмування логіки поведінки;

• зручність застосування для експертних систем;

• механізм відкотів (backtrack).

Недоліки:

• обмежене коло завдань;

• нелінійна структура програми;

• недостатньо ефективна реалізація.

Приклади: *Prolog, Mercury.*

Об’єктно-орієнтоване програмування (Object-oriented programming)

Усі універсальні мови програмування, незважаючи на відмінності у синтаксисі і використовуваних ключових словах, реалізують одні й ті самі канонічні структури: оператори присвоювання, цикли і розгалуження.

У всіх сучасних мовах наявні базові типи даних (цілі і речові арифметичні типи, символьний і, можливо, рядковий тип), є можливість використання агрегатів даних, у тому числі масивів і структур (записів).

Разом з тим при розробленні програми для розв’язання конкретної прикладної задачі потрібна як можна більша близькість тексту програми до опису завдання. Одним із шляхів вирішення цієї проблеми було створення розширюваної мови, що містить невелике ядро і допускає розширення, доповнює мову типами даних і операторами, відбиває концептуальну сутність конкретного класу задач.

Розвитком цього підходу і стало **об’єктно-орієнтоване програмування (ООП)** – стиль програмування, що фіксує поведінку реального світу таким способом, при якому деталі його реалізації приховані. У рамках даного підходу програма є описом об'єктів, їх властивостей (або атрибутів), сукупностей (або класів), відносин між ними, способів їх взаємодії та операцій над об'єктами (або методів).

Механізм успадкування атрибутів і методів дозволяє будувати похідні поняття на основі базових і таким чином створювати модель як завгодно складної предметної області із заданими властивостями.

Ще однією важливою властивістю ООП є підтримка механізму обробки подій, які змінюють атрибути об'єктів і моделюють їх взаємодію у предметній області. Переміщаючись по ієрархії класів від більш загальних понять предметної області до більш конкретних і навпаки, програміст отримує можливість змінювати ступінь абстрактності погляду на модельований ним реальний світ.

Використання раніше розроблених (можливо, іншими колективами програмістів) бібліотек об'єктів і методів дозволяє значно заощадити трудовитрати при виробництві програмного забезпечення, особливо типового.

Об'єкти, класи і методи можуть бути поліморфними, що робить реалізоване програмне забезпечення більш гнучким і універсальним.

Складність адекватної (несуперечливої і повної) формалізації об'єктної теорії породжує труднощі тестування та верифікації створеного програмного забезпечення. Ця обставина є одним з найбільш істотних недоліків ООП.

Найбільш відомим прикладом об'єктно-орієнтованої мови програмування є мова C++, що виникла з імперативної мови C. Її логічним продовженням є мова C#.

Час появи: *1970 роки.*

Стисла характеристика: *програма - опис об'єктів, їх сукупностей, відносин між ними і способів їх взаємодії.*

Переваги:

- *інтуїтивна близькість до довільної предметної області;*
- *моделювання як завгодно складних предметних областей;*
- *подієва орієнтованість;*
- *високий рівень абстракції;*
- *повторне використання описів;*
- *параметризація методів обробки об'єктів.*

Недоліки:

• складність тестування та верифікації програм.

Приклади: *C ++, Visual Basic, C #, Eiffel, Oberon.*

Подієво-кероване програмування (Event-driven programming)

Розвитком об'єктно-орієнтованого підходу став перехід до подієво-керованої концепції у 90-х рр. XX століття і виникнення цілого класу мов програмування, які отримали назву мов сценаріїв або скриптів.

У рамках даного підходу програма є сукупністю можливих сценаріїв обробки даних, вибір яких ініціюється настанням тієї чи іншої події (клік по кнопці мишки, перехід курсора в ту чи іншу позицію, зміна атрибутів того чи іншого об'єкта, переповнення буфера пам'яті та ін.). Події можуть ініціюватися як операційною системою, так і користувачем.

Час появи: *1990 роки.*

Коротка характеристика: *програма - сукупність описів можливих сценаріїв обробки даних.*

Переваги:

• інтуїтивна зрозумілість;

• близькість до предметної області;

• високий ступінь абстракції;

• можливість повторного використання коду;

• сумісність з інструментальними засобами автоматизованого проектування (CASE) і швидкого розроблення (RAD) прикладного програмного забезпечення.

Недоліки:

• складність тестування та верифікації програм;

• множинні побічні ефекти.

Приклади: *VBScript, PowerScript, LotusScript, JavaScript.*

Паралельні обчислення (Parallel computing)

Ще одним дуже важливим класом мов програмування є мови підтримки паралельних обчислень.

Програми, написані на цих мовах, є сукупністю описів процесів, які можуть виконуватися як одночасно, так і в псевдопаралельному режимі. В останньому випадку пристрій, що обробляє процеси, функціонує в режимі поділу часу, виділяючи час на обробку даних, що надходять від процесів, у міру необхідності (а іноді з урахуванням послідовності або пріоритетності виконання операцій).

Час появи: *1980 роки.*

Стисла характеристика*: програма - сукупність описів процесів, які можуть виконуватися одночасно або псевдопаралельно.*

Переваги:

• висока обчислювальна ефективність для великих програмних систем (тисячі одночасно працюючих користувачів або комп'ютерів);

• висока ефективність функціонування в системах реального часу (системи життєзабезпечення та прийняття рішень).

Недоліки:

• висока собівартість розроблення невеликих програм (сотні рядків коду);

• відносно вузький спектр застосування.

Приклади: *Ada, Modula-2, Oz.*

Компонентне програмування (Component-based programming)

Компонентне програмування – парадигма програмування, що виникла як набір певних обмежень, що накладаються на механізм об'єктно-орієнтованого програмування, коли стало зрозуміло, що безконтрольне застосування ООП приводить до виникнення проблем з надійністю великих програмних комплексів.

Компонентне програмування визначає набір правил та обмежень, спрямованих на побудову великих програмних систем, здатних до розвитку впродовж тривалого життєвого циклу. При компонентному програмуванні програмна система складається із окремо створених елементів (компонентів), які

викликають один одногой через інтерфейси. Зміни в існуючу систему вносяться додаванням нових компонентів або заміною існуючих. При цьому нові компоненти, які заміняють раніше створені, повинні наслідувати інтерфейси базового.

Поняття технології програмування як процесу

Технологія – це сукупність правил, методик та інструментів, які дозволяють налагодити виробничий процес випуску певного продукту, в тому числі процеси планування, вимірювання характеристик, оцінки якості, відповідальності виконавця та ін.

До цього часу наявне протистояння двох позицій щодо поняття технологій програмування: з одного боку, під ним розуміють широке використання інструментальних засобів, а з іншого – технологія – це набір формальних методик та регламентів, які дозволяють на кожному етапі проводити експертизу, архівування та визначення обсягу та якості виконаної роботи. Перший варіант підтримують професійні програмісти, другий – керівники проектів.

Виконання замовлення у промисловості має багато особливостей. Наприклад, у промисловості ротація кадрів є частим явищем, звідси необхідність архівації та інших засобів відчуження результатів роботи від виконавця; у будь-якому колективі постійно виникають проблеми оцінки індивідуальної роботи та поділу відповідальності за прийняті рішення та помилки, звідси суворе документування та стандартні процедури роботи.

Особливо багато проблем виникає із розумінням поставленої задачі. Тому обов’язково потрібно оформлювати технічне завдання за стандартними правилами. Неувага до формалізації завдання на розроблення ПЗ приводить до того, що створену систему неможливо здати в експлуатацію через величезну кількість зауважень, викликаних різночитаннями і незрозумілостями в постановці завдання.

Процес розроблення ПЗ за замовленням включає в себе питання документування, ціноутворення, способами

регламентування і контролю за ходом робіт, але основним результатом застосування технології є програма, що діє в заданому обчислювальному середовищі, добре налагоджена і документована, доступна для розуміння і розвитку в процесі супроводу.

На сьогодні замовник потребує комплексного вирішення своїх завдань, а це приводить до потреби у складних програмних системах. Щоб створити такі системи, недостатньо лише кваліфікованих програмістів, потрібні системні аналітики, які проаналізують замовлення і створять проект системи, та системні інженери, які зможуть реалізувати завдання як складну систему. Без стандартизованих підходів використання технологій неможливо виконати такий проект. Виходячи із вищезазначеного, можна сформулювати визначення технологій програмування.

Технологія програмування (ТП) - це сукупність методів і засобів розроблення (написання) програм та порядок застосування цих методів та засобів.

У англомовній літературі замість терміна «технологія програмування» використовується термін «методологія» (methodology) для позначення пов'язаних між собою методів та технік програмування [5].

Усі технології, які використовують програмісти, підпорядковуються основній **меті**:

- створити якісний продукт;
- вкластися в бюджет;
- дотриматися строків.

Розвиток технологій програмування

Ускладнення програмних систем та процесів розроблення викликало потребу створення механізмів керування та контролю процесів, тобто підключення технологічних підходів, що розпочалося із систематизації робіт. Наступними кроками стали встановлення технологічних маршрутів діяльності розробників ПЗ, визначення можливості їх автоматизації та виявлення ризиків, розроблення інструментів для автоматизації.

Перший етап автоматизації процесу створення програмного забезпечення був пов'язаний із підтримкою процесу програмування та систематизацією робіт. Розуміння, що інструменти підтримки процесу кодування є необхідною умовою підвищення продуктивності праці, але недостатньою для промислового розроблення програм. На цій стадії з'явились інструменти програмування із підтримкою написання коду. Після цього стали з'являтися засоби автоматизованого налагодження програм. У цей самий час (кінець 60-х рр. XX ст.) потреба створення великих за розміром програм із складними алгоритмами привела до розуміння необхідності розвитку теоретичної бази і запровадження поняття життєвого циклу ПЗ.

Відразу стали помітними проблемні місця виробництва програмних продуктів: нерозвиненість методології проектування та неможливість оцінити якість ПЗ лише у ході тестування. Фактично перша проблема викликає другу: нечітко поставлене завдання з програмування не дає параметрів для перевірки якості роботи. Однією з основних початкових вимог у процесі програмування стала вимога чіткої специфікації проекту. На основі даних із специфікації почали регламентувати процес проектування. Стали з'являтися певні технології розроблення програмних продуктів. Подальший розвиток методологій розроблення призвів до появи формалізованого менеджменту вимог у ході аналізу.

Сьогодні технології виробництва програмних продуктів автоматизують процес створення коду та дозволяють виконувати автоматичне тестування програм. Для етапів аналізу та проектування повністю автоматичних інструментів немає, хоча існують засоби автоматизованої підтримки і систематизації вимог та моделювання програмних систем.

Щоб зрозуміти, які технології використовувати для певного проекту, розібратися з методикою програмування, необхідно в першу чергу вивчити розвиток мов та підходів до програмування.

Життєвий цикл програмного забезпечення

У процесі створення ПЗ можна виділити 4 базових етапи/стадії (рис.2):

- **Специфікація –** визначення основних вимог.
- **Розроблення –** створення ПЗ відповідно до специфікацій.
- **Тестування** – перевірка ПЗ на відповідність вимогам клієнта.
- **Супровід/Модернізація –** розвиток ПЗ відповідно до змін потреб замовника.

Рисунок 2 – Схема життя програмного забезпечення

Перехід від ручних засобів розроблення ПЗ до промислового виробництва програм потребував розвитку теоретичних основ розроблення ПЗ. Постійна необхідність внесення змін у програми як спричинена помилками, так і розвитком вимог до них, є принциповою властивістю програмного забезпечення. Діяльність, пов'язана з рішенням широкого ряду завдань для постійного розвитку, отримала назву супроводу програмного забезпечення. Якщо зусилля, спрямовані на модернізацію ПЗ, перевищують вигоду від його використання, говорять про моральне старіння програм.

Оскільки розроблення та супровід ПЗ фактично є проектною діяльністю, частина ключових понять управління проектів знайшла широке застосування у програмній інженерії. Таким є і поняття життєвого циклу проекту (Project Lifecycle Management, PLM), що в програмній інженерії трансформувалось у поняття життєвого циклу програмного забезпечення.

Життєвий цикл програмного забезпечення - період часу, що починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації [6]. Цей цикл - процес побудови і розвитку ПЗ.

ГОСТ 34.601-90 [7] визначає життєвий цикл автоматизованої системи (АС) як сукупність взаємозв'язаних процесів створення та послідовної зміни стану АС, від формування вхідних вимог до неї до закінчення експлуатації та утилізації комплексу засобів автоматизації.

Як і будь-що модель, модель ЖЦ є абстракцією реального процесу, в якій відсутні деталі, несуттєві з точки зору призначення моделі.

Поняття ЖЦ виникло під впливом потреби у систематизації робіт у процесі розроблення ПЗ. Систематизація була першим етапом на шляху до автоматизації процесу розроблення ПЗ. Наступними кроками переходу до автоматизації процесу розроблення ПЗ були такі: встановлення

технологічних маршрутів діяльності розробників ПЗ, визначення можливості їх автоматизації та виявлення ризиків, розроблення інструментів для автоматизації.

Спочатку з'явилися інструменти підтримки розроблення програмного коду та налагодження програм (60-ті рр. XX ст.). Після усвідомлення недостатності таких засобів для істотного підвищення якості програм та створення інструментів керування процесом розроблення виникло поняття життєвого циклу ПЗ.

Виявлення закономірностей розвитку програмного забезпечення одразу показало нерозвиненість методик конструювання ПЗ та недостатність тестування для визначення якості програмних продуктів. Також на цьому етапі стало зрозуміло, що нечіткість завдання на створення ПЗ викликає більшість проблем розроблення та перевірки програм. У результаті виникли вимоги до постановки завдання, сформувалися підходи до управління вимогами на етапі аналізу та інструменти зв'язку вимог на етапах аналізу та реалізації.

Використання поняття життєвого циклу дозволяє обрати підходи, які найбільш ефективні для завдань певного етапу життя ПЗ. Залежно від особливостей процесів розроблення та супровіду програм існують різні моделі ЖЦ.

Використання певної моделі ЖЦ дозволяє визначитися з основними моментами процесу замовлення, розроблення та супроводу ПЗ навіть недосвідченому програмісту. Також використання моделей дозволяє чітко зрозуміти, в який період переходити від версії до версії, які дії з удосконалення виконувати, на якому етапі. Знання про закономірності розвитку програмного продукту, які відбиваються в обраній моделі ЖЦ, дозволяють отримати надійні орієнтири для планування процесу розроблення та супроводу ПЗ, економно витрачати ресурси та підвищувати якість управління усіма процесами.

Також моделі життєвого циклу є основою знань технологій програмування та інструментарію, що їх підтримує. Будь-що технологія базується на певних уявленнях про життєвий цикл та організує свої методи та інструменти навколо фаз та етапів ЖЦ.

Розвиток методологій програмування у 70-х рр. XX ст. привів до формування потреби вивчення життєвого циклу ПЗ. До цього часу моделі ЖЦ розвиваються і модифікуються, уточнюючи та доповнюючи дві базові моделі – каскадну та ітеративну. Ці зміни обумовлені потребою організаційної та технологічної підтримки проектів з розроблення ПЗ.

Каскадна модель (waterflow model)

Каскадна модель ЖЦ ПЗ виникла для задоволення потреби у систематизації робіт ще на ранніх етапах розроблення програм. Згідно з цією моделлю програмні системи проходять в своєму розвитку дві фази:

- розроблення;
- супровід.

Фази розбиваються на ряд етапів (рис. 3).

Каскадна модель передбачає послідовне виконання всіх етапів проекту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі.

Розроблення починається з ідентифікації потреби в новому додатку, а закінчується передачею продукту розроблення в експлуатацію. Усі етапи розроблення програмного забезпечення регламентуються стандартами підприємства та державним стандартом ГОСТ 34.601-90 [7].

Першим етапом фази розроблення є ***специфікація*** *(Requirements Specification)* – постановка завдання і визначення вимог. На етапі постановки завдання замовник спільно з розробниками приймають рішення про створення системи. Визначення вимог включає опис загального контексту задачі, очікуваних функцій системи та її обмежень. Особливо важливим є цей етап для нетрадиційних додатків. У разі позитивного рішення починається аналіз системи відповідно до вимог. Розробники програмного забезпечення намагаються осмислити висунуті замовником вимоги і зафіксувати їх у вигляді специфікацій системи. Призначення специфікацій – описувати зовнішню поведінку системи, а не її внутрішню організацію.

Перш ніж розпочати створювати проект за специфікаціями, вимоги повинні бути ретельно перевірені на відповідність вихідним цілям, повноту, сумісність (несуперечність) та однозначність. Завдання етапу аналізу полягає в тому, щоб вибудувати опис програми у вигляді логічної системи, зрозумілої як для замовника, майбутніх користувачів, так і для виконавців проекту. На етапі специфікації обов'язково формується технічне завдання на розроблення ПЗ [8].

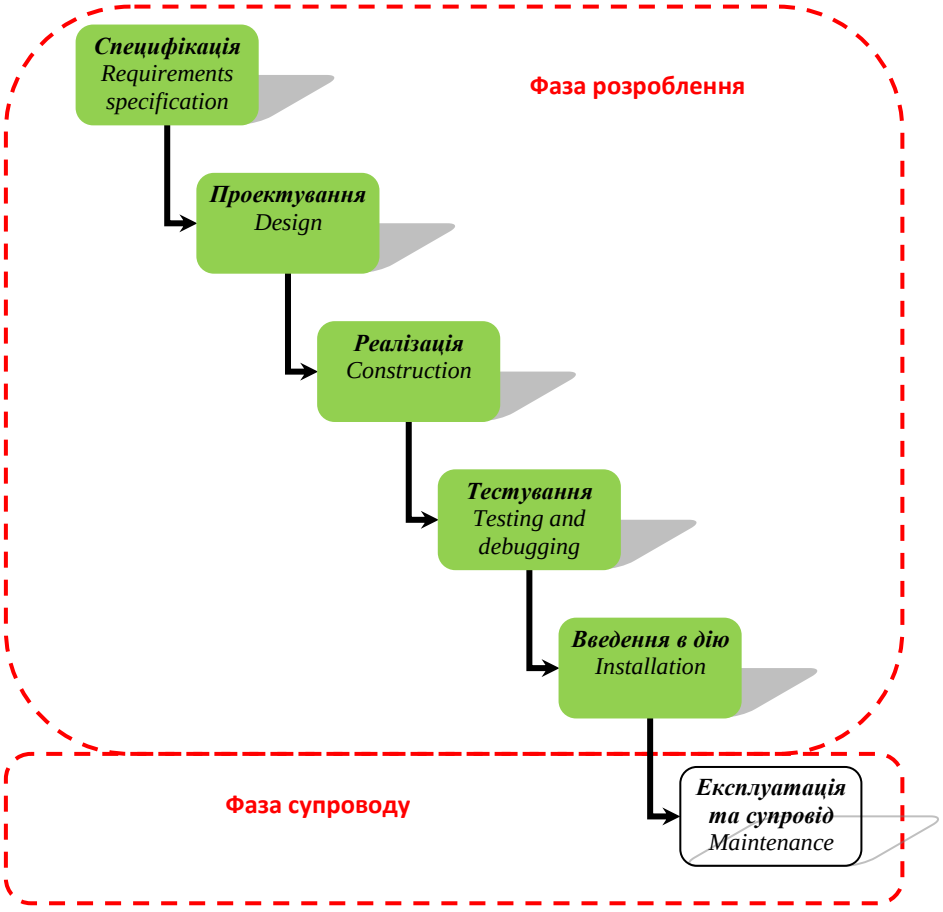


Рисунок 3 – Каскадна модель ЖЦ

Розроблення проектних рішень, що відповідають на питання, як повинна бути реалізована система, щоб вона могла задовольняти визначені вимоги, виконується на етапі ***проектування*** *(Design)*. Оскільки складність системи в цілому може бути дуже великою, головним завданням цього етапу є послідовна декомпозиція системи до рівня очевидно реалізованих модулів або процедур. Результати виконання цього етапу оформлюютья як технічний проект, вимоги до документів якого встановлені стандартом ГОСТ 34.201-89 [9].

На наступному етапі ***реалізації*** *(Construction)*, або кодування, кожен з цих модулів програмується на найбільш підходящій для даного застосування мові. З точки зору автоматизації цей етап традиційно є найбільш розвиненим.

У каскадній моделі фаза розроблення закінчується етапом ***тестування*** *(Testing and debugging),* автономного і комплексного, та ***передачею системи в експлуатацію*** *(Installation).*

Фаза ***експлуатації та супроводу*** включає в себе всю діяльність щодо забезпечення нормального функціонування програмних систем, у тому числі фіксування розкритих під час виконання програм помилок, пошук їх причин та виправлення, підвищення експлуатаційних характеристик системи, адаптацію системи до довкілля, а також за необхідності і більш суттєві роботи з удосконалення системи. Фактично відбувається еволюція системи. У ряді випадків на дану фазу припадає більша частина коштів, що витрачаються в процесі життєвого циклу програмного забезпечення.

Зрозуміло, що увага програмістів до тих чи інших етапів розроблення залежить від конкретного проекту. Часто розробнику немає необхідності проходити через усі етапи, наприклад, якщо створюється невелика, добре зрозуміла програма із чітко поставленою метою.

Стисла характеристика:

- *фіксований набір стадій;*
- *кажна стадія закінчується документованим результатом;*

- *наступна стадія починається лише після закінчення попередньої.*

Недоліки:

- *негнучкість;*
- *фаза повинна бути завершена до переходу до наступної;*
- *набір фаз фіксований;*
- *важко реагувати на зміни вимог.*

***Використання**: там, де вимоги добре зрозумілі та стабільні.*

Ітеративна модель (Iterative and incremental development)

Каскадна модель життєвого циклу є ідеальною, оскільки лише дуже прості завдання проходять усі етапи без будь-яких ітерацій (повернень на попередні кроки процесу). Наприклад, при програмуванні може виявитися, що реалізація деякої функції неефективна і вступає у протиріччя з вимогами до продуктивності системи. У цьому випадку потрібні зміни проекту, а можливо, і переробка специфікацій. Для врахування повторюваності етапів процесу розроблення створювались альтернативи каскадної моделі.

Із таких альтернатив утворилася **ітеративна модель.** Ця модель передбачає розбиття життєвого циклу проекту на послідовність ітерацій, кожна з яких нагадує "міні-проект" з усіма фазами життєвого циклу.

Класична ітераційна модель абсолютизує можливість повернень на попередні етапи (рис.4). Ця обставина відбиває істотний аспект програмних розробок: прагнення заздалегідь передбачати всі ситуації використання системи та неможливість у переважній більшості випадків досягти цього. Усі традиційні технології програмування спрямовані лише на те, щоб мінімізувати повернення. Але суть від цього не змінюється: при поверненні завжди доводиться повторювати побудову того, що вже вважалося готовим.

Мета кожної ітерації в розробленні ПЗ – отримання працюючої версії програмної системи, що включає

функціональність, визначену інтегрованим змістом усіх попередніх і поточної ітерації. Результат фінальної ітерації містить усю необхідну функціональність продукту. Таким чином, із завершенням кожної ітерації продукт розвивається інкрементально (нарощує функціональність).

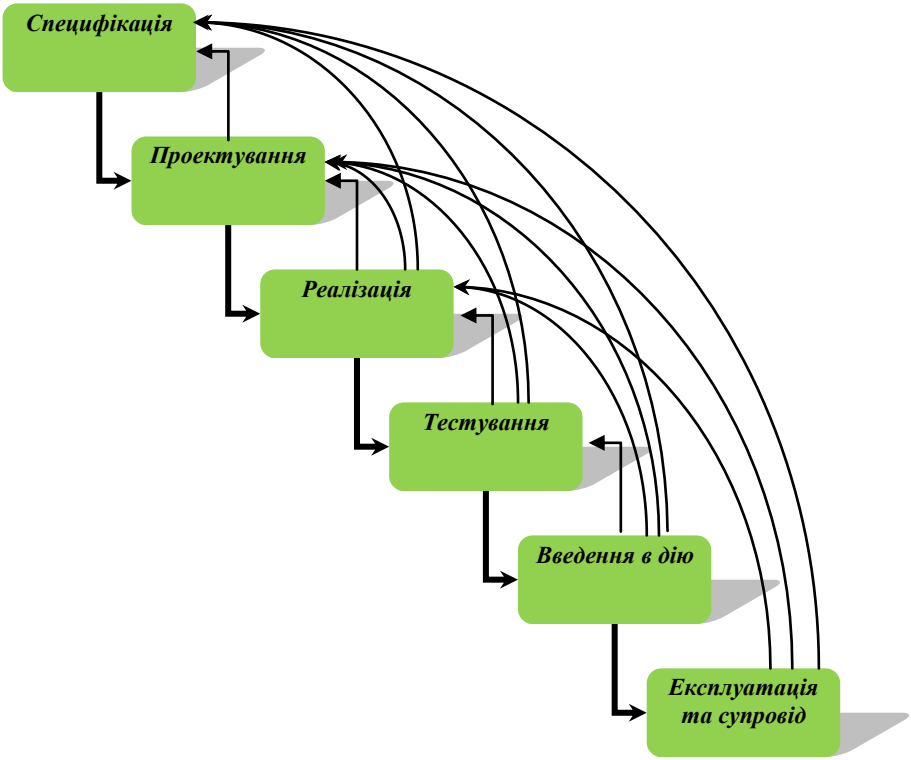


Рисунок 4 – Класична ітеративна модель

З точки зору структури життєвого циклу та процесу розроблення така модель є ітеративною (iterative) (рис.5). З точки зору розвитку продукту - інкрементальною (incremental). Досвід індустрії розроблення ПЗ показує, що неможливо розглядати кожен з цих поглядів ізольовано [10]. Тому цю модель часто називають моделлю ітеративного та інкрементного розроблення (Iterative and incremental development, IID) [11].

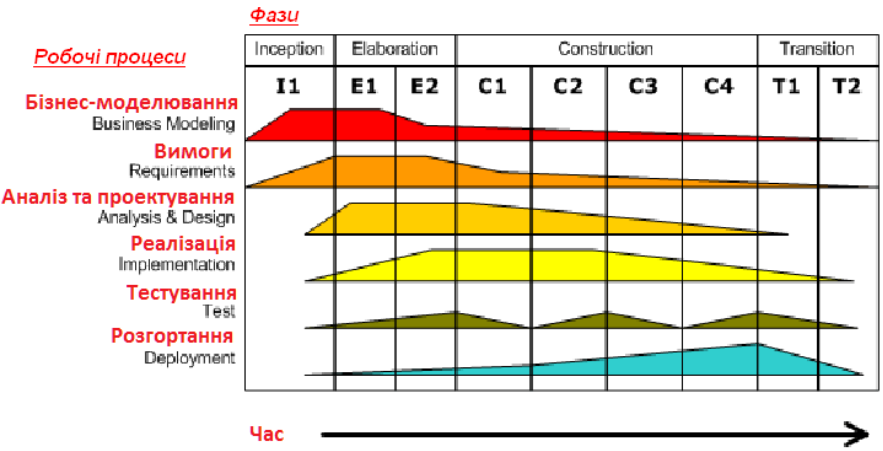


Рисунок 6 – Ітеративне та інкрементне розроблення програм

Різні варіанти ітераційного підходу реалізовані в більшості сучасних методологій розроблення (RUP, MSF, XP).

Стисла характеристика:

- *стадії повторюються неодноразово.*

Недоліки:

- *система часто погано структурована, проект «не прозорий»;*
- *після уточнення вимог відкидається частина раніше виконаної роботи;*
- *потрібні засоби для швидкого розроблення.*

Використання: *підходить для малих та середніх проектів.*

Спіральна модель

Найбільш відомим і поширеним варіантом ітераційної моделі є спіральна модель (рис.7), що була вперше сформульована Баррі Боемом (Barry Boehm) у 1986 році [12]. Відмінною особливістю цієї моделі є спеціальна увага ризикам, що впливає на організацію життєвого циклу.

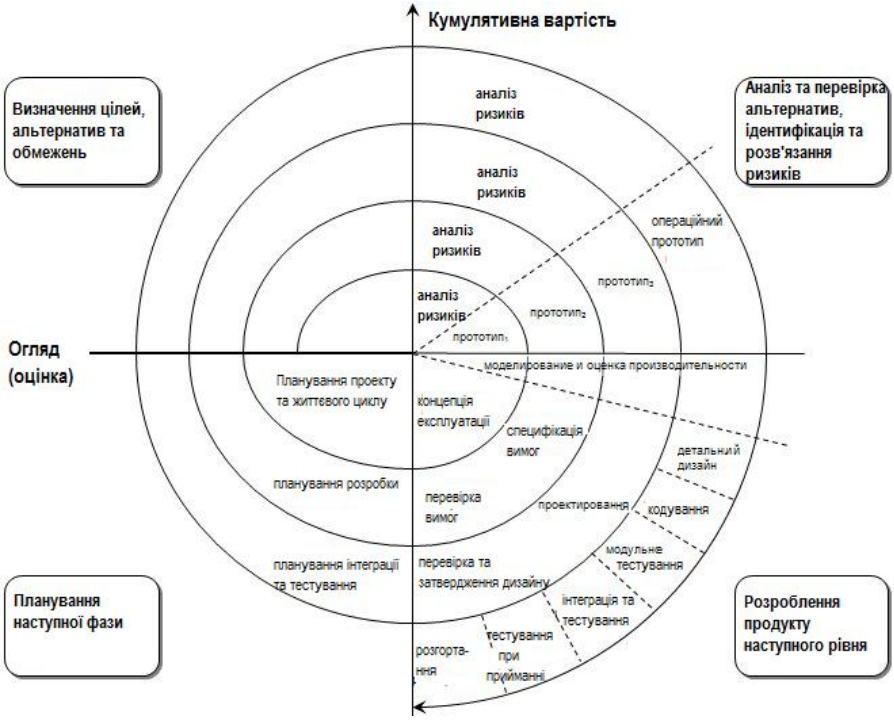


Рисунок 7 – Спіральна модель ЖЦ

У спіральній моделі розроблення програми має вигляд серії послідовних ітерацій. На перших етапах уточнюються специфікації продукту, на наступних - додаються нові можливості і функції. Мета цієї моделі - по закінченні кожної ітерації заново здійснити оцінку ризиків продовження робіт.

На кожному витку спіралі виконується створення чергової версії продукту, уточнюються вимоги проекту, визначається його якість і плануються роботи наступного витка. Особлива увага приділяється початковим етапам розроблення – аналізу і проектуванню, де реалізованість тих чи інших технічних рішень перевіряється і обґрунтовується за допомогою створення прототипів (макетування).

Завдяки ітеративній природі спіральна модель допускає коригування у ході роботи, що сприяє поліпшенню продукту.

При великому числі ітерацій розроблення за цією моделлю потребує глибокої автоматизації усіх процесів, інакше вона стає неефективною. На практиці у замовників і користувачів іноді виникає відчуття нестабільності продукту, оскільки вони не встигають стежити за швидкими змінами в ньому.

Спіральна модель розроблення ПЗ вимагає визначення ключових контрольних точок проекту - milestones. Виділяють такі основні контрольні точки [12]:

- *Concept of Operations (COO)* – концепція використання системи;
- *Life Cycle Objectives (LCO)* – цілі та зміст життєвого циклу;
- *Life Cycle Architecture (LCA)* – архітектура життєвого циклу;
- *Initial Operational Capability (IOC)* – перша версія ПЗ, що перевіряється у ході дослідницької експлуатації;
- *Final Operational Capability (FOC)* – готове ПЗ, що експлуатується в реальних умовах.

Аналіз у ключових точках особливо актуальний для менеджерів і лідерів проектів, які відстежують хід виконання проекту і планують подальші роботи.

Стисла характеристика:

- *проект має контрольні точки – milestones;*
- *на кожному витку спіралі створюється прототип На виході – продукт, що відповідає вимогам користувачів.*

Переваги:

- *ранній аналіз можливостей повторного використання;*
- *наявні механізми досягнення параметрів якості;*
- *модель дозволяє контролювати джерела проектних робіт і відповідних витрат;*
- *модель дозволяє вирішувати інтегровані завдання системного розроблення, які охоплюють програмну і апаратну складові створюваного продукту.*

Недоліки:

- *після уточнення вимог відкидається частина раніше виконаної роботи;*
- *потрібні засоби для швидкого розроблення.*

Використання*: підходить для малих та середніх проектів.*

Незважаючи на розвиток теоретичної бази, стадія комплексної автоматизації технологій програмування стала можливою лише при відповідному рівні розвитку техніки. Істотно вплинуло на перехід до комплексної автоматизації усвідомлення того, що не можна розвивати промислове програмування без підтримки технологічних функцій на всіх етапах життя програм. На початку 90-х рр. XX ст. з'явилася CASE-технологія (Computer Added Software Engineering – сукупність інструментів та методів програмної інженерії проектування ПЗ для забезпечення високої якості, мінімальної кількості помилок та спрощення проектування), що об’єднувала системи комплексної автоматизації підтримки розроблення і супроводу програм.

Аналіз моделей ЖЦ ПЗ показує, що саме програмування не є основним видом діяльності колективу, зайнятого промисловими розробленнями. Багато досліджень віддають на фазу програмування не більше 15-20% часу, витраченого на розроблення (супровід може бути нескінченним).

Питання для самоконтролю

1. Що таке технології програмування?
2. Як розвивалися стилі програмування?
3. Що таке життєвий цикл ПЗ?
4. Які моделі життєвого циклу ПЗ вам відомі?
5. Що виконується під час етапів фази розроблення ПЗ?
6. Які моделі життєвого циклу дозволяють оперативно реагувати на зміну вимог до ПЗ?
7. Які характеристики спіральної моделі життєвого циклу є її перевагою? Поясніть.
8. Які ключові точки проекту розроблення ПЗ вам відомі?
9. Які параметри характеризують якість програмного продукту?
10. Що модель життєвого циклу ПЗ має фіксований набір стадій? Яким чином була модифікована ця модель?
11. Що таке CASE-засоби? Як вони розвивались?

ТЕМА 2. Планування та управління процесом розроблення та супроводу програмного забезпечення

Постановка завдання

Під час перших зустрічей щодо замовлення на розроблення ПЗ замовник дуже приблизно уявляє, що йому потрібно. Як правило, він надає кілька сторінок тексту завдання і одразу просить оцінити час виконання замовлення та його вартість. Без чіткого визначення процесів, для автоматизації яких буде використовуватись ПЗ, неможливо навіть приблизно оцінити обсяг робіт. Приблизна оцінка з мінімумом вхідної інформації може призвести до помилки в кілька разів, що негативно відіб'ється на точності визначення строків виконання та вартості робіт.

Потреби (needs) – відображають проблеми бізнесу, персоналій або процесу, що повинні співвідноситися з використанням або придбанням системи [13].

Щоб тримати уявлення про можливі обсяги робіт, потрібно пропонувати замовнику надати або розробити технічне завдання. Завдяки цьому системні аналітики зможуть розібратися в задачі, за допомогою інструментальних засобів виконати декомпозицію системи на компоненти, приблизно визначити обсяги цих компонентів і відповідно час їх реалізації. Ця початкова стадія ЖЦ ПЗ є "оцінкою здійсненності".

Постановка завдання – найбільш творча частина ЖЦ ПЗ. Потрібно описати поведінку розроблюваної системи. Ця система отримує якісь сигнали з її оточення, тому треба описати поведінку оточення, але оточення само залежить і змінюється під впливом системи, її сигналів, особливо аварійних.

Вирішують це протиріччя ітераційно, поетапно уточнюючи поведінку як системи, так і її оточення. Для відповідальних систем замовник може запропонувати розробити імітаційну модель системи та оточення, що є досить складною.

У поставленому завданні замовник визначає вимоги до створюваної системи, які повинні задовольняти потреби користувачів і бути зрозумілими для розробників.

Вимога (Requirement) – умова або можливість, що визначена користувачем для вирішення проблеми або досягнення мети, та якій повинна відповідати або якою повинна володіти система чи її компонент, щоб задовольняти умови контракту, стандарту, специфікації або іншого формально репрезентованого документа [14]. Вимоги поділяються на:

- ***вимоги користувача (User Requirements)*** – описують цілі/задачі користувачів системи, які повинні досягатися/виконуватися користувачами за допомогою створюваної програмної системи [13];
- ***функціональні вимоги (Functional Requirements)*** – вимоги, що конкретизують функції, які система або її компонент повинен виконувати [14];
- ***програмні вимоги (Software Requirements)*** – вимоги до створюваної системи, зрозумілі користувачами (замовниками) і розробниками (виконавцями) стосовно того, що робитиме система і чого від неї не варто чекати.

Досвід показує, що оцінка складності системи, що є сумою оцінок її компонентів, отриманих у результаті декомпозиції, значно точніша, ніж первісна оцінка системи в цілому. Використання засобів формалізації результатів аналізу для їх документального оформлення також підвищує якість початкового опису вимог до системи.

Коли говорять про "формалізацію постановки завдання", мають на увазі розроблення послідовності моделей, кожна з яких описує систему та її оточення з різних точок зору із поступовою деталізацією. Важливо, щоб усі уявлення про систему, отримані в різних моделях, збиралися в єдиному репозитарії (деякій спеціалізованій базі даних). Цей репозитарій полегшить наскрізне проектування, при якому кожна наступна модель використовує результати попередньої і ніяк їм не суперечить. Відповідно всі можливі перевірки повинні бути наскрізними.

Для якісного визначення вимог до ПЗ потрібно спочатку провести аналіз та сформувати їх специфікації.

Аналіз вимог *(Requirements Analysis)* – трансформація інформації, отриманої від користувачів (та інших зацікавлених осіб) у чітко та однозначно визначені програмні вимоги, що передаються інженерам для реалізації у програмному коді. Аналіз вимог включає:

- виявлення і розв'язання конфліктів між вимогами;
- визначення меж задачі, що вирішується створюваним програмним забезпеченням; у загальному випадку – визначення меж (Scope) і змісту програмного проекту;
- деталізацію системних вимог для встановлення програмних вимог.

Специфікація *(Specification)* – документ, що в закінченій, точній і перевіреній формі описує вимоги, проект, поведінку або інші характеристики компонента або системи, а також процедури, спрямовані на визначення того, чи задовольняються описані характеристики [14]. Для опису комплексних проектів (у частині вимог) використовують три основні специфікації:

- визначення системи (System Definition), або специфікація вимог користувачів (User Requirements Specification);
- системних вимог (System Requirements);
- програмних вимог (Software Requirements).

Специфікація програмних вимог *(Software Requirements Specification – SRS)* встановлює основні угоди між користувачами (замовниками) і розробниками (виконавцями) стосовно того, що робитиме система і чого від неї не варто чекати. Цей документ може включати процедури перевірки створеного ПЗ на відповідність вимогам, що висуваються (у т.ч. плани тестування), описи характеристик стосовно якості та методів його оцінювання, питань безпеки тощо [13].

Специфікація вимог користувачів *(User Requirements Specification)* або *концепція (concept <of operation>)* визначає високорівневі вимоги, часто – стратегічні цілі, для досягнення яких створюється програмна система. Важливо, що цей

документ описує вимоги до системи з позицій предметної області – домену [13].

Специфікація системних вимог *(System Requirements)* – описує програмну систему в контексті системної інженерії. Зокрема високорівневі вимоги до програмного забезпечення, що містить кілька або багато взаємозв'язаних підсистем і застосувань. При цьому система може бути як цілком програмною, так і містити програмні та апаратні компоненти. У загальному випадку до складу системи може входити персонал, що виконує певні функції системи, наприклад, авторизацію виконання певних операцій з використанням програмно-апаратних підсистем [13].

При постановці завдання потрібно, щоб програмні вимоги були зрозумілі, зв'язки між ними прозорі, а зміст специфікації не допускав різночитань та інтерпретацій, через які програмний продукт не буде відповідати потребам зацікавлених осіб. Тому потрібні інструменти управління вимогами.

Управління вимогами *(Requirements Management)* – діяльність, виконання якої забезпечує опис вимог, відстежування їх змін, перевірки на несуперечливість і на порушення наперед визначених правил [14].

Від вхідної інформації про майбутній програмний продукт залежить те, яку методологію буде обрано в проекті з розроблення ПЗ. Методологій багато: і дуже формалізованих, і тих, що дають творчу свободу програмістам. Вибір методології обумовлюється досвідом керівника групи розробників та умовами, які встановлюють замовники до документування етапів робіт. У роботі [5] методології розроблення ПЗ класифікуються за кількістю виконавців та критичністю проекту. Чим більше виконавців та/або вища критичність, тим більш формальна та регульована методика потрібна.

Розроблення ПЗ як проектна діяльність

Діяльність під час розроблення ПЗ, як і будь-що, складається з виконання операцій і проектів. Ті й інші мають багато спільного, наприклад, виконуються людьми та на їх виконання виділяються обмежені ресурси.

Головна відмінність операцій від проектів полягає в тому, що операції виконуються постійно і повторюються, тоді як проект тимчасовий і унікальний. Виходячи з цього, ***проект*** визначається як тимчасове зусилля, розпочате для створення унікального продукту чи послуги. «Тимчасове» означає, що кожен проект має точно визначені дати початку та закінчення. Говорячи про унікальність продукту, ми маємо на увазі, що вони мають помітні відмінності від усіх аналогічних продуктів або послуг. Таким чином, розроблення ПЗ відповідає визначенню проекту і для організації цього процесу можна застосовувати методи та інструментарій управління проектами. Наприклад, розроблення веб-сайту є проектом, тоді як підтримка його впродовж тривалого часу – це операційна діяльність.

У кожного проекту є чітко визначені початок і кінець. Кінець проекту настає разом із досягненням усіх його цілей або коли стає зрозумілим, що ці цілі не будуть або не можуть бути досягнуті. Тимчасовість не означає короткостроковість проекту – розроблення складної програмної системи може тривати кілька років, хоча, як правило проекти мають обмежені часові рамки для створення ПЗ, оскільки сприятлива для них ситуація на ринку складається на обмежений час. Крім того, проектна команда після його закінчення розпадається, а її члени переходять в інші проекти.

Проект дуже часто плутають із програмою, тобто координованим управлінням групою проектів всередині однієї організації. Управління відразу декількома проектами скоординоване для того, щоб отримати вигоду, яку неможливо одержати від окремого управління кожним із них.

Проект виконується для досягнення певного результату в певні терміни і за певні гроші. Це тріо *часу (time), бюджету*

(cost) і обсягу робіт (scope) часто називають ***проектним,*** або залізним, ***трикутником*** (рис. 8) [15], оскільки при внесенні змін в один із цих елементів змінюються інші. Хоча для проекту однаковою мірою важливі всі три елементи, як правило, тільки один із них залежно від пріоритетів має найбільший вплив на інші.

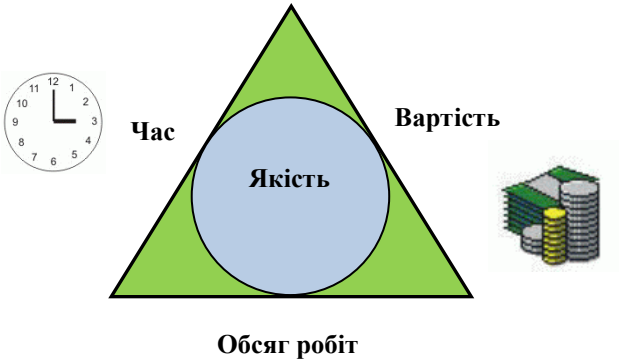


Рисунок 8 – Проектний трикутник

Те, як зміни у плані впливають на інші сторони трикутника, залежить від обставин і специфіки проекту. У деяких випадках зменшення терміну збільшує вартість, а в інших - зменшує.

Якість (quality) – це четвертий елемент проектного трикутника. Вона знаходиться у центрі, і будь-що зміна сторін впливає на неї.

Наприклад, якщо керівникові вдалося знайти додатковий час у розкладі, то можна збільшити обсяг робіт, додавши завдання і збільшивши тривалість проекту. Ці додаткові завдання і час дозволяють домогтися більш високого рівня якості проекту і виробленого продукту.

Якщо ж потрібно знизити витрати, щоб вкластися в бюджет, можливо, знадобиться зменшити обсяг робіт, прибравши деякі із завдань або зменшивши їх тривалість. Зі зменшеним обсягом робіт у проекті буде менше шансів вийти на

необхідний рівень якості, тому зниження витрат може призвести до погіршення якості проекту.

План проекту складається для того, щоб визначити, за допомогою яких робіт буде досягатися результат проекту, які люди й устаткування потрібні для виконання цих робіт і в що час ці люди й устаткування будуть зайняті роботою в проекті. Тому проектний план містить три основні елементи: завдання, ресурси і призначення.

Завдання (Tasks)

Завданням є робота, здійснювана у рамках проекту для досягнення певного результату. Наприклад, у проекті створення програми для автоматизації формування документації підприємства завданням буде Створення шаблонів. Оскільки, як правило, проект містить багато завдань, то для зручності відстеження плану їх об'єднують у групи, або фази. Сукупність фаз проекту називається його життєвим циклом.

Фази (Summary tasks)

Фаза проекту складається з одного або кількох завдань, у результаті виконання яких досягається один або кілька основних результатів проекту. Таким чином, результати, досягнуті завдяки виконанню кожної із задач, що входять у фазу, формують її результат. Фази можуть складатися як із завдань, так і з інших фаз.

Якщо для досягнення результатів завдання потрібно виконати тільки її, то для досягнення результату фази потрібно виконати групу інших завдань.

При плануванні робіт потрібно пам'ятати, що чим детальніше буде план проекту, тим точніше він буде відповідати реальній ситуації. У тих випадках, де це можливо, варто розбивати великі завдання на підзадачі (перетворювати завдання в фази). Формальними критеріями, які показують, що завдання можна розбити на підзадачі, є тривалість (завдання рідко тривають довше 2-3 днів) і велике число задіяних виконавців (як правило, якщо над вирішенням завдання працюють більше 2-

3 осіб, то кожен вирішує свою власну задачу, яку можна окремо врахувати у плані проекту).

Завершальні завдання

Завдання, у результаті виконання яких досягаються проміжні цілі, називаються завершальними завданнями.

Тривалість (Duration) і трудовитрати (Work)

Тривалість завдання - це період робочого часу, що необхідний для того, щоб виконати її.

Тривалість може не відповідати трудовитратам співробітника, що займається завданням. Тривалість відповідає часу, через що буде отриманий результат задачі, а трудовитрати– часу, витраченому співробітниками на отримання результату.

Залежності (Dependencies) та зв'язки (Links)

Завдання у плані проекту взаємозв'язані. Наприклад, часто одна задача не може початися, поки не закінчена інша (тестування модуля не може початися раніше написання його коду).

На плані проекту залежності позначаються за допомогою зв'язків. Обидва ці терміни – залежність і зв'язок – використовуються з одним і тим самим змістом і позначають логіку, певну послідовність робіт у плані проекту.

Ролі (Roles) і ресурси (Resources)

Під ресурсами розуміють співробітників та обладнання, які необхідні для виконання проектних завдань.

Кожен співробітник, що бере участь у проекті, отримує певну роль відповідно до своєї кваліфікації, вимог проекту та регламентів, що діють в організації. Наприклад, в одному проекті співробітник може виступати в ролі архітектора додатків, а в іншому той самий працівник може бути задіяний у ролі програміста.

При складанні списку ресурсів часто використовується рольове планування. Наприклад, спочатку визначається, що для виконання робіт потрібні три програмісти і один менеджер, а потім, коли план проекту затверджений, вибираються конкретні співробітники для участі в цих ролях.

Призначення (Assignments)

Призначення - це зв'язок певної задачі і ресурсів, необхідних для її виконання. При цьому на одну задачу можуть бути призначені декілька ресурсів, як матеріальних, так і нематеріальних.

Призначення об'єднують у плані ресурси і завдання, роблячи план цілісним. Завдяки призначенням вирішується цілий ряд завдань планування. По-перше, визначаються відповідальні за виконання завдань. По-друге, коли визначені завдання, за які відповідає ресурс, можна розрахувати загальний обсяг часу, що витрачається ним на проект, а отже, його вартість для проекту. По-третє, визначивши вартість участі всіх ресурсів у проекті, можна підрахувати його загальну вартість. Нарешті, призначаючи ресурси на завдання, можна скорочувати термін виконання робіт, виділяючи на них більше ресурсів і тим самим скорочуючи загальну тривалість проекту.

Основні форми планів робіт

Результатом фази оцінки здійсненності є детальна специфікація (технічне завдання), план роботи та оцінка вартості. Найбільш традиційними формами плану можна вважати *мережевий графік та діаграму Ганта (Gantt diagram).* Діаграма Ганта дозволяє визначити, що частина робіт повинна бути виконана у кожен момент часу, тому її частіше використовують керівники проектів, тоді як технічні фахівці віддають перевагу мережевим графікам, оскільки вони містять інформацію про тривалість кожного виду роботи.

Мережевий графік подається у вигляді орієнтованого графа з двома виділеними вершинами - початок і кінець роботи. Вершинами графа є події, що відповідають пунктам плану, а

ребрами - роботи. Події повинні виражатися дієсловами доконаного виду: "тексти програм передані в базу вхідних даних", "усі тести пропущені", "група оцінки якості дала позитивний висновок" і т.д. Ребра навантажуються оцінками тривалості робіт, наприклад, у днях або тижнях.

Переваги використання мережевого графіка при плануванні робіт полягають у такому:

- на ньому чітко видно залежність робіт одна від одної;
- на основі мережевого графіка можна обчислити тривалість усієї роботи;
- користуючись результатами цих розрахунків, можна оптимізувати тривалість і витрати на роботу.

Тривалість робіт обчислюється так: підсумовуємо тривалості робіт за всіма можливими шляхами в графі. Той шлях від початку до кінця, що є найдовшим, оголошується ***критичним,*** оскільки затримка будь-якої роботи, що лежить на цьому шляху, приводить до затримки всієї роботи в цілому. Зрозуміло, що критичних шляхів може бути декілька.

Ще однією популярною формою графічного представлення плану робіт є діаграма Ганта. ***Діаграма Ганта*** являє собою прямокутник: зліва направо рівномірно відлічуються періоди часу (тижні, місяці), зверху вниз перераховуються роботи, причому кожна робота представляється у вигляді відрізка, початок і кінець якого розміщуються у відповідному періоді.

Переваги використання діаграм Ганта при плануванні робіт полягають у такому:

- чітко видно, що виконується кожного тижня;
- зручно позначати кількість виконавців роботи.

Приклад використання мережевого графіка та діаграми Ганта

А. М. Терехов у своєму курсі [16] наводить класичну задачу планування: компанія, що працює у нормальному режимі з повним навантаженням, отримала замовлення на розроблення

ПЗ, що хоче виконати. Для цієї задачі побудуємо мережевий графік та діаграму Ганта.

Перелічимо назви подій, тобто вузлів у графі, мережевого графіку(рис.9):

1. Початок роботи.
2. Колектив сформований, робочі місця підготовлені.
3. Проектування ПЗ завершено.
4. Програмування завершено.
5. Комплексне налагодження завершене.
6. Обладнання закуплене.
7. Група з документування отримала опис проекту та необхідні пояснення від проектувальників.
8. Група із документування отримала опис ПЗ, розроблення проектної документації завершене.
9. Група із документування отримала всю необхідну інформацію про користувацькі інтерфейси, розроблення програмної документації завершене.
10. Група оцінки якості (Quality Assurance - QA) розробила тести.
11. Група QA оцінила проект позитивно.
12. Група QA завершила автономне тестування.
13. Група QA завершила комплексне тестування, отримала всю документацію та діючий варіант системи.
14. Перевірка якості завершена.
15. Закінчення роботи з розроблення ПЗ.

Під кожним ребром графа записана планована тривалість роботи (в тижнях).

Критичними шляхами є шляхи *1-6-2-3-4-5-9-13-14-15* та *1-6-2-3-4-5-12-13-14-15*, тобто вся робота не може бути виконана швидше ніж за 18,3 тижня. Зрозуміло, що з точки зору оптимального завантаження колективу було б краще, щоб усі шляхи в графі від початку до кінця мали приблизно однакову тривалість для того, щоб якось зменшити довжину критичного шляху. Наприклад, є спокуса примусити групу QA проводити навіть початкове тестування, зменшивши навантаження на

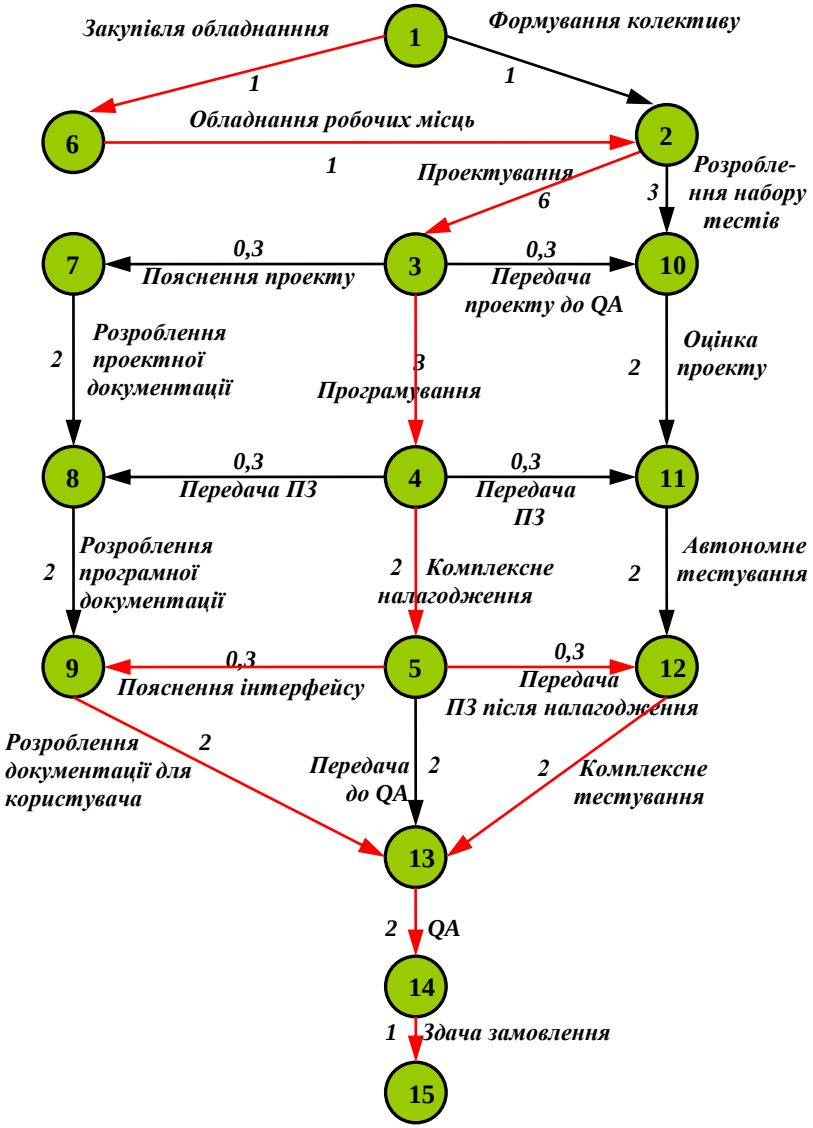


Рисунок 9 – Мережевий графік робіт при розробленні ПЗ

програмістів, робота яких знаходиться на критичному шляху. Але тоді важко визначити межі відповідальності, програмісти можуть для дотримання строків видавати «сирий» результат, і повернення на доопрацювання затягне проект ще більше. У реальних проектах, де робіт дуже багато, можна шляхом перерозподілу робіт покращувати мережевий графік.

Якщо уважно розглянути цей приклад, можна помітити, що невдало сплановані роботи між подіями 1, 2, 6. Колектив сформований за один тиждень, а робочі місця ще не готові. Група із документування має працювати на шість тижнів пізніше від проектувальників, а група QA має майже тритижневу перерву перед завершенням проектування та ін.

Ці проблеми складні, кожна компанія вирішує їх по-своєму, наприклад, очевидним рішенням є використання однієї і тієї самої групи QA або групи із документування для декількох груп розробників.

Для порівняння з мережевим графіком сформуємо діаграму Ганта для того самого прикладу (рис. 10).

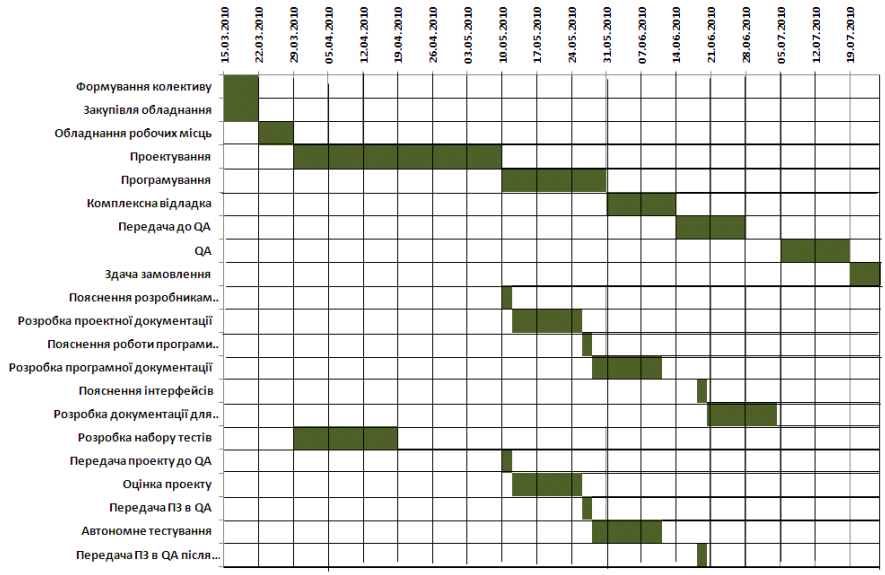


Рисунок 10 – Діаграма Ганта

Якщо в мережевому графіку наочно видно, як залежать роботи одна від одної, (наприклад, робота 12-13 може початися тільки після завершення робіт 5-12 і 11-12), то діаграма Ганта показує, що відбувається кожного конкретного тижня. Наприклад, видно, що група QA має перерву в роботі майже 2 тижні між роботами 11-12 (автономне тестування) та 12-13 (комплексне тестування). Саме тому керівники проектів віддають перевагу діаграмі Ганта: у кінці кожного тижня видно, які роботи проводились, що повинно закінчитися, а що розпочатися. Також на діаграмі Ганта прийнято над кожним відрізком роботи зазначати, скільки співробітників бере участь у цій роботі (на мережевому графіку це можливо, але не дуже зручно, оскільки треба зазначати ще й тривалість роботи). Ця властивість важлива для керівника проекту.

Технічні менеджери частіше використовують мережеві графіки, оскільки їм важливіша інформація про взаємну залежність робіт. Окрім цього за мережевим графіком можна перераховувати критичні шляхи, що доводиться робити досить часто.

Керування та організація робіт

Особливості процесу керування визначаються в першу чергу набором методів та технологій, які будуть використовуватися під час розроблення ПЗ. Цей вибір виконує керівник проекту, і обрана ним методологія в першу чергу визначається його особистим досвідом, а потім вимогами замовника.

У роботі А.Коуберна [5] формулюються **принципи вибору методології розроблення**. *Перший принцип* визначає, що велика за розміром методологія потрібна великим командам розробників. Розмір методології визначається кількістю елементів керування у проекті, до яких належать види діяльності, застосовані стандарти, створювані артефакти, показники якості й інші характеристики. На рис.11 наведені складові методології.

Коуберн так пояснює зміст цих складових:

1. ***Ролі*** *(Roles)* – визначення посади, що наводиться в оголошенні про роботу (наприклад, менеджер проекту, тестувальник, розробник інтерфейсу та ін.).

2. ***Уміння*** *(Skills)* – навички, які повинні мати претенденти на посаду.

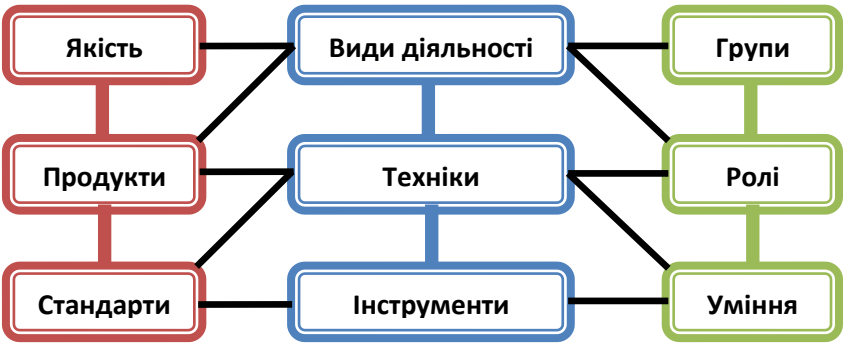


Рисунок 11 – Складові методології

3. ***Групи*** *(Teams)* – розподіл розробників за групами із визначенням їх ролей (наприклад, дизайнери, розробники документації, тестери та ін.).

4. ***Інструменти*** *(Tools)* – інструменти, які використовують розробники, відповідно до обраних технік та стандартів.

5. ***Техніки*** *(Techniques)* – техніки, які розробники використовують (наприклад, Java-програмування, моделювання варіантів використання та ін.).

6. ***Види діяльності*** *(Activities)* – зустрічі, відгуки, ключові точки (milestones) та інша діяльність, яку виконує або ініціює розробник.

7. ***Продукти*** *(Work Products)* – результат, що передають розробники або групи іншим розробникам або групам (наприклад, варіанти використання, визначення класів, визначення тестів, документація, визначення інтерфейсів та ін.).

8. ***Стандарти*** *(Standards)* – що дозволяється або не дозволяється для продуктів. Розрізнюють стандарти позначень,

у тому числі і мови програмування, стандарти керування та прийняття рішень та домовленості проекту. Деякі методології обирають не всі стандарти одразу, а відкладають визначення деяких до певного етапу проекту.

9. ***Якість*** *(Quality)* – правила, запитання або зауваження, які необхідно відстежувати для кожного результату.

Додатковою характеристикою методології також є ***Цінності*** *(Values)* – те, до чого прагне команда, які види комунікації та організації роботи переважають. Цінності визначають ефективність методології: одна й та сама методологія для команд із різними цінностями буде мати відмінні показники ефективності.

Потрібно зауважити, що проекти з використанням полегшених методологій (тих, у яких мало уваги приділяється формалізації робіт) частіше приходять до успіху, ніж проекти із дуже формалізованими процедурами.

Другий принцип полягає у тому, що рівень критичності системи визначає «щільність» метології. Щільність методології визначається рівнем деталізації та взаємодії при її використанні. Більш щільні методології є більш формалізованими. А.Коуберн пропонує розділяти системи на чотири групи залежно від рівня можливих втрат:

- втрата комфортності – *найнижча критичність* – наприклад, збій у роботі модуля автоматизованого проектування призведе до збільшення ручних розрахунків;
- втрата незначної суми матеріальних ресурсів, наприклад, неправильне нарахування заробітної плати автоматизованою системою можна виправити у ручному режимі;
- матеріальні втрати не можна відшкодувати, наприклад, збій системи національного банку країни;
- втрата життя – *найвища критичність* – наприклад, збій системи керування на атомній електростанції. Також до систем найвищої критичності відносять аерокосмічні системи, системи керування польотами.

Тобто для підвищення надійності систем із високим рівнем критичності потрібно розробляти детальний опис

проекту та докладну програмну документацію, регулярно надавати розроблені артефакти як для внутрішної перевірки, так і для перевірки замовниками. Усі ці заходи підвищують вартість артефактів, але зменшують кількість помилок ще на ранніх стадіях проекту.

Третій принцип визначає те, що збільшення розміру методології та/або її щільності приводить до збільшення вартості проекту. Зрозуміло, що координація різних груп розробників для передачі інформації, оновлення документації після перевірок потребують часу і зусиль для якісного цілісного сприйняття. Пояснюючи цей принцип, А.Коуберн зауважує, що розмір проекту та розмір методології знаходяться у відношенні зворотної пропорції – для невеликої команди розробників не потрібна дуже формалізована методологія, бо вони легко можуть контактувати між собою та передавати будь-які артефакти. Головна проблема полягає у тому, що на початку проекту неможливо точно уявити його обсяги.

Четвертий принцип визначає, що найкращий спосіб комунікації між розробниками – безпосереднє спілкування. Цей принцип не абсолютизує вимогу бути усім розробникам в одній кімнаті, а надає рекомендації щодо підвищення ефективності роботи невеликої команди.

А. М. Терехов зауважує, що керування проектом потребує постійних перевірок відповідності реальної ситуації плану робіт [16]. Будь-яка методологія передбачає періодичні перевірки ходу проекту. У формалізованих методиках розроблення ПЗ обов'язково застосовуються звіти від керівників груп, за якими відбувається оцінка відповідності стану виконання кожної роботи порівняно з мережевим графіком.

Сама задача визначення затримки виконання певної роботи є нетривіальною. Відвертість звітів залежить від психологічного клімату в колективі, рівня довіри між керівниками та підлеглими. Зменшити суб'єктивний вплив на якість звітів про виконання робіт дозволяють технічні засоби. Для підвищення рівня контролю усі артефакти та звіти доцільно зберігати в єдиному репозиторії. Це дає можливість керівникові

переглянути результати у розвитку проекту, оцінити роботу не тільки групи в цілому, але і кожного її учасника.

Усунути більшість проблем керування можна за рахунок кваліфікованого проектування, урахування можливих ризиків, забезпечення можливості регулярного спілкування між розробниками, виділення максимально незалежних компонентів (тобто компонентів, які можуть бути передані стороннім розробникам).

Також на стиль керування впливають ті стандарти, яких потрібно дотримуватися під час розроблення ПЗ. Ці стандарти можуть бути міжнародними, як ISO 12207 [3], регіональними, як нормативні документи серії ГОСТ 34.х та 36.х [7,8,9], або стандартами підприємства, як наприклад, Oracle Unified Method - стандарт компанії Оракл [17].

Баррі Боемом були сформульовані 10 основних ризків розроблення ПЗ [18]:

- дефіцит фахівців;
- нереалістичні терміни і бюджет;
- реалізація невідповідної функціональності;
- розроблення неправильного користувальницького інтерфейсу;
- перфекціонізм, непотрібна оптимізація і відточування деталей;
- безперервний потік змін;
- брак інформації про зовнішні компоненти, які визначають оточення системи або залучені в інтеграцію;
- недоліки робіт, виконуваних зовнішніми (по відношенню до проекту) ресурсами;
- недостатня продуктивність роробленої системи;
- "розрив" у кваліфікації фахівців різних галузей знань.

У цьому переліку більша частина ризиків пов'язана з організаційними та процесними аспектами взаємодії фахівців у проектній команді.

Кент Бек (Kent Beck), батько методології eXtreme Programming, називає ризики основними проблемами

розроблення програмного забезпечення [19], до яких він відносить:

- зміщення графіків;
- закриття проекту через недотримання строків та бюджету;
- втрата корисності системи – розроблене ПЗ, успішно встановлене у реальному робочому середовищі, недоцільно розвивати або усувати в ньому виявлені дефекти, оскільки стає дешевшим замінити систему новою розробкою;
- велика кількість дефектів і недоліків не дозволяють замовникові використовувати систему;
- невідповідність ПЗ розв'язуваній проблемі – програмна система не розв’язує проблему бізнесу, для вирішення якої вона спочатку призначалася;
- зміна характеру бізнесу – ПЗ втратило актуальність;
- нестача можливостей – програма має можливості, жодна з яких не приносить замовникові досить багато користі;
- плинність кадрів.

Більшість із виділених проблем пов’язані із нечіткими вимогами та неправильним плануванням робіт.

Цікаві поради наведені в роботі Фредеріка Брукса (Frederick Brooks) [20], що наголошує на тому, що при правильній організації добре підготованого колективу можна зменшити вірогідність зривів роботи та спрогнозувати їх появу заздалегідь, але якщо зрив вже стався – тільки особистий досвід та інтуїція керівника допоможуть знайти шлях вирішення проблем з мінімальними втратами.

Забезпечення якості ПЗ

Якість програмного забезпечення (Software quality) асоціація IEEE визначає як ступінь відповідності програмного забезпечення встановленій комбінації властивостей [14]. У Міжнародному стандарті якості ISO 9000:2007 це поняття визначається як сукупність характеристик ПЗ, які забезпечують встановлені та очікуваі вимоги [21].

До **характеристик якості ПЗ** відносять (рис.12):

- функціональність – виконання заявлених функцій, відповідність стандартам, функціональна сумісність, безпека, точність;
- надійність – стійкість до відмов, можливість відновлення, завершеність;
- ефективність – економія часу, ефективність використання;
- зручність у використанні - ергономічність, інтуїтивна зрозумілість, повна документація;
- зручність для супроводу – стабільність, придатність для контролю та внесення змін;
- портативність – зручність установки, можливість заміни, сумісність.

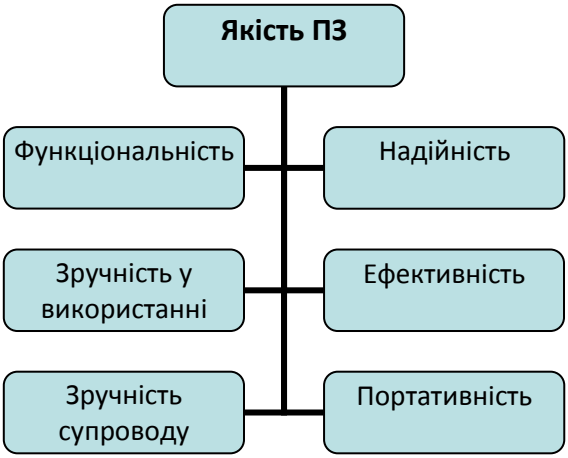


Рисунок 12 – Характеристики якості ПЗ

Забезпечення якості (Quality assurance, QA) – сукупність заходів, що охоплюють усі технологічні етапи розроблення, випуску та експлуатації ПЗ інформаційних систем на різних стадіях життєвого циклу ПЗ, для забезпечення його якості.

При виконанні цих заходів для кожного продукту перевіряються:

- повнота та коректність документації;

- коректність процедур встановлення та запуску;
- ергономічність використання;
- повнота тестування.

У 80-х рр. XX ст. розмір проектів із створення програмних систем зріс настільки, що вручну стало неможливо тестувати ПЗ, тому активно стали розроблятися засоби автоматизації процесу тестування. Через дестиліття поняття якості ПЗ розширилося настільки, що було виділено окремий вид діяльності при створенні ПЗ – забезпечення якості (Quality assurance, QA). На цей час автоматизоване тестування значно поширене, а засоби автоматизованого тестування часто вбудовані у середовище програмування.

Для виконання заходів забезпечення якості ПЗ розробники часто використовують спеціальні групи контролю якості, які мають назву QA. Група QA всередині компанії фактично виконує роль вимогливого користувача. У деяких компаніях заборонено неформальне спілкування між групою розробників та групою тестувальників. Від відповідальності групи QA залежить успіх продукту. Якщо ПЗ не сподобалося, з будь-яких причин користувачам продукт назавжди втрачає репутацію і споживача.

Незнайдені на стадії розроблення помилки коштують дорого. Традиційна стратегія розроблення ПЗ підпорядковується фундаментальному правилу, що визначає, що впродовж роботи над проектом вартість внесення змін у створюване ПЗ збільшується за експонентою (рис.13) [19].

Фактично від того, наскільки якісно виконані початкові етапи розроблення ПЗ залежить його вартість. З метою підвищення якості розроблення та уникнення ризиків, пов’язаних із нечіткими або постійно змінюваними вимогами, була створена методологія XP (eXtreme Programming) [19].

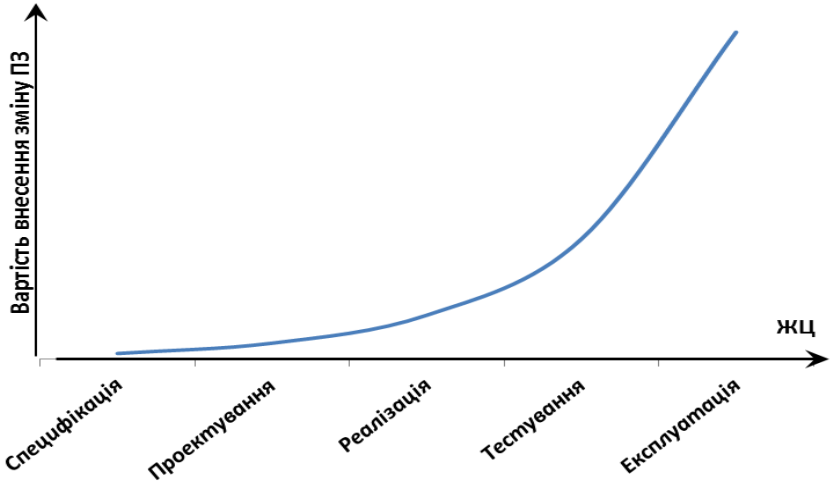


Рисунок 13 – Залежність вартості змін проектів від часу

Тестування *(Software Testing)* – діяльність, що виконується для оцінювання та поліпшення якості ПЗ. Ця діяльність базується на виявленні дефектів і проблем програмного забезпечення [14]. Тестування ПЗ включає в себе діяльність із планування робіт (Test Management), проектування тестів (Test Design), виконнання тестування (Test Execution) та аналізу отриманих результатів (Test Analysis).

Потрібно відмітити, що програмування та тестування – різні за суттю види діяльності. Якщо програмування – процес синтезу, то тестування – процес аналізу. Тестування потребує від виконавця в пешу чергу уважності та педантичності. Тестувальник повинен шукати недоліки будь-де в системі. Якщо початкове тестування для налагодження коду виконує сам програміст, то наступні етапи перевірки повинні готувати і виконувати інші особи, щоб перевірка була повноцінною, а не тільки для очікуваних проблемних місць.

Для планування потрібно мати уявлення про потрібні обсяги тестування. У роботі [22] наведені важливі статистичні дані щодо тестування:

- на 1000 рядків коду програміст у середньому робить 100 помилок, 70% з яких усуваються на стадії налагодження коду;
- при системному тестуванні у відділі контролю якості на ліквідацію однієї помилки потрібно від чотирьох до шістнадцяти годин;
- для усунення помилки, що була виявлена у ході експлуатації, потрібно від 33 до 88 годин.

Ці дані показують, як важливо виявити та усунути проблеми ще на ранніх етапах розроблення. Цікавим підходом для підвищення якості програмування є парне програмування, що зменшує кількість помилок на 15 % порівняно із традиційним одиночним кодуванням [23].

Найдавнішим прийомом тестування є перевірка усіх вводів та виводів даних, передбачених та неприпустимих (complete testing). Програма повинна адекватно реагувати на помилкові ситуації, без втрати стійкості в роботі. Також при тестуванні потрібно перевірити виконання усіх передбачених функцій системи. Перелік таких тестів складається на ранніх етапах проектування паралельно із описом функцій програмного продукту.

Види тестування можна класифікувати за різними показниками [24]:

1. За рівнем знання системи:

- чорний ящик (Black-box testing) – без перегляду програмного коду;
- білий ящик (White-box testing) – тестування із вивченням коду програми;
- сірий ящик (Gray-box testing) – тестування із частковим вивченням коду програми.

2. За об'єктом тестування:

- функціональне (Functional testing) – перевірка виконання заданих функцій;
- тестування продуктивності (Performance testing) – перевірка стабільності роботи програми або її частини при заданому навантаженні (Load testing), при перевантаженні (Stress

testing) та тривалому середньому навантаженні (Stability testing);

- юзабіліті-тестування (Usability testing) – перевірка ергономічності ПЗ;
- перевірка інтерфейсу користувача (UI testing);
- перевірка безпеки (Security testing) – тестування роботи системи з точки зору безпеки інформації;
- тестування сумісноті (Compatibility testing) – нефункціональне тестування, метою якого є перевірка коректної роботи ПЗ у певному оточенні.

3. За часом виконання тестування:

- альфа-тестування (Alpha testing) – перевірка роботи ПЗ перед передачею в експлуатацію силами компанії-розробника;
    - півгодинне тестування (Smoke testing) – коротка перевірка функцій системи, як правило, один тест для кожної функції;
    - тестування нової функціональності (New feature testing);
    - регресійне тестуванн (Regression testing) – перевірка роботи вже протестованих модулів;
    - тестування під час передачі ПЗ (Acceptance testing);
- бета-тестування (Beta testing) – тестування ПЗ перед випуском сторонніми силами.

4. За ступенем ізольованості компонентів:

- компонентне тестування (Component / Unit testing) – перевірка коректності окремого модуля або компонента системи;
- інтеграційне тестування (Integration testing) – перевірка роботи модулів, об’єднаних у групу;
- системне тестування (System / end-to-end testing) – перевірка роботи зібраного ПЗ з метою встановлення відповідності очікуваним функціям.

5. За ступенем автоматизації:

- ручне тестування (Manual testing);
- автоматизоване тестуванняя (Automated testing);
- напівавтоаматизоване тестування (Semiautomated testing).

6. За ступенем підготовленості до тестування:

- формальне тестування (Formal testing) – тестування відповідно до плану, встановлених процедур;
- інтуїтивне тестування (Ad hoc testing) – тестування без попереднього плану дозволяє на раніх стадіях виявляти помилки.

7. За ознакою позитивності сценаріїв:

- перевірка позитивних сценаріїв (Positive testing);
- перевірка негативних сценаріїв (Negative testing).

Тестування, як і будь-який процес, повино мати методи оцінки якості тестування. Одним із способів є спосіб, якому в програму спеціально вставляється певна кількість помилок. Після проведення тестування оцінюється, скільки таких помилок було знайдено. Частка знайдених помилок показує якість тестування і допомагає оцінити, що обсяг робіт потрібно виконати для повного усунення проблем. Плануючи тестування, потрібно визначати очікувані результати тестування, які покажуть готовність створюваного продукту. Бажано затвердити їх із замовником для чіткого визначення показників завершення проекту.

Якщо проектувальники тестів повинні мати інтегральне уявлення про систему, то виконавці тестів не потребують високої кваліфікації. Часто цю роль виконують новачки у компаніях, які не мають досвіду роботи.

Для підвищення рівня контролю якості кожного проекту повинна формуватися **база даних помилок**, в яку вноситься така інформація:

- хто знайшов помилку,коли;
- опис помилки;
- модуль, у якому була знайдена помилка;
- версія продукту;
- статус помилки:
  - open: знайдена;
  - fixed: виправлена;
  - can't reproduce: неможливо повторити;
  - by design: помилка проектування;

- wont fix: не помилка;
- postponed: зараз виправити важко, буде виправлена у наступній версії;
- regression: виправлена помилка з'явилася знову;

- важливість (severity) помилки:
  - crash: повна втрата даних;
  - major problem: програма частково не працює, часткова втрата даних;
  - minor problem: програма працює не так, як очікувалось, але дані не втрачаються;
  - trivial: зараз не потрібно виправляти;
- пріоритет помилки:
  - highest: не можна поставити продукт із такою помилкою, не можна перейти до наступної версії;
  - high: поставити не можна, але можна перейти до наступної версії;
  - medium: помилка буде виправлена;
  - low: косметичні поліпшення – помилка залишиється до наступної версії.

Така база даних дозволяє проаналізувати якість роботи окремих програмістів, їх груп, оцінювати якість проекту та можливості використовуваних інструментальних засобів. Також керівники можуть відслідковувати ризики розроблення за помилками найвищих рівнів пріоритету та важливості.

Дані з бази даних помилок дозволяють приймати рішення про можливість випуску програмного продукту (помилки з важливістю "crash" або з пріоритетом "highest/high" не дозволяють поставляти ПЗ). Якщо ПЗ обов'язково потрібно поставити замовнику, а часу на виправлення помилок немає, за домовленістю можна відкинути функції, які виконуються із помилками.

Питання для самоконтролю

1. Які типи вимог до ПЗ можна виділити? Як виконується опис цих вимог?
2. Як пов’язані між собою строки проектних робіт, їх обсяг та вартість? Яким чином, керуючи цими показниками, можна забезпечити якість проекту?
3. Які елементи повинен містити проектний план?
4. Які форми планів проектних робіт вам відомі? У чому принципова різниця між ними?
5. Охарактеризуйте принципи керування проектом.
6. Які складові методології розроблення потрібно враховувати при її виборі?
7. З якими ризиками стикаються розробники ПЗ?
8. Перелічіть та коротко опишіть характеристики якості ПЗ.
9. Що таке Quality assurance, чим воно відрізняється від тестування?
10. Опишіть види тестування.
11. Яку інформацію доцільно зберігати в базі даних помилок?

ТЕМА 3. Стандарти на розроблення та супровід програмного забезпечення

Стандартизація розроблення ПЗ

Розвиток будь-якої галузі економіки обов'язково супроводжується формалізацією використовуваних підходів та появою стандартів різного рівня. На ранніх етапах окремі підприємства формалізують внутрішні процеси, щоб запезпечити повторюваність результатів процесу або створення певного продукту. Для полегшення взаємодії підприємств та зручності споживачів розробляються *галузеві стандарти*. Розвиток кожного виду господарської діяльності приводить до потреби у державних засобах забезпечення якості продукції або процесу, тому розробляються та затверджуються *державні стандарти*. Для поліпшення умов співробітництва, розроблення загальнозрозумілих правил конкуренції на міжнародному ринку створюються об'єднання галузевих органів стандартизації, результатом діяльності яких є регіональні стандарти (діють у обмеженому переліку держав, які приєдналися) або *міжнародні стандарти*.

Одним із перших стандартів, що мав істотний вплив на розвиток теорії проектування та розроблення ІС, був стандарт BSP (Business System Planning). Даний стандарт був розроблений компанією IBM у середині 70-х рр. XX ст. Процес BSP передбачав виділення в ході розроблення ІС таких кроків: отримання підтримки керівництва, визначення процесів підприємства, визначення класів даних, проведення інтерв'ю, обробка та організація результатів інтерв'ю. Найважливіші кроки процесу BSP спостерігаються у більшості формальних методик.

На сьогодні діють такі стандарти, які регламентують процес розроблення ПЗ:

- ГОСТ 34.601-90 [7] – державний стандарт, що поширюється на автоматизовані системи і встановлює стадії та етапи їх

створення. У стандарті міститься опис змісту робіт на кожному етапі.

- ISO/IEC 12207. Systems and software engineering – Software Life Cycle Processes [3] – міжнародний стандарт на процеси розроблення та організацію життєвого циклу ПЗ. Поширюється на всі види замовленого ПЗ. Стандарт не містить опису фаз, стадій та етапів.
- Guide to the Software Engineering Body of Knowledge (SWEBOK) [25] – Керівництво до зведення знань з програмної інженерії – галузевий стандарт Інституту інженерів з радіоелектроніки та електротехніки (IEEE), що систематизує основні види діяльності з програмної інженерії.

Міжнародні стандарти ISO

Базовим стандартом розроблення ПЗ є **ISO 12207. Systems and software engineering – Software Life Cycle Processes**, в якому усі процеси ЖЦ ПЗ розподілені на три групи (рис.14).

У табл. 1 наведений опис основних процесів ЖЦ.

Рисунок 14 – Процеси ЖЦ ІС відповідно до стандарту ISO 12207

Таблиця 1 – Зміст основних процесів ЖЦ ПЗ ІС відповідно до ISO 12207

Процес (виконавець)	**Дії**	**Вхід**	**Результат**
Купівля (замовник)	– Ініціювання – Підготовка вимог заявки – Підготовка угоди – Контроль діяльності постачальника – Приймання ІС	– Рішення про початок впровадження ІС – Результати дослідження діяльності замовника – Результати аналізу ринку ІС/ тендера – План постачання/ розроблення – Комплексний тест ІС	– Техніко-економічне обгрунтування впровадження ІС – Технічне завдання на ІС – Угода на постачання/ розроблення – Акти приймання етапів роботи – Акт приймально-передавальних випробувань
Постачання (розробник ІС)	– Ініціювання – Відповідь на замовлення – Підготовка угоди – Планування виконання – Постачання ІС	– Технічне завдання на ІС – Рішення про участь у розробленні – Результати тендера – Технічне завдання на ІС – План керування проектом – Створена ІС та документація	– Рішення про участь в розробленні – Комерційна пропозиція/ конкурсна заявка – Угода про постачання/ розроблення – План керування проектом – Реалізація/ коригування – Акт приймально-передавальних випробувань

Продовження таблиці 1

Процес (виконавець)	**Дії**	**Вхід**	**Результат**
Розроблення (розробник ІС)	– Підготовка – Аналіз вимог до ІС – Проектування архітектури ІС – Розроблення вимог до ІС – Проектування архітектури ПЗ – Детальне проектування ПЗ – Кодування та тестування ПЗ – Інтеграція ПЗ та кваліфікаційне тестування ПЗ – Інтеграція ІС та кваліфікаційне тестування ІС	– Технічне завдання на ІС – Технічне завдання на ІС, модель ЖЦ – Технічне завдання на ІС – Підсистеми ІС – Специфікації вимог до компонентів ПЗ – Архітектура ПЗ – Інформація про деталі проектування ПЗ – План інтеграції ПЗ, тести – Архітектура ІС, ПЗ, документація на ІС, тести	– Модель ЖЦ, стандарти розроблення – План робіт – Склад підсистем, компоненти обладнання – Специфікації вимог до компонентів ПЗ – Склад компонентів ПЗ, інтерфейси з БД, план інтеграції ПЗ – Проект БД, специфікації інтерфейсів між компонентами ПЗ, вимоги до тестів – Тексти модулів ПЗ, акти автономного тестування – Оцінка відповідності ПЗ вимогам ТЗ – Оцінка відповідності ПЗ, БД, технічного комплексу та комплекту документації вимогам ТЗ

Допоміжні процеси призначені для підтримки виконання основних процесів, забезпечення якості проекту, організації верифікації та тестування ПЗ. **Організаційні процеси** визначають дії та завдання замовників та розробників для керування процесами у ході проекту.

Для підтримки практичного використання стандарту ISO 12207 розроблені такі технологічні документи: Керівництво для ISO/IEC 12207 (ISO/IEC TR 24748-3:2011 Systems and software engineering - Life cycle management - Part 3: Guide to the application of ISO/IEC 12207 (Software life cycle processes)) та Керівництво з використання ISO/IEC 12207 в керуванні проектами (ISO/IEC TR 16326:2009 Systems and software engineering - Life cycle processes - Project management).

У 2002 р. був опублікований стандарт на процеси життєвого циклу систем **ISO/IEC 15288 Systems and software engineering - System life cycle processes** [26], у розробленні якого брали участь фахівці різних галузей: системної інженерії, програмування, управління якістю, людськими ресурсами, безпекою та ін. Даний документ враховує практичний досвід створення систем в урядових, комерційних, військових та академічних організаціях і може бути застосований для широкого класу систем, але його основне призначення – підтримка створення комп'ютеризованих систем. На цей час діє версія стандарту 2008 р. У стандарті ISO/IEC 15288:2008 у структурі ЖЦ виділені групи процесів за видами діяльності (рис.15).

Стадії створення системи, передбачені у стандарті ISO/IEC 15288:2008, та основні результати, які повинні бути досягнуті до моменту їх завершення, наведені в таблиці 2.

Стандарти ISO/IEC 12207 та ISO/IEC 15288 мають єдину термінологію і розроблені таким чином, щоб могли використовуватись одночасно у проекті.

Таблиця 2 – Стадії створення систем (ISO/IEC 15288)

Ном. поряд.	Стадія	Опис
1	Формування концепції	Аналіз потреб, вибір концепції та проектних рішень
2	Розроблення	Проектування системи
3	Реалізація	Створення системи
4	Експлуатація	Введення в експлуатацію та використання системи
5	Підтримка	Забезпечення функціонування системи
6	Зняття з експлуатації	Зупинення використання, демонтаж, архівування системи

Рисунок 15 – Процеси ЖЦ систем відповідно до стандарту ISO/IEC 15288

Також потрібно відмітити, що у процесі промислового розроблення ПЗ обов'язково використовуються стандарти якості серії **ISO 9000.** Серія ISO 9000 [19] (управління якістю) містить у собі такі стандарти:

- ISO 9000-1. Керування якістю і гарантії якості. Частина 1. Посібник з вибору й використання.
- ISO 9000-2. Керування якістю й гарантії якості. Частина 2. Загальний посібник із застосування стандартів ISO 9001, ISO 9002 і ISO 9003.
- ISO 9000-3. Керування якістю й гарантії якості. Частина 3. Посібник із застосування стандарту ISO 9001 при розробленні, установці й супроводі ПЗ.
- ISO 9000-4. Керування якістю й гарантії якості. Частина 4. Посібник з керування надійністю програм.

Основний стандарт **ISO 9001:2009** [26] задає модель системи якості для процесів проектування, розроблення, виробництва, установки й обслуговування (продукту, системи, послуги). У моделі ISO 9000 лише згадуються вимоги, які повинні бути реалізовані, але не говориться, як це можна зробити. Тому для побудови повноцінної системи якості за ISO, крім основної моделі ISO 9001, необхідно використовувати допоміжні галузеві та рекомендаційні стандарти.

Стандарти організації IEEE

У 1963 р. у результаті злиття Інституту радіотехніків (Institute of Radio Engineers, IRE) і Американського інституту інженерів-електриків (American Institute of Electrical Engineers, AIEE) була створена міжнародна некомерційна асоціація технічних фахівців, світовий лідер у галузі розроблення стандартів з радіоелектроніки та електротехніки Інститут інженерів з радіоелектроніки та електротехніки IEEE (Institute of Electrical and Electronics Engineers). Дана міжнародна організація об'єднує понад 400 тис. фахівців із 170 країн. IEEE здійснює інформаційну і матеріальну підтримку фахівців для організації та розвитку наукової діяльності в електротехніці, електроніці, комп'ютерній техніці та інформатиці.

Керівництво до зведення знань із програмної інженерії (SWEBOK, 2004) містить опис 10 галузей знань [25]:

- Software requirements – програмні вимоги;
- Software design – дизайн (архітектура);
- Software construction – конструювання програмного забезпечення;
- Software testing – тестування;
- Software maintenance – експлуатація (підтримка) програмного забезпечення;
- Software configuration management – управління конфігураціями;
- Software engineering management – управління у програмній інженерії;
- Software engineering process – процеси програмної інженерії;
- Software engineering tools and methods – інструменти та методи;
- Software quality – якість програмного забезпечення.

Для кожної галузі SWEBOK містить опис ключових елементів у вигляді підобластей (subareas). Для кожної підобласті наведена декомпозиція у вигляді списку тем (topics) із їх описом.

Стандарт зрілості компанії-розробника ПЗ CMM

Говорячи про стандартизацію процесів підприємства потрібно розглянути модель зрілості технологічних процесів організації Capability Maturity Model (CMM) [28], розроблену Інститутом інженерів програмного забезпечення (Software Engineering Institute, SEI) та корпорацією Mitre під керівництвом Уоттса Хамфри (Watts Humphrey).

Методологія CMM розроблялася й розвивалася в США як засіб, що дозволяє вибирати кращих виробників ПЗ для виконання держзамовлень у першу чергу міністерства оборони. Для цього були розроблені критерії оцінки зрілості ключових процесів компанії та визначений набір дій, необхідних для їхнього подальшого вдосконалювання. У підсумку методологія виявилася надзвичайно корисною для більшості компаній, що

прагнуть якісно поліпшити існуючі процеси проектування, розроблення, тестування програмних засобів і звести керування ними до зрозумілих і легко реалізованих алгоритмів і технологій, описаних у єдиному стандарті. У подальшому ця модель переросла у методологію підвищення якості процесів підприємства Capability Maturity Model Integration (CMMI). Застосування CMMI дозволяє поставити розроблення ПЗ на промислову основу, підвищити керованість ключових процесів і виробничу культуру в цілому, гарантувати якісну роботу й виконання проектів у строк.

Основою для створення CMM стало базове положення про те, що фундаментальна проблема "кризи" процесу розроблення якісного ПЗ полягає не у відсутності нових методів і засобів розроблення, а в нездатності компанії організувати технологічні процеси й керувати ними.

Для оцінки ступеня готовності підприємства розробляти якісний програмний продукт CMM використовує ключове поняття ***зрілість організації (Maturity)***. *Незрілою* вважається організація, у якій:

- відсутнє довгострокове й проектне планування;
- процес розроблення програмного забезпечення і його ключові моменти не ідентифіковані, реалізація процесу залежить від поточних умов, конкретних менеджерів і виконавців;
- методи і процедури не стандартизовані й не документовані;
- результат не визначений реальними критеріями, які встановлюються за запланованими показниками із застосуванням стандартних технологій і розроблених метрик;
- процес вироблення рішення відбувається стихійно, на грані мистецтва.

У цьому випадку велика ймовірність появи несподіваних проблем, перевищення бюджету або невиконання строків здачі проекту. У такій компанії, як правило, менеджери й розробники не керують процесами - вони змушені займатися поточними та спонтанними проблемами.

Основні ознаки *зрілої організації*:

- у компанії є чітко визначені й документовані процедури керування вимогами, планування проектної діяльності, керування конфігурацією, створення й тестування програмних продуктів, відпрацьовані механізми керування проектами;
- ці процедури постійно уточнюються й удосконалюються;
- оцінки часу, складності й вартості робіт ґрунтуються на накопиченому досвіді, розроблених метриках і кількісних показниках, що робить їх достатньо точними;
- актуалізовано зовнішні й створені внутрішні стандарти на ключові процеси й процедури;
- існують обов'язкові для всіх правила оформлення методологічної програмної й користувальницької документації;
- технології незначно змінюються від проекту до проекту на основі стабільних і перевірених підходів і методик;
- максимально використовуються напрацьовані у попередніх проектах організаційний і виробничий досвід, програмні модулі, бібліотеки програмних засобів;
- активно апробуються і впроваджуються нові технології, виробляється оцінка їхньої ефективності.

СММ визначає п'ять рівнів технологічної зрілості компанії (рис.16), за якими замовники можуть оцінювати потенційних претендентів на підпис контракту, а розробники – удосконалювати процеси створення ПЗ.

Кожний із рівнів, крім першого, складається з декількох ключових областей процесу (Key Process Area), що містять цілі (Goal), зобов'язання щодо виконання (Commitment to Perform), можливість виконання (Ability to Perform), виконувані дії (Activity Performed), їхній вимір і аналіз (Measurement and Analysis) та перевірку впровадження (Verifying Implementation). Таким чином, СММ фактично є комплексом вимог до ключових параметрів ефективного стандартного процесу розроблення ПЗ та засобом його постійного поліпшення. Виконання цих вимог

збільшує ймовірність досягнення підприємством поставлених цілей у сфері якості.

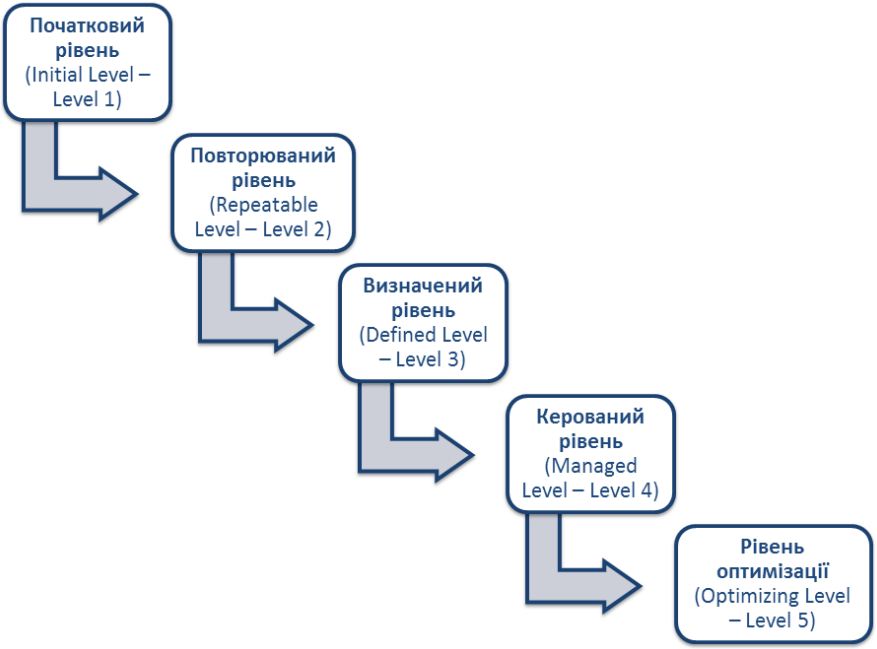


Рисунок 16 – Рівні зрілості компанії за CMM

Початковий рівень *(Initial Level – Level 1).*

На даному рівні компанія може одержати замовлення, розробити і передати замовникові програмний продукт. Стабільність розробок відсутня. Лише деякі процеси визначені, результат цілком залежить від зусиль окремих співробітників. Успіх одного проекту не гарантує успішності наступного. До цієї категорії можна віднести будь-яку компанію, що хоч якось виконує взяті на себе зобов'язання.

Ключові області процесу цього рівня не зафіксовані.

Повторюваний рівень *(Repeatable Level – Level 2).*

Цьому рівню відповідають підприємства, які володіють певними технологіями керування й розроблення. Керування вимогами та планування у більшості випадків ґрунтуються на

розробленій документованій політиці й накопиченому досвіді. Установлені й уведені в повсякденну практику базові показники для оцінки параметрів проекту. Менеджери відслідковують виконання робіт і контролюють тимчасові й виробничі витрати.

У компанії розроблені деякі внутрішні стандарти й організовані спеціальні групи перевірки якості QA. Зміни версій кінцевого програмного продукту й створених проміжних програмних засобів відслідковуються в системі керування конфігурацією. Є необхідна дисципліна дотримання встановлених правил. Ефективні методики й процеси установлюються, що забезпечує можливість повторення успіху попередніх проектів у тій самій прикладній області.

Ключові області процесу розроблення ПЗ цього рівня наведені на рис.17.

Визначений рівень *(Defined Level – Level 3).*

Рівень характеризується деталізованим методологічним підходом до керування (описані й закріплені у документованій політиці типові дії, необхідні для багаторазового повторення: ролі й відповідальність учасників, стандартні процедури й операції, порядок дій, кількісні показники й метрики процесів, формати документів та ін.).

Для створення й підтримки методологій в актуальному стані в організації підготовлена й постійно функціонує спеціальна група. Компанія регулярно проводить тренінги для підвищення професійного рівня своїх співробітників.

Починаючи з цього рівня, організація практично перестає залежати від особистісних якостей конкретних розроблювачів і не має тенденції опускатися на нижчі рівні. Ця незалежність обумовлена продуманим механізмом постановки завдань, планування заходів, виконання операцій і контролю виконання.

Управлінські й інженерні процеси документовані, стандартизовані й інтегровані в уніфіковану для всієї організації технологію створення ПЗ. Кожний проект використовує затверджену версію цієї технології, адаптовану до особливостей поточного проекту.

Ключові області процесу розроблення ПЗ цього рівня наведені на рис.17.

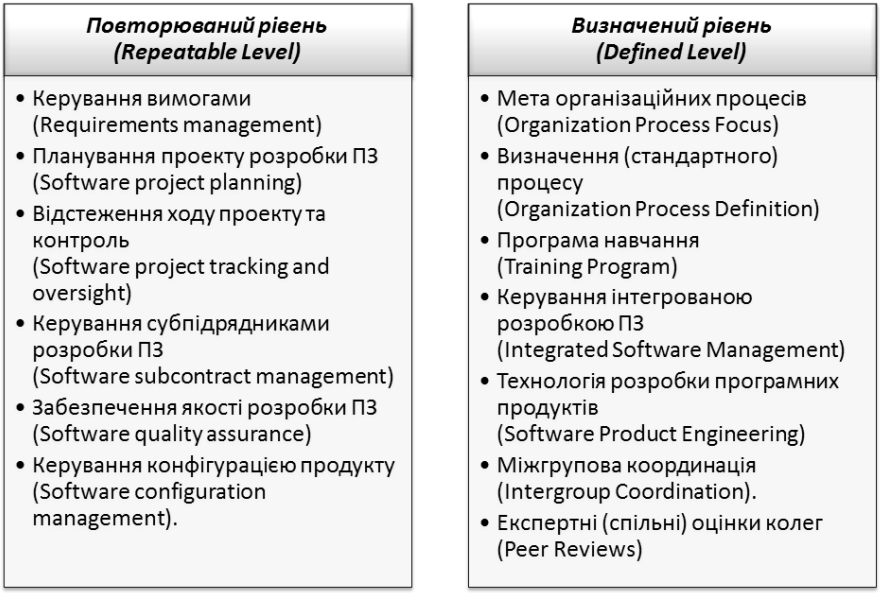


Рисунок 17 – Ключові області повторюваного та визначеного рівнів зрілості компанії

Керований рівень *(Managed Level – Level 4).*

Рівень, на якому розроблені та закріплені у відповідних нормативних документах кількісні показники якості. Більш високий рівень керування проектами досягається за рахунок зменшення відхилень різних показників проекту від запланованих. При цьому тенденції зміни продуктивності процесу можна відділити від випадкових варіацій на підставі статистичної обробки результатів вимірювань у процесах.

Ключові області процесу розроблення ПЗ цього рівня наведені на рис.18.

Рівень оптимізації *(Optimizing Level – Level 5).*

Для цього рівня заходи щодо вдосконалювання розраховані не лише на існуючі процеси, але й на запровадження, використання нових технологій і оцінку їхньої

ефективності. Основним завданням усієї організації на цьому рівні є постійне вдосконалювання існуючих процесів, що в ідеалі спрямовано на запобігання відомим помилкам або дефектам і попередження можливих. Застосовується механізм повторного використання компонентів від проекту до проекту (шаблони звітів, формати вимог, процедури й стандартні операції, бібліотеки модулів програмних засобів).

Ключові області процесу розроблення ПЗ цього рівня наведені на рис.18.

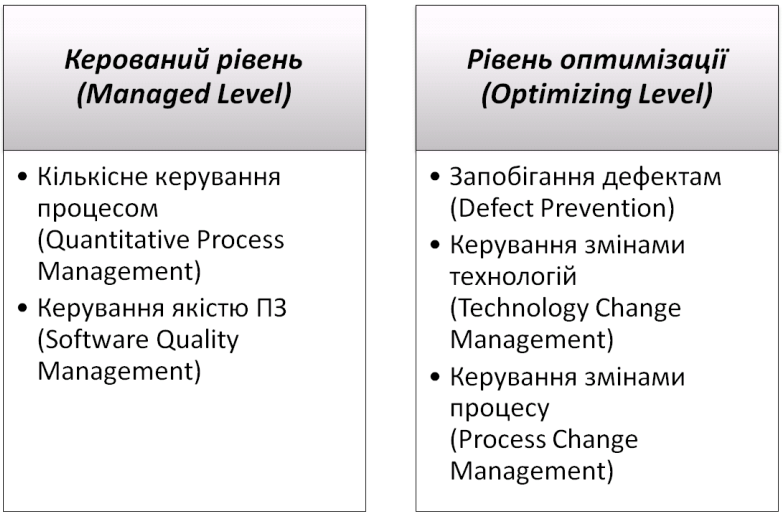


Рисунок 18 – Ключові області керованого рівня зрілості та рівня оптимізації компанії

CMM визначає такий мінімальний набір вимог: реалізувати 18 ключових областей процесу розроблення ПЗ, що містять 52 цілі, 28 зобов'язань компанії, 70 можливостей виконання (гарантій компанії) і 150 ключових практик.

У результаті аудиту та атестації компанії присвоюється певний рівень, що після наступних аудитів може підвищуватися або знижуватися. Кожний наступний рівень в обов'язковому порядку містить у собі всі ключові характеристики попередніх. У зв'язку з цим сертифікація компанії щодо одного з рівнів

припускає безумовне виконання всіх вимог більш низьких рівнів.

До *переваг* моделі СММ належить те, що вона орієнтована на організації, які займаються розробленням програмного забезпечення. У даній моделі вдалося більш детально визначити вимоги, специфічні для процесів, пов'язаних з розробленням ПЗ. Із цієї причини в СММ наведені не тільки вимоги до процесів організації, але й приклади реалізації таких вимог.

Основний *недолік* СММ полягає в тому, що модель не авторизована як стандарт ні міжнародними, ні національними органами зі стандартизації. Втім, СММ давно стала промисловим стандартом. До *недоліків* моделі також необхідно віднести більші зовнішні накладні витрати на приведення процесів компанії у відповідність до моделі СММ, ніж при використанні моделей Міжнародного стандарту ISO 9000.

Питання для самоконтролю

1. Які види стандартів вам відомі? Наведіть приклади.
2. Що відповідно до стандарту CMM повинна запровадити компанія-розробник програмних систем, щоб якісно виконувати замовлення?
3. Які міжнародні стандарти ISO регулюють розроблення інформаційних систем?
4. Які міжнародні стандарти якості використовуються під час розроблення інформаційних систем?
5. Які документи відповідно до державних стандартів України розробляються у ході створення програмного забезпечення?
6. Які групи процесів у ході розроблення програмного забезпечення виділяють стандарти ISO?
7. Які стандарти організації IEEE регламентують розроблення програмних систем?
8. Які області охоплює керівництво SWEBOK?
9. Які державні стандарти України регламентують розроблення інформаційних систем та програмного забезпечення?
10. Які дії, передбачені стандартом ISO 12207, виконуються в ході розроблення програмних продуктів?
11. Які рівні зрілості компанії виділяє стандарт CMM? Чим вони характеризуються?
12. На якому рівні зрілості компанія може контролювати якість створюваного програмного забезпечення?

ТЕМА 4. Сучасні методології розроблення програмних систем

CASE–засоби та нотації моделювання програмних систем

На становлення технологій виробничого програмування найбільш помітний вплив мали методологія структурного програмування та об'єктно-орієнтоване програмування. Перша дозволила усвідомити обмеженість здібностей людини, необхідних для створення великих програм. Об'єктно-орієнтоване програмування дало поштовх до розроблення методів декомпозиції, пристосованих для подолання складності проектів, та привело до модернізації основних принципів проектування програм. Незважаючи на ці та інші впливи, стадія комплексної автоматизації технологій програмування стала можливою лише при відповідному рівні розвитку техніки. Важливою обставиною, що дозволила перейти до комплексної автоматизації, стало усвідомлення того, що не можна говорити про промислове програмування без підтримки технологічних функцій на усіх етапах життя програм.

Приблизно на початку 90-х рр. XX ст. з'явився термін - CASE-технологія (Computer Aid Software Engineering - комп'ютерна підтримка розроблення програм), яким стали позначати використання систем, що містять комплексні автоматизовані засоби підтримки розроблення і супроводу ПЗ. Найбільш вдалим виявилося використання CASE-систем у тих спеціальних областях, в яких вже були успіхи і досвід технологічної практичної роботи, а також у тих випадках, коли ця область вже була забезпечена надійною теоретичною базою. На рис.19 наведено етапи розвитку CASE-засобів за областями застосування.

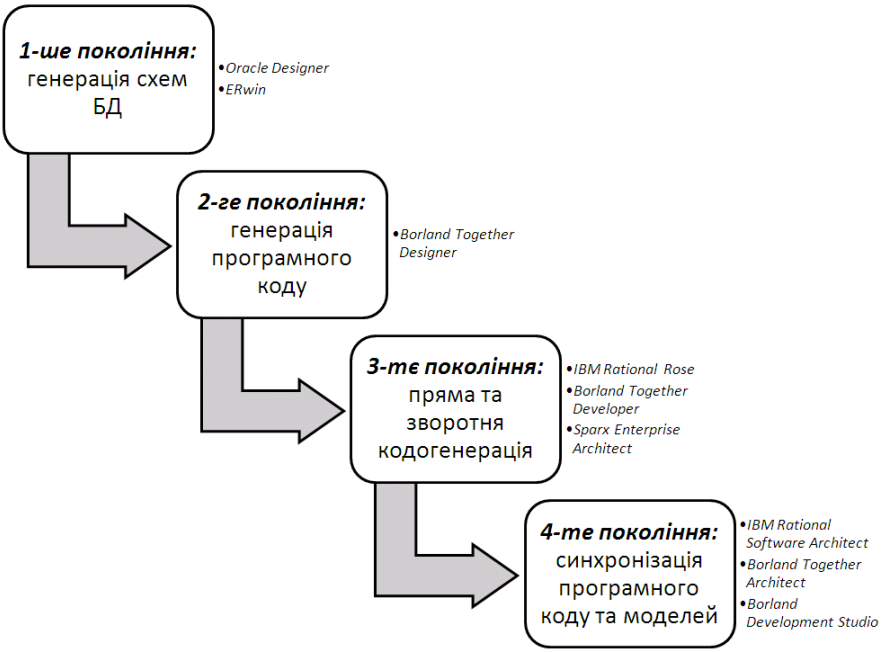


Рисунок 19 – Розвиток CASE-засобів

Першими з'явилися CASE-системи розроблення баз даних у розвинених реляційних СУБД (наприклад, Oracle Designer в системі Oracle). Наступними стали засоби генерації програмного коду та налагодження програм. Потреба підтримки не лише процесу програмування, а й етапу аналіз та проектування програмних продуктів привело до створення CASE-систем, які зв’язували моделі ПЗ із програмним кодом – спочатку лише для прямої та зворотної кодогенерації, а потім із підтримкою синхронного зв'язку коду із моделями аналізу та проектування.

Сьогодні універсальні CASE-системи будуються у рамках застосування розвинених, але все ж спеціальних методологій. Безперечний прогрес у даній сфері досягнутий для проектування, орієнтованого на моделювання на етапах аналізу і конструювання (CASE-системи 4-го покоління).

Складність процесу розроблення інформаційних систем викликала появу візуальних засобів моделювання (рис.20). У

рамках об'єктно-орієнтованого підходу Object management group (OMG) розроблена спеціальна уніфікована мова моделювання UML (Unified Modeling Language) [29, 30], що розглядається як основа проектування в методології ітеративного нарощування можливостей об'єктно-орієнтованих програмних систем. Широкого вжитку для моделювання процесів різних галузей екномічної діяльності набула родина візуальних мов IDEF (Integration Definition) [31], що є стандартом моделювання Міністерства оборони США. Паралельно із нотаціями UML та IDEF розвиваються методологія та нотація для професійного моделювання бізнес-процесів ARIS (ARchitecture of Integrated Information Systems) [32].

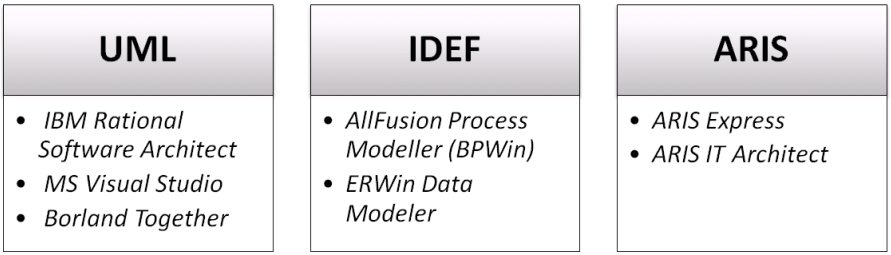
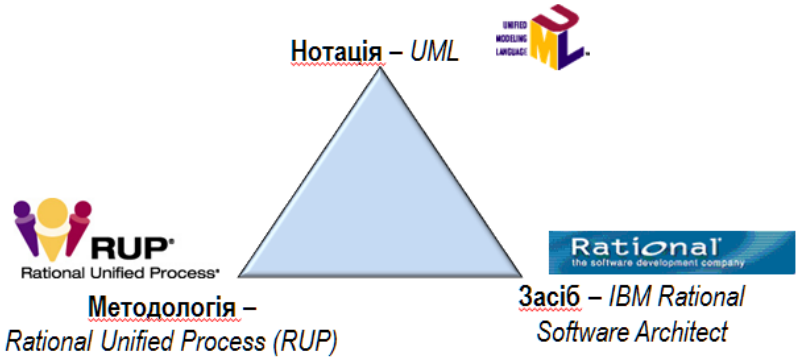


Рисунок 20 – Нотації візуального моделювання систем

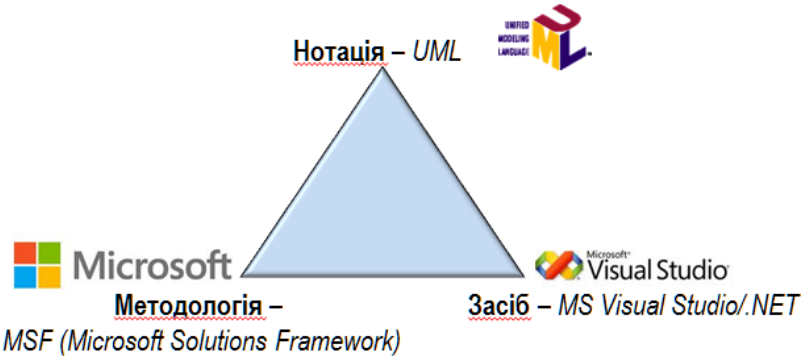
Візуальне моделювання мовою UML

У даному курсі будемо розглядати нотацію UML (uml.org), що є широкозастосовною при проектуванні ПЗ, і підтримується більш ніж 50 CASE-інструментами. На базі цієї мови побудований ряд CASE-систем загального призначення з розвиненими засобами (рис.21).

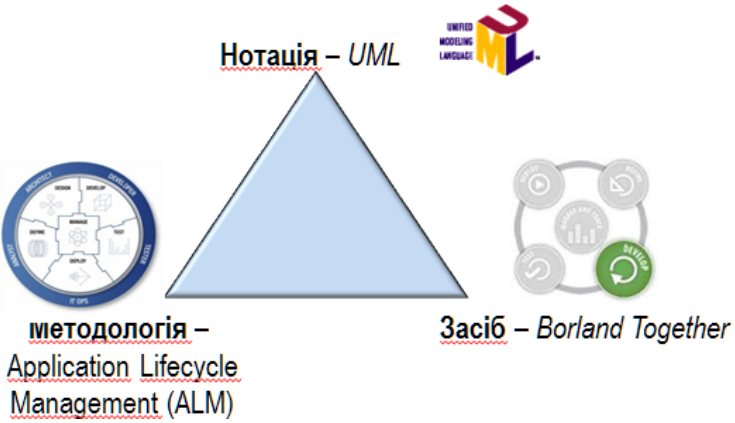
Інструменти використовують споріднені або кардинально відмінні методології розроблення, але для запису моделей вони використовують одну мову моделювання – UML.



а)



б)



в)

Рисунок 21 –Методології та інструменти візуального моделювання систем мовою UML

Інструменти використовують спорібнені або кардинально відмінні методології розроблення, але для запису моделей вони використовують одну мову моделювання - UML.

Необхідність використання UML залежить від складності проекту (рис.22). Обов'язково потрібно використовувати візуальне моделювання у технічно складних масштабних проектах, які розробляються для великої кількості користувачів. Використання візуальних моделей для невеликих проектів, які вирішують нескладні завдання, недоцільне, оскільки це лише збільшить вартість проекту та час розроблення.

Рисунок 22 – Використання UML залежно від складності проекту

Базові терміни та нотація

UML створена для візуального моделювання систем на основі об'єктно-орієнтованого підходу. Ця мова створювалася для оптимізації процесу розроблення програмних систем, що дозволяє збільшити ефективність їх реалізації у кілька разів і помітно поліпшити якість кінцевого продукту. Вона дозволяє будувати моделі для усіх фаз ЖЦ ПЗ.

Мова була створена як результат «війни методів» моделювання у 1995 р. та увібрала у себе риси нотацій Грейді Буча (Grady Booch), Джеймса Рамбо (James Rumbaugh), Айвара Якобсона (Ivar Jacobson) і багатьох інших (рис.23). З 1997 р. розробленням мови займається консорціум OMG (Object Management Group), створений у 1989 році для розроблення індустріальних стандартів з їх наступним використанням у процесі створення масштабованих неоднорідних розподілених об'єктних середовищ.

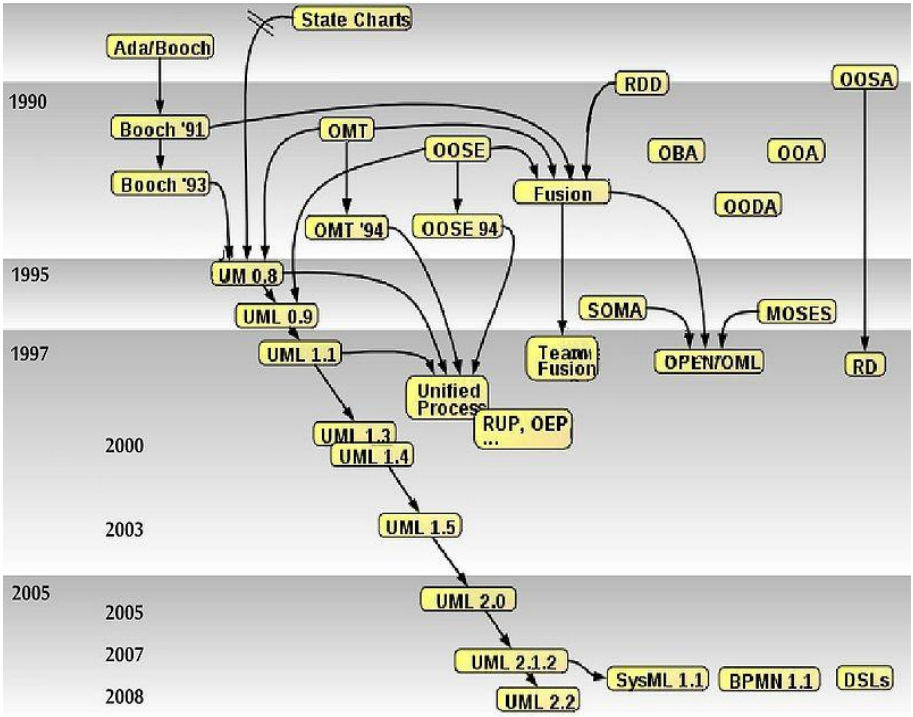


Рисунок 23 – Історія розвитку об’єктно-орієнтованих методів
(джерело - http://commons.wikimedia.org/wiki/File:OO-historie.jpg)

Універсальність UML підтверджує її широке застосування, особливо у галузі створення програмних систем та затвердження у 2005 р. версії UML 1.4.2 як стандарту ISO/IEC 19501:2005. Найновіша стабільна версія мови UML 2.4.1 вийшла у серпні 2011 р.

Моделювання за допомогою мови UML грунтується на таких *принципах*:

- ***абстрагування*** - у модель необхідно включати тільки ті елементи проектованої системи, які мають безпосереднє відношення до виконання нею своїх функцій;
- ***багатомодельність*** - ніщо єдина модель не може з достатнім ступенем точності описати різні аспекти системи. Можна описувати систему кількома взаємозалежним уявленнями, кожне з яких відображає певний бік її структури або поведінки;
- ***ієрархічність*** - при описі системи використовуються різні рівні абстрагування і деталізації у рамках фіксованих уявлень. При цьому перше уявлення системи описує її в найбільш загальних рисах і є представленням концептуального рівня, а наступні рівні розкривають різні сторони системи із зростаючим ступенем деталізації аж до фізичного рівня. Модель фізичного рівня в мові UML відображає компонентний склад проектованої системи з точки зору її реалізації на апаратурній і програмній платформах конкретних виробників.

Елементи (класифікатори) мови UML можна поділити на три групи (рис.24).

Рисунок 24 – Класифікатори мови UML

Сутності (entity) – абстракції, що є основними об'єкто-орієнтованими елементами мови UML, за допомогою яких будуються моделі. В UML визначено чотири типи сутностей: структурні, поведінкові, сутності групування та сутності-примітки.

- ***Структурні сутності*** - це іменники в моделях UML. Як правило, вони є статичними частинами моделей, які відповідають концептуальним або фізичним елементам системи. Існує *сім різновидів структурних сутностей*: Клас (Class), Інтерфейс (Interface), Кооперація (Collaboration), Варіант використання/Прецедент (Use case), Активний клас (Active class), Компонент (Component), Вузол (Node).
- ***Поведінкові сутності*** є динамічними складовими моделі UML, які описують поведінку моделі в часі і просторі. Це дієслова мови. Існують два основних типи поведінкових сутностей: Взаємодія (Interaction) та Автомат (State).
- ***Сутності групування*** є організуючими частинами моделі UML. Це блоки, на які можна розкласти модель. Первинна сутність групування – пакет (Package).
- ***Сутності-примітки*** - пояснювальні частини моделі UML. Це коментарі для додаткового опису, роз'яснення або

зауваження до будь-якого елемента моделі. Є тільки один базовий тип анотаційних елементів – примітка (Note).

Для зв'язування сутностей у моделях UML визначено чотири типи *відносин* (рис.25):

- ***Залежність (dependency)*** - це семантичне відношення між двома сутностями, при якому зміна однієї з них, незалежної, може вплинути на семантику іншої, залежної.

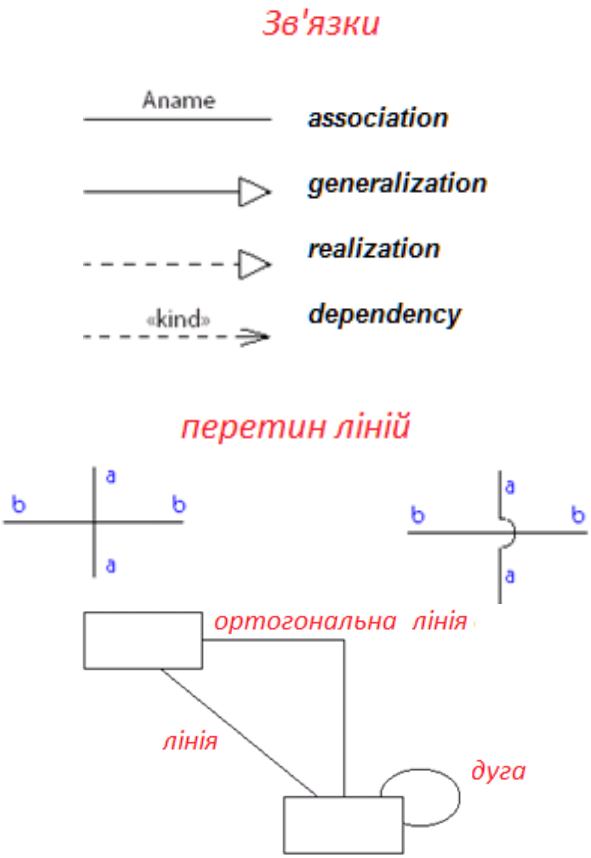


Рисунок 25 – Нотація відносин мови UML

- ***Асоціація (association)*** - структурне відношення, що описує сукупність змістовних або логічних зв'язків між об'єктами.
- ***Узагальнення (generalization)*** - це відношення, при якому об'єкт-нащадок (child) може бути підставлений замість об'єкта-батька (parent). При цьому відповідно до принципів об'єктно-орієнтованого програмування нащадок успадковує структуру і поведінку свого батька.
- ***Реалізація (realization)*** є семантичним відношенням між класифікаторами, при якому один класифікатор визначає зобов'язання, а інший гарантує його виконання. Відношення реалізації спостерігаються у двох випадках:
    - між інтерфейсами, реалізують класами або компонентами;
    - між варіантами використання, реалізують кооперації.

Усі різновиди сутностей та відносини UML в діаграмах мають свій спосіб графічного зображення (рис.25, 26).

В UML використовуються 13 діаграм [29], які поділені на дві основні групи (рис.27):

- ***структурні діаграми (Structure Diagrams)*** – моделі, призначені для опису статичної структури сутностей або елементів системи, включаючи їх класи, інтерфейси, атрибути та відношення;
- ***діаграми поведінки (Behavior Diagrams)*** – моделі, призначені для опису процесу функціонування елементів системи, включаючи їх методи та взаємодію між ними, а також процес зміни станів окремих елементів та системи в цілому.

Кожна діаграма уточнює уявлення про систему. При цьому діаграма варіантів використання є концептуальною моделлю системи, початковою для побудови усіх інших діаграм. Діаграма класів є логічною моделлю, що відбиває статичні аспекти структурної побудови системи, а діаграма діяльності, що також є різновидами логічної моделі, відображає динамічні аспекти її функціонування.

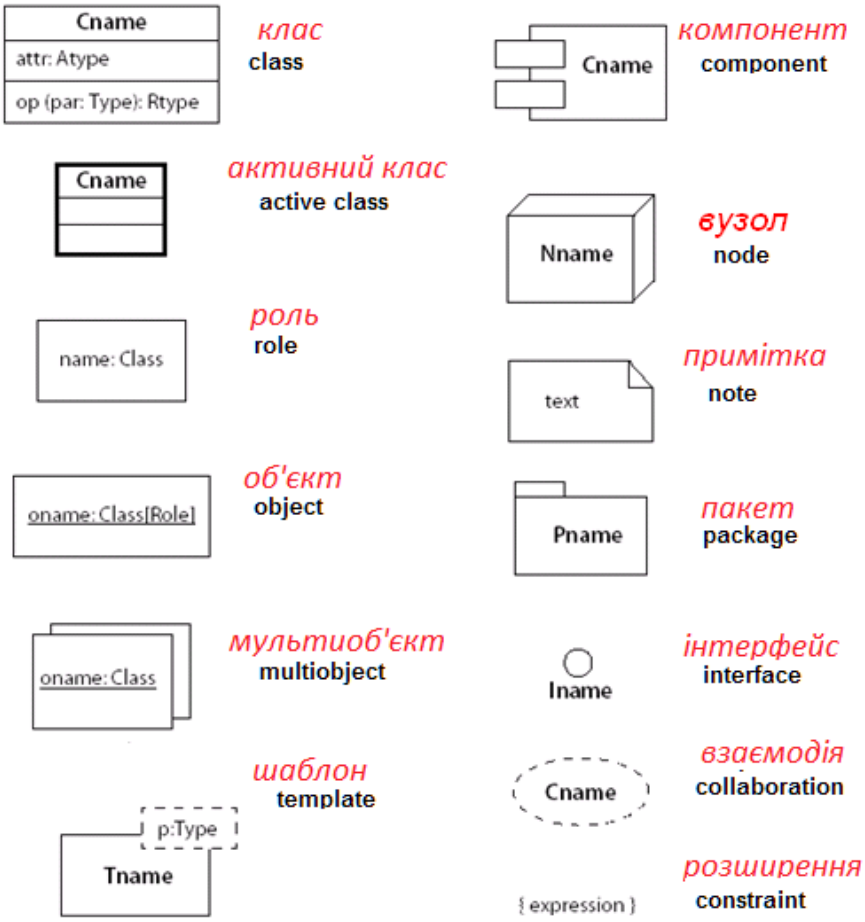


Рисунок 26 – Нотація сутностей мови UML

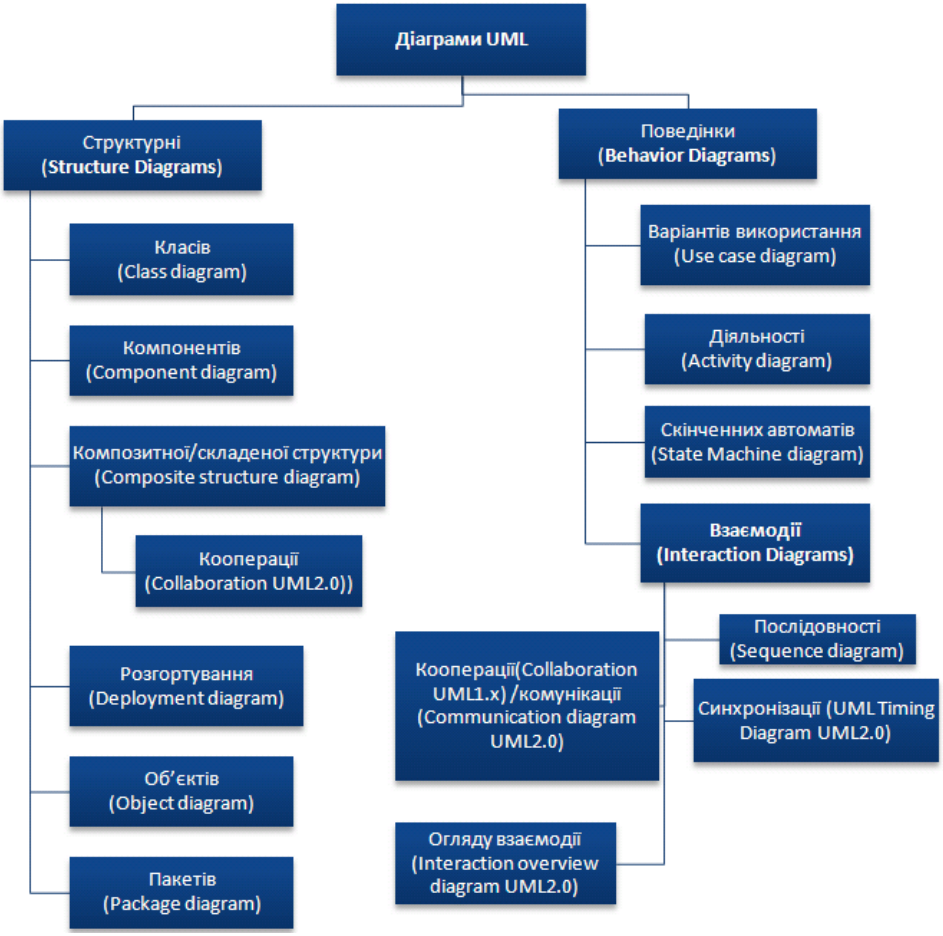


Рисунок 27 – Діаграми мови UML

Методології розроблення ПЗ

Найбільш помітними є такі методології розроблення ПЗ:

- *Rational Unified Process (RUP),* що пропонує ітеративну модель розроблення, що включає чотири фази: початок, дослідження, побудову та впровадження [33]. Проходження через чотири основні фази називається циклом розроблення, кожен цикл завершується генерацією працездатної версії системи. У ході супроводу продукт продовжує розвиватися і знову проходить ті самі фази. Розроблення ПЗ на базі RUP базується на створенні та супроводі моделей на базі UML.
- *Microsoft Solution Framework (MSF)* – методологія, що розроблялася для підвищення керованості процесів розроблення окремого підприємства (Microsoft), а потім стала застосовуватися розробниками, які використовують продукти Microsoft. Методологія, подібна до RUP, також включає чотири фази: аналіз, проектування, розроблення, стабілізацію, є ітераційною, припускає використання об'єктно-орієнтованого моделювання. MSF порівняно з RUP здебільшого орієнтована на розроблення бізнес-додатків [34].
- *eXtreme Programming (XP)* – методологія, що приділяє основну увагу ефективній комунікації між замовником і виконавцем упроовж усього проекту з розроблення ІС для відслідковування зміни вимог [19]. В основу підходу покладена командна робота та постійне тестування прототипів.
- *Гнучке розроблення ПЗ (Agile)* – клас методологій розроблення програмного забезпечення на основі ітеративної моделі ЖЦ, в якій вимоги та рішення еволюціонують через співпрацю між самоорганізовуваними командами [35].

Методологія Rational Unified Process (RUP)

Уніфікований процес розроблення ПЗ *Rational Unified Process* (RUP) [33] виник як відповідь на потребу розробників у

процесі, що об'єднує безліч аспектів розроблення програм і відповідає таким вимогам:

- забезпечує керівництво діяльністю команди;
- керує завданнями окремого розробника і команди в цілому;
- показує, які артефакти необхідно розробити;
- надає критерії відстеження та вимірювання продуктів і функціонування проекту.

Уніфікований процес розроблення ПЗ передбачає розподіл ЖЦ програмного забезпечення на фази (рис.28).

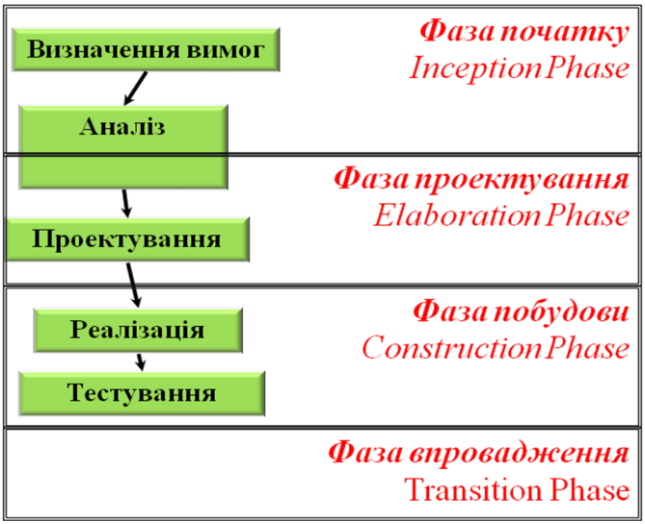


Рисунок 28 – Робочі процеси ЖЦ в уніфікованому процесі розроблення ПЗ

Специфіка уніфікованого процесу полягає у трьох словосполученнях – керований варіантами використання, архітектурно-орієнтований, ітеративний та інкрементний.

Уніфікований процес керується варіантами використання

Програмна система створюється для обслуговування її користувачів. В уніфікованому процесі поняття ***користувач*** стосується когось чи чогось (наприклад, іншої системи,

зовнішньої щодо даної системи), що взаємодіє із системою, яка розробляється. У відповідь на вплив користувачів система здійснює послідовність дій, які забезпечують користувачеві відчутний і значущий для нього результат. Користувача в RUP називають ***актором*** *(actor)* – категорія користувачів, що використовує певну частину функцій системи (відповідного варіанта використання).

Взаємодія актора із системою називається варіантом використання, або прецендентом. ***Варіант використання (ВВ),*** або ***прецендент,*** *(use case)* – це частина функціональності системи, необхідна для отримання користувачем значущого для нього, відчутного і вимірюваного результату. ВВ забезпечують функціональні вимоги. Сума всіх ВВ становить ***модель ВВ*** *(use-case model),* що описує повну функціональність системи. Ця модель заміняє традиційно опис функцій системи. На основі прецедентів в уніфікованому процесі будуються інші моделі для кожного етапу ЖЦ (рис.29).

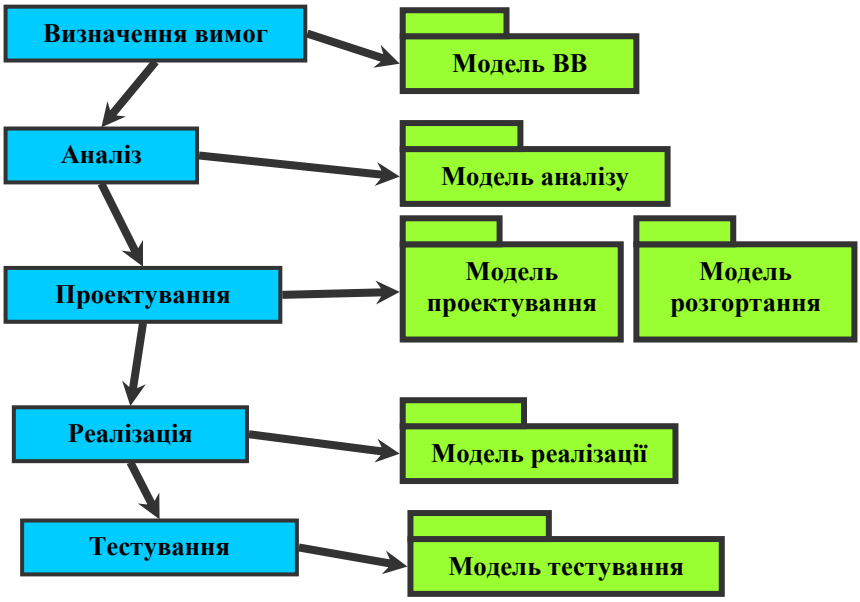


Рисунок 29 – Моделі ПЗ на кожному етапі ЖЦ

За набором реалізацій ВВ будується концептуальна модель системи – ***модель аналізу*** *(analysis model),* що є детальним описом усіх вимог та потрібних для реалізації робіт. Ґрунтуючись на моделі ВВ, розробники створюють серію моделей проектування і реалізації, які здійснюють прецеденти.

Модель проектування *(design model)* є об'єктною моделлю, що містить опис фізичної реалізації ВВ та сконцентрована на тому, які функціональні та нефункціональні вимоги разом із обмеженнями середовища розроблення реалізують систему.

Модель розгортання *(deployment model)* – визначає фізичний розподіл системи по вузлах, перевіряє, чи можуть ВВ бути реалізовані у вигляді компонентів, які виконуються в цих вузлах.

Модель реалізації *(implementation model)* – дає опис, як елементи моделі проектування реалізуються у вигляді компонентів, таких, як файли із кодом програм та виконувані файли.

Модель тестування *(test model)* – дає опис, як виконувані компоненти моделі реалізації тестуються на цілісність та проходять системні тести.

Тестери тестують реалізацію для того, щоб гарантувати, що компоненти моделі реалізації правильно виконують варіанти використання.

Таким чином, прецеденти не лише ініціюють процес розроблення, але й слугують для зв'язку окремих його частин. Процес розроблення проходить серії робочих процесів, кожен з яких ініціюється ВВ: прецеденти визначаються, вони проектуються, і врешті-решт варіанти використання є вхідними даними, за якими тестери створюють тестові приклади. Оскільки прецеденти дійсно керують процесом, вони не виділяються ізольовано, а розробляються в парі з архітектурою системи. Відповідно і архітектура системи, і ВВ розвиваються впродовж життєвого циклу ПЗ.

Уніфікований процес, орієнтований на архітектуру

Архітектура програми включає в себе найбільш важливі статичні і динамічні аспекти системи та будується на основі вимог до результату. Як було відмічено вище, вимоги відбиваються у варіантах використання. Однак прецеденти також залежать від безлічі інших чинників, таких, як вибір платформи для роботи програми (тобто комп'ютерної архітектури, операційної системи, СКБД, мережевих протоколів), доступність готових блоків багаторазового використання (наприклад, каркасу GUI), спосіб розгортання, успадковані системи і нефункціональні вимоги (наприклад, продуктивність і надійність). ***Архітектура ПЗ*** – це представлення всього проекту із виділенням важливих характеристик без акценту на деталях.

Кожен продукт має функції і форму. Одне без іншого не існує. Функції відповідають ВВ, а форма - архітектурі. В реальних умовах архітектура і прецеденти розробляються паралельно. Архітектура повинна бути спроектована так, щоб дозволити системі розвиватися не тільки в момент початкового розроблення, але і в майбутніх поколіннях. Щоб знайти таку форму, архітектор повинен працювати, повністю розуміючи ключові функції, тобто ключові варіанти використання системи. Ці ключові варіанти використання становлять 5-10% усіх ВВ, але вони дуже важливі, оскільки містять функції ядра системи. Архітектор виконує такі кроки:

- визначає платформу системи – створює грубий ескіз архітектури, починаючи з тієї частини, що не пов'язана з ВВ. Хоча ця частина архітектури не залежить від прецедентів, архітектор повинен у загальних рисах розуміти варіанти використання до створення ескізу архітектури;
- далі архітектор працює із підмножиною виділених ВВ, кожен з яких відповідає одній із ключових функцій розроблюваної системи. Кожен з обраних прецедентів детально описується і реалізується в поняттях підсистем, класів і компонентів;

- після того як ВВ описані та повністю розроблені, більша частина архітектури досліджена. Створена архітектура, у свою чергу, буде базою для повного розроблення інших ВВ. Цей процес продовжується до того часу, поки архітектура не буде визнана стабільною.

Уніфікований процес є ітеративним та інкрементним

Процеси розроблення комерційного ПЗ є серйозними, часто тривалими проектами. Для підвищення керованості та якості процесу доцільно розділити таку роботу на невеликі частини. Кожен міні-проект є ітерацією, результатом якої буде приріст. *Ітерації* належать до кроків робочих процесів, а *приріст (інкремент)* - до виконання проекту. Для максимальної ефективності ітерації повинні вибиратися і виконуватися за планом, що дозволяє вважати кожну ітерацію міні-проектом. Ітеративність передбачає повторення робіт з розроблення у різних фазах (рис.30).

Розробники вибирають завдання, які повинні бути вирішені у ході ітерації, враховуючи два фактори:

- у ході ітерації необхідно працювати з групою ВВ, що підвищує можливість застосування продукту під час подальшого розроблення;
- у ході ітерації необхідно займатися найбільш серйозними ризиками.

Послідовні ітерації використовують артефакти розробки в тому вигляді, в якому вони залишилися при закінченні попередньої ітерації. У ході кожного міні-проекту для обраних ВВ проводиться послідовне розроблення - аналіз, проектування, реалізація та тестування. Зрозуміло, прирощення необов'язково адитивне, особливо на ранніх фазах життєвого циклу. На кожній ітерації розробники визначають і описують відповідні ВВ, створюють проект, що використовує обрану архітектуру як напрямну, реалізують проект у вигляді компонентів та перевіряють відповідність компонентів варіантами використання. Якщо ітерація досягла своєї мети, процес розроблення переходить на наступну ітерацію. Якщо ітерація не

виконала свого завдання, розробники повинні переглянути свої рішення і спробувати інший підхід.

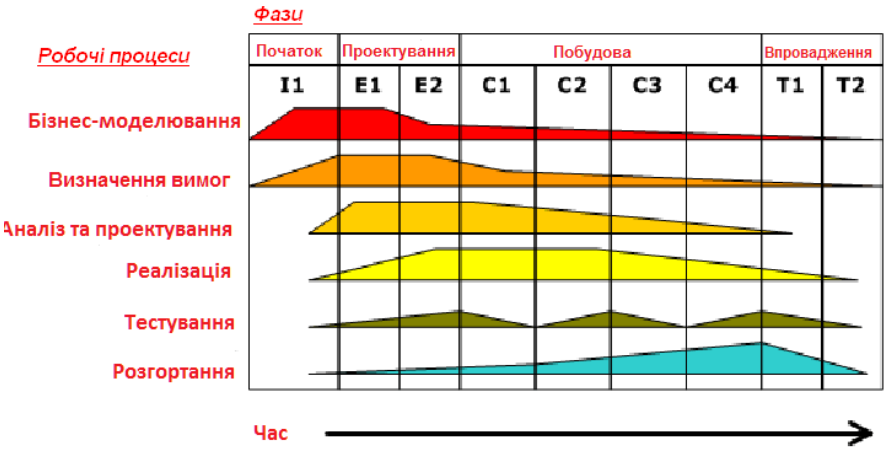


Рисунок 30 – Ітеративність уніфікованого процесу розроблення ПЗ

Для отримання максимальної економії команда, що працює над проектом, повинна вибирати тільки ті ітерації, які потрібні для досягнення мети проекту. Для цього необхідно розмістити ітерації в логічній послідовності. Непомічені раніше проблеми приведуть до збільшення ітерацій або зміни їх послідовності, а це збільшує кількість зусиль і часу для виконання процесу розроблення. Мінімізація непомічених проблем є однією з цілей зниження ризиків.

Керований ітеративний процес має такі переваги [32]:

- керована ітерація обмежує фінансові ризики витратами на одне прирощення. Якщо розробникам потрібно повторити ітерацію, організація втрачає лише зусилля, витрачені на одну ітерацію, а не вартість усього продукту;
- керована ітерація знижує ризик непостачання продукту на ринок у заплановані терміни. При ранньому виявленні відповідного ризику час, що витрачається на його нейтралізацію, вноситься в план на ранніх стадіях, коли співробітники менш завантажені, ніж у кінці планового

періоду. При традиційному підході, коли серйозні проблеми вперше проявляються на етапі тестування системи, час, необхідний для їх усунення, як правило, перевищує час, що залишився за планом для завершення всіх робіт, що майже завжди приводить до затримок поставок;

- керована ітерація прискорює темпи процесу розроблення в цілому, оскільки для більш ефективної роботи розробників і швидкого отримання ними гарних результатів короткий і чіткий план ефективніший;
- керована ітерація визнає, що бажання користувачів та пов'язані з ними вимоги не можуть бути чітко визначені на початку розроблення. Вимоги уточнюються в послідовних ітераціях, що полегшує адаптацію до змін вимог.

Усі три характеристики уніфікованого процесу – керований варіантами використання, архітектурно-орієнтований, ітеративний та інкрементний процес розроблення – однаково важливі. Архітектура надає структуру, що направляє роботу в ітераціях, у кожній з яких варіанти використання визначають цілі та спрямовують роботу.

Моделі уніфікованого процесу розроблення ПЗ

Модель варіантів використання (use-case model) є концептуальною моделлю системи, що описує повну функціональність системи. Для побудови моделі варіантів використання в команді розробників виділяються співробітники, які виконують відповідні функції та створюють певні артефакти (рис.31). На рис.32 наведено схему робочого процесу розроблення моделі ВВ.

Співробітник – позначення ролі, що доручена одній або кільком особам, визначає потрібні для цієї дії якості та здібності.

Артефакт *(artifact)* – частина інформації, що створюється, змінюється або використовується співробітником під час роботи системи, визначає область відповідальності та дає змогу керувати своїми версіями.

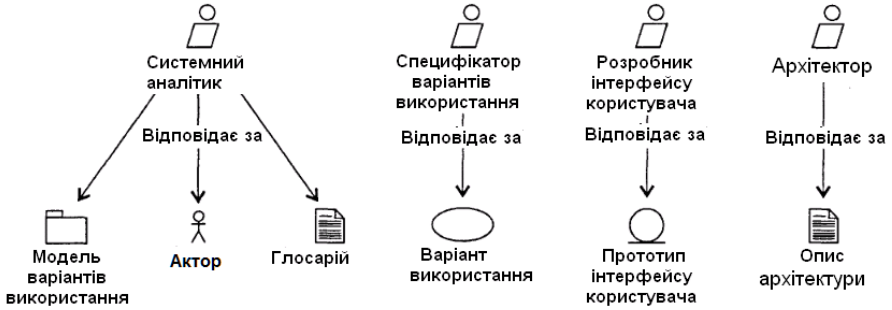


Рисунок 31 – Співробітникі, які будують модель варіантів використання

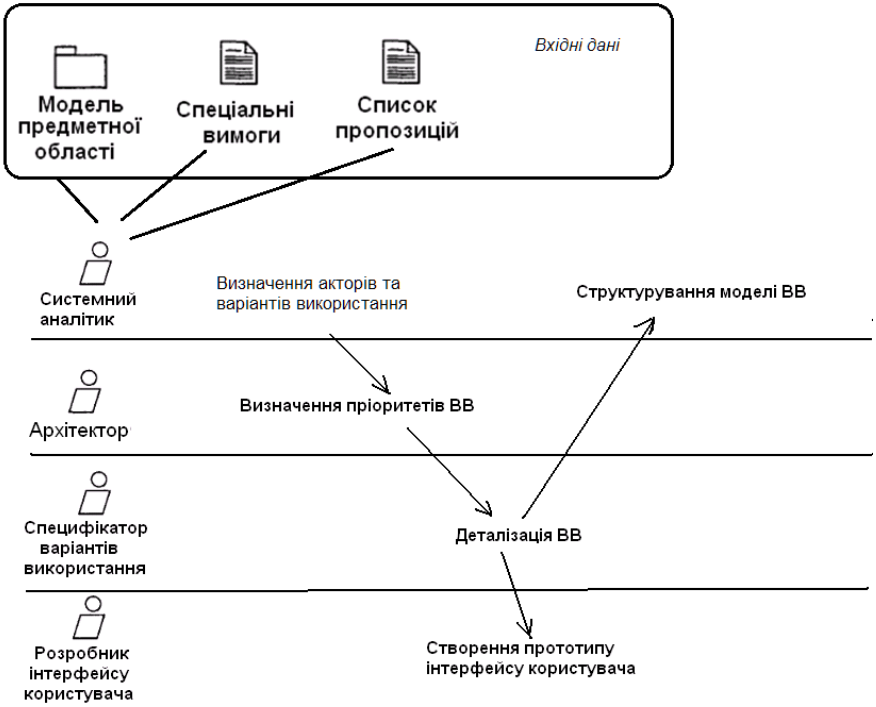


Рисунок 32 – Робочий процес побудови моделі ВВ

Модель аналізу *(analysis model)* будується на базі моделі ВВ для більш точного розуміння вимог, створення їх простого

опису та для виявлення структури системи, в тому числі архітектури. На рис.33, 34 показані учасники процесу аналізу, створювані артефакти та схема робочого процесу розроблення моделі

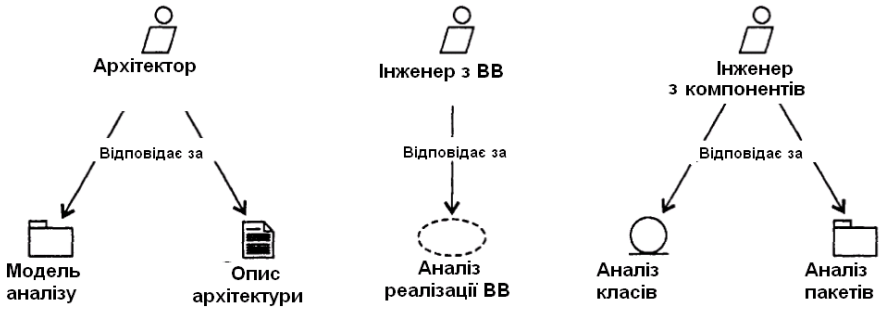


Рисунок 33 – Співробітники, які будують модель аналізу

У табл.3 подано порівняння моделі аналізу із моделлю ВВ, з якого видно, що модель аналізу використовується розробниками, тоді як модель ВВ повинна бути зрозумілою і замовникові, і розробникам.

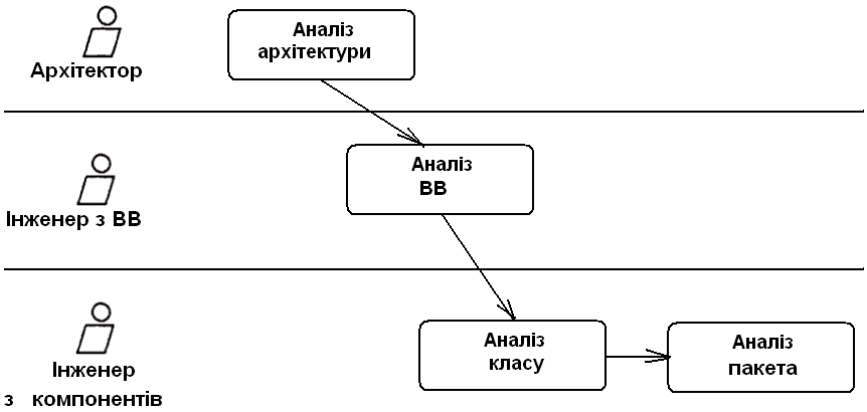


Рисунок 34 – Робочий процес побудови моделі аналізу

Таблиця 3 – Зіставлення моделі аналізу та моделі варіантів використання

Характеристика	**Модель ВВ**	**Модель аналізу**
Використовує мову	замовника	розробника
Містить вигляд системи	зовнішній	внутрішній
Модель структурована за	варіантами використання	класами та пакетами
Використовується	для домовлення між замовником та розробниками	розробниками для розуміння системи
Містить надлишковість, несумісні вимоги	так	ні
Визначає	функції системи та ВВ для подальшого аналізу	як функції реалізуються у реалізаціях ВВ

Модель проектування *(design model)* є об'єктною моделлю, сконцентрованою на вимогах, функціональних та нефункціональних, які разом із обмеженнями середовища розроблення реалізують систему.

Зіставлення моделей проектування та аналізу (табл.4) показує, що модель проектування є формалізованим описом проекту системи, сконцентрованим на її реалізації.

Модель проектування має такі *властивості*:

- має ієрархічну структуру;
- реалізації ВВ є стереотипами кооперації;
- модель є кресленням реалізації.

Модель проектування розробляється разом із моделлю розгортання, оскільки проект системи неможливо створити без визначення фізичного розподілу системи за розрахунковими вузлами, що містить **модель розгортання** *(deployment model).* Також ця модель перевіряє, чи можуть ВВ бути реалізовані у вигляді компонентів, які виконуються у цих вузлах.

Таблиця 4 – Зіставлення моделей проектування та аналізу

Характеристика	**Модель аналізу**	**Модель проектування**
Тип моделі	концептуальна	фізична
Залежність	від проекту	від реалізації
Рівень формалізації	низький	високий
Кількість стереотипів класів	Три – сутність (entity), керування (control), межа (boundary)	будь-яка
Розмір	невеликий	значний
Детальність	ескіз проекту системи	опис проекту системи з увагою до послідовності
Підтримка у ЖЦ	може не підтримуватись	підтримується весь ЖЦ
Визначає	структуру	систему, зберігаючи структуру моделі аналізу

Розробники моделей проектування та розгортання, створювані ними артефакти наведені на рис.35.

Рисунок 35 – Співробітники та артефакти моделі проектування

Модель проектування розробляється відповідно до заданого порядку (рис.36):

1. Ідентифікуються класи.
2. Виділяються відповідальності.
3. Проектуються класи та реалізації ВВ.
4. Класи проектування збирають у підсистеми
5. Визначають інтерфейси між підсистемами.

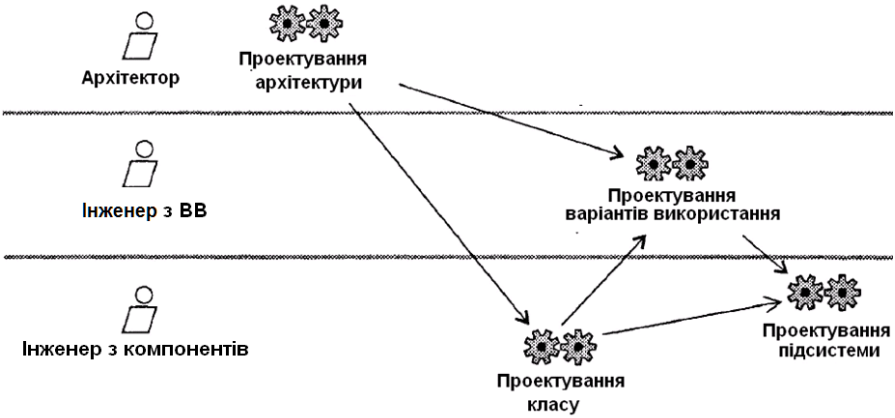


Рисунок 36 – Робочий процес побудови моделі проектування

Модель реалізації *(implementation model)* дає опис реалізації моделі проектування у вигляді компонентів програмного продукту. Елементи моделі та її розробники наведені на рис. 37.

Рисунок 37 – Співробітники та артефакти моделі реалізації

В уніфікованому процесі створення ПЗ передбачається покрокове розроблення. Результатом кожного кроку є ***білд*** *(build)* – виконувана версія системи.

Після визначення складових компонентів моделі створюється опис інтерфейсів їх взаємозв'язку та розробляється ***план складання***, що дає опис послідовності ітерацій. Для кожного білду план містить опис:

- функцій, які потрібно реалізувати у білді (це перелік ВВ та/або сценаріїв або їх частин);
- частини моделі реалізації, що стосуються білду (перелік підсистем та компонентів, потрібних для реалізації функціональності).

У результаті виконання робочого процесу будується модель реалізації системи (рис.38).

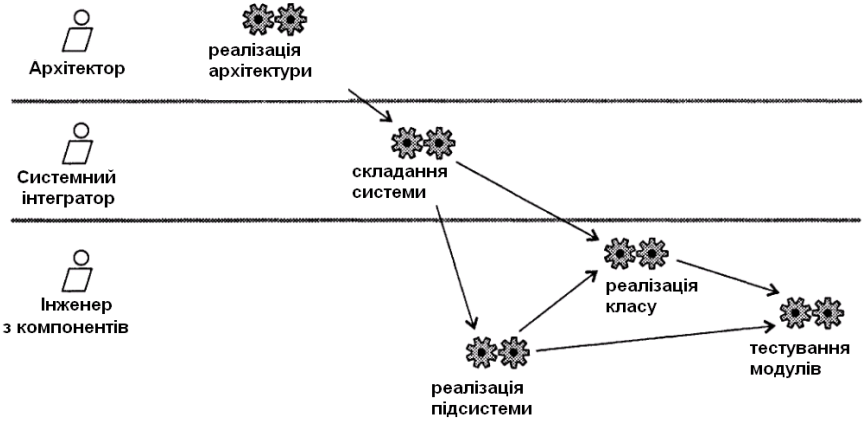


Рисунок 38 – Робочий процес побудови моделі реалізації

Модель тестування *(test model)* описує, як виконувані компоненти моделі реалізації тестуються на цілісність та проходять системні тести. У розробленні моделі тестування задіяні чотири співробітники (рис. 39,40,41) – інженер з тестування (розробник тестів), інженер з компонентів, тестувальник цілісності та системний тестувальник.

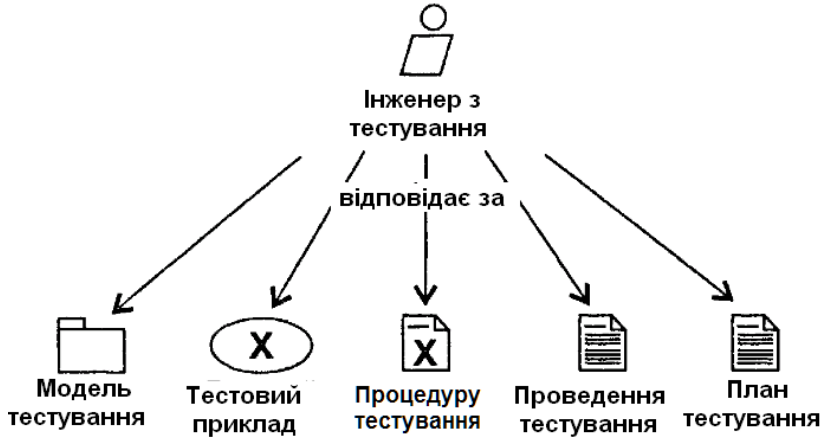


Рисунок 39 – Артефакти, розроблювані інженером з тестування

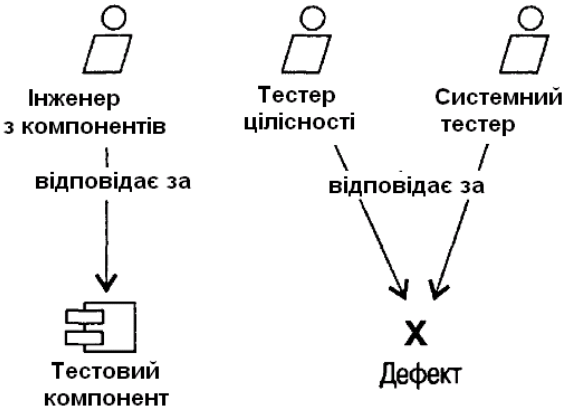


Рисунок 40 – Артефакти моделі тестування, розроблювані інженером з компонентів, тестувальником цілісності та системним тестувальником

Модель тестування розробляється (рис. 39) з урахуванням вимоги, що кожен білд є об'єктом тестування і керується системою контролю версій системи. У моделі тестування розробляються ***тестові приклади*** – шляхи тестування системи, що містить предмет тестування, вхідні дані, результат та умови тестування, – та ***тестові процедури*** – методики запуску

одного/кількох тестових прикладів або їх частин. Обов'язково складається ***план тестування***, що містить опис стратегії тестування, виділених ресурсів та графіка робіт.

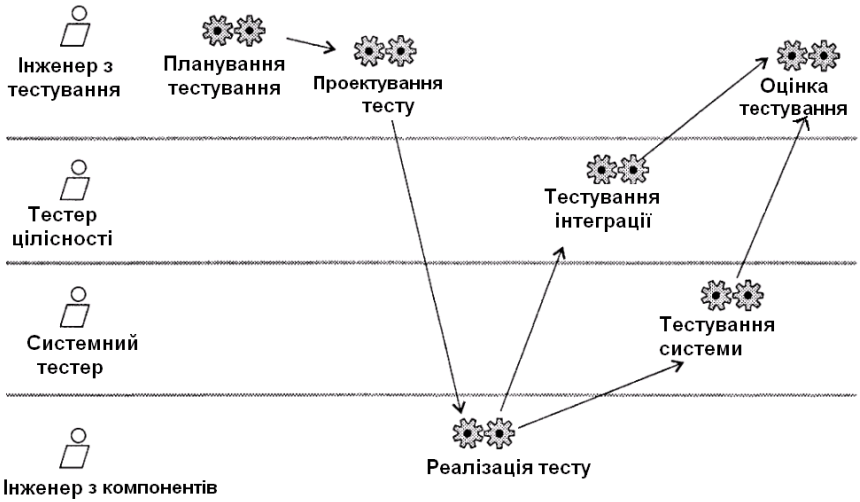


Рисунок 41 – Робочий процес побудови моделі тестування

Методологія RUP реалізована у програмному продукті Ration Software Architect компанії IBM. Як CASE-система, Ration Software Architect має велику кількість інструментів, корисних для підтримки зв'язку перших етапів проектування з етапом реалізації програм (кодування), а також з етапом оцінки. Зокрема перевіряється, що моделювання на різних етапах погоджено, що модельні угоди, визначення класів, інших елементів моделей та їх взаємозв'язку несуперечливі. Рівень автоматичного аналізу високий настільки, що дозволяє будувати за моделями так звані реалізації за замовчуванням. Це заготовки програмного коду, що включають у себе описи класів та їх методів у тому вигляді, що можна зробити з моделей. Програміст доповнює заготовки фрагментами, деталізує конкретну реалізацію.

У Ration Software Architect та інших UML CASE-системах підтримується побудова реалізацій за замовчуванням

за моделями загального, а не спеціального призначення. Реалізація за замовчуванням є лише одним із прийомів підтримки зв'язків між етапами життєвого циклу розроблення програмного забезпечення. Саме ідея комплексної підтримки зв'язаності робочих продуктів різних етапів, а не окремі прийоми, які з'являлися і раніше, - головне для даної CASE-системи. Програмне втілення цієї ідеї, нехай навіть з істотними недоробками, необхідно віднести до явних переваг даного інструментарію.

Методологія Microsoft Solution Framework (MSF)

На цей час абсолютна більшість користувачів в Україні працюють на продуктах компанії Microsoft. Ця компанія підтримує університети по усьому світові, надає ряд ПЗ для підтримки основних процесів господарської діяльності. Тому при вивченні технологій програмування не можна пропустити досвід Microsoft у галузі створення програм.

Аналіз результатів виконання програмних проектів, виконаний службою Microsoft Consulting Services, виявив, що лише чверть проектів можна визнати успішними, половина проектів мала великі проблеми (рис. 42).

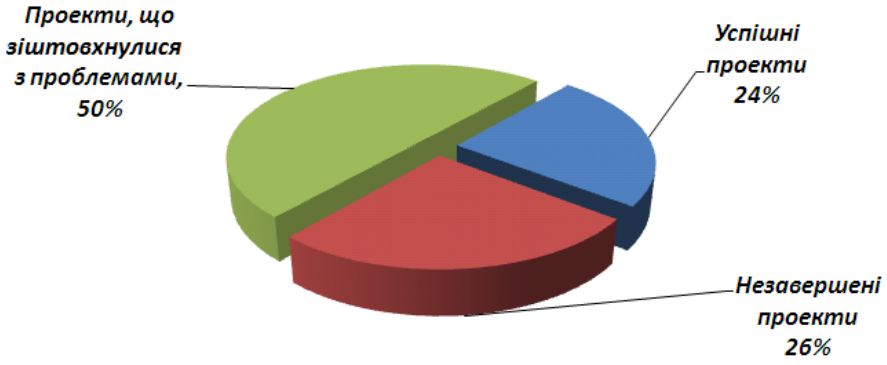


Рисунок 42 – Статистика успіхів проектів з розроблення ПЗ компанії Microsoft

Основними причинами невдач були визнані такі:

- постійна зміна вимог;
- нечіткі або неповні специфікації;
- низька якість коду;
- занадто широка постановка завдання;
- помилка в підборі кадрів;
- погана організація роботи;
- нечітко сформульовані цілі.

Для подолання цих труднощів був запропонований набір моделей Microsoft Solution Framework (MSF) [34], в якому врахований досвід, накопичений групами розроблення програмних продуктів.

В основу MSF покладено *вісім базових принципів (*foundational principles*)*:

- **Сприяння відкритої комунікації (Foster open communications).** Модель процесу MSF запроваджує вільний інформаційний потік сред членів команди та зацікавлених сторін (key stakeholders) для забезпечення однакового розуміння завдань. Документування ходу проекту та доступ до цих даних членів команди, зацікавлених сторін та замовників.
- **Робота у напрямку спільного бачення проекту (Work toward a shared vision)**. Модель процесу MSF запроваджує фазу формування концепції (Envisioning Phase) та окремий ключовий момент затвердження бачення (milestone Vision/Scope Approved) для формування спільного бачення проекту. Бачення включає детальне розуміння цілей та завдань для досягнення рішення поставленої проблеми. Спільне бачення виявляє припущення команди та замовників, потрібні для отримання рішення.
- **Надання прав і можливостей членам команди (Empower team members).** Розширення прав і можливостей членів команди для прийняття ними відповідальності за виконану роботу. Таке збільшення відповідальності може бути затверджене у графіках, де фіксується дата закінчення робіт,

що також може бути засобом виявлення можливих затримок проекту.

- **Визначення індивідуальної та спільної відповідальності (Establish clear accountability and shared responsibility).** Модель командної групи MSF ґрунтується на принципі важливості роботи кожного для отримання якісного рішення проблеми. Усі члени групи розділяють відповідальність за проект.
- **Зосередження на бізнес-цілях (Focus on delivering business value).** Рішення повинне приносити користь організації у вигляді додавання вартості бізнесу. Це додавання досягається тільки після повного розгортання рішення у виробничому середовищі.
- **Бути гнучкими, очікувати на зміни (Stay agile, expect change).** MSF припускає, що у виробничому середовищі на рішення постійно впливають зміни. Команда повинна бути обізнаною та готовою до керування змінами вимог.
- **Інвестування у якість (Invest in quality).** У MSF кожен член команди відповідальний за якість вирішення завдання.Для підтримки якості протягом проекту формується команда тестувальників. Це гарантує, що рішення відповідає рівню якості, визначеному зацікавленими сторонами.
- **Навчання за досвідом (Learn from all experiences).** MSF вимагає використання досвіду, отриманого у попередніх проектах. Це дозволяє знайти найкращі методики розроблення.

MSF складається із *двох моделей* та *трьох дисциплін*:

- моделі командної групи;
- моделі процесу;
- дисципліни управління проектами;
- дисципліни управління ризиками;
- дисципліни управління підготовкою.

Модель командної групи (MSF Team Model) описує, як організувати колективи і якими принципами керуватися для досягнення максимального успіху в розробленні програм. Різні колективи можуть по-своєму застосовувати на практиці різні

елементи цієї моделі – все залежить від масштабу проекту, розміру колективу і кваліфікації його учасників та моделі процесу розроблення.

Формування колективу є складним завданням, що повинне виконуватися психологами та враховувати такі основні положення:

- не повинне бути команди з одних лідерів;
- не повинне бути команди з одних виконавців;
- у випадку невдачі команда розформовується;
- система штрафів (якщо проект провалюється - карають усіх).

Модель командної групи визначає тільки ролі, кожна з яких може виконуватися кількома людьми (рис. 43). Цікаво, що в моделі командної групи не передбачено єдиноначальності – усі ролі важливі, усі ролі рівноправні, тому MSF називають моделлю рівних (team of peers).

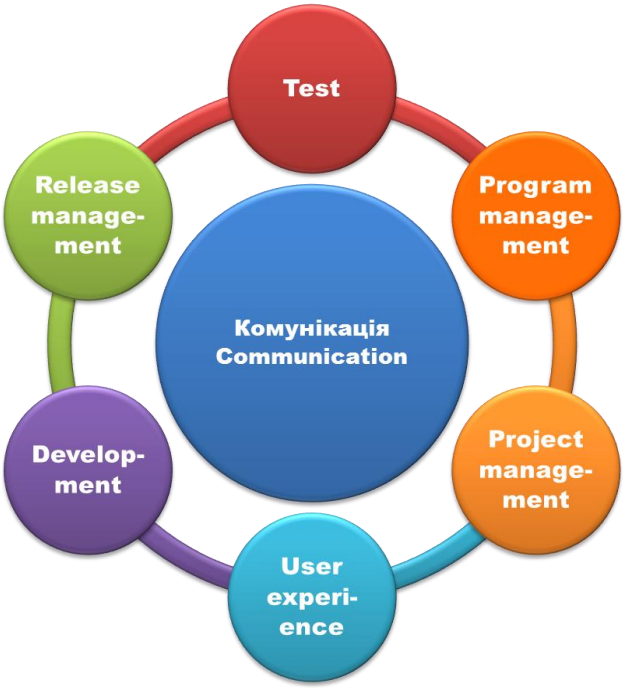


Рисунок 43 – Модель командної групи (MSF Team Model)

Program management – керування програмою. Виконавець цієї ролі відповідає за організацію (але не керує): здійснює ведення графіка робіт, ранкові 15-хвилинні наради, забезпечує відповідність стандартам і специфікаціям, фіксацію порушень, написання технічної документації.

Product management – керування продуктом. Виконавці цієї ролі відповідають за спілкування із замовником, написання специфікації, роз'яснення завдань розроблювачам.

Development – найбільш традиційна роль – ***розроблення і початкове тестування продукту***.

User expirience – підвищення ефективності роботи користувачів, написання користувальницької документації.

Release management – розгортання релізу продукту, супровід і його технічна підтримка.

Test – визначення відповідності показників якості релізу встановленим значенням. Виявлення й усунення недоробок, виправлення помилок, інші функції QA.

У MSF затверджується, що таку модель можна масштабувати, розбиваючи систему за функціями.

Модель процесу (MSF Process Model) визначає, коли і які роботи повинні бути виконані.

Основні принципи і практичні прийоми, на яких ґрунтується модель:

- ітеративний підхід (послідовний випуск версій);
- підготовка чіткої документації;
- урахування невизначеності майбутнього;
- облік компромісів;
- керування ризиками;
- підтримка відповідального відношення колективу до строків випуску продукту;
- розбивка великих проектів на більш дрібні керовані частини;
- щоденне складання проекту;
- постійний аналіз ходу робіт.

Process model має три *основні особливості*:

- розбивка всього процесу на фази;
- використання опорних точок (milestones);

- ітеративність.

Увесь процес розбивається на п’ять взаємозалежних фаз (рис. 44). Перш ніж переходити до наступної фази, на попередній повинні бути отримані певні результати (досягнуті головні опорні точки). Фактично процес ітеративний і відповідає спіральній моделі ЖЦ ПЗ.

Рисунок 44 – Модель процесу (MSF Process Model)

Envisioning Phase – вироблення єдиного розуміння проекту всіма членами колективу. Ця фаза закінчується розробленням формалізованого документа, що містить:

- *problem statement* – опис завдання на розроблення ПЗ обсягом не більше однієї сторінки;
- *vision statement* – опис того, від чого відштовхується розроблення і яким результатом закінчується;
- *solution concept* - що буде впроваджене в результаті вирішення поставленої проблеми;

- *user profiles* – опис потенційних користувачів системи;
- *business goals* – опис бізнес-функцій, виконання яких за допомогою розробленого ПЗ поверне інвестиції;
- *design goals* – конкретні цілі й обмеження програмного продукту, його конкретні властивості.

Planning Phase – планування чергового циклу розроблення:

- функціональні специфікації;
- план-графік робіт;
- оцінка ризиків.

Developing Phase – розроблення, причому рекомендуються різні технологічні прийоми, наприклад, повторне використання фрагментів коду, програмування за контрактом, написання захищеного від помилок ПЗ та ін.

Stabilizing Phase – створення стабільної β-версії, готової до використання.

Важливу роль відіграють *опорні точки (milestones)*, у яких аналізується стан робіт і виробляється їхня синхронізація. У цих точках додаток або його специфікації не заморожуються. Опорні точки дозволяють проаналізувати стан проекту і внести необхідні корективи, наприклад, переналагоджуватися під вимоги, що змінилися, замовника або відреагувати на ризики, можливі в ході подальшої роботи. Для кожної опорної точки визначається, які результати повинні бути отримані до цього моменту.

Кожна фаза процесу розроблення завершується *головною опорною точкою (major milestone)* – моментом, коли всі члени колективу синхронізують отримані результати. Призначення таких точок у тому, що вони дозволяють оцінити життєздатність проекту. Їхні результати видимі не тільки колективу розроблювачів, але й замовникові. Після аналізу результатів колектив розробників і замовник спільно вирішують, чи можна переходити на наступну фазу. Таким чином, головні опорні точки - це критерії переходу з однієї фази проекту на іншу.

Усередині кожної фази визначаються *проміжні опорні точки (interium milestones).* Вони, як і головні, слугують для

аналізу й синхронізації досягнутого, а не для заморожування проекту. Але на відміну від головних опорних точок проміжні видні тільки членам колективу розробників.

Ітеративність процесу MSF полягає в його багаторазовому повторенні упродовж усього циклу розроблення та існування продукту. На кожній успішній ітерації у продукт включаються тільки ті нові інструменти та функції, які задовольняють постійно змінювані вимоги бізнесу.

Методологія eXtreme Programming (XP)

Екстремальне програмування *(eXtreme Programming, XP)* – спрощена методологія організації виробництва для невеликих і середніх за розміром команд розробників, які займаються розробленням програмного продукту в умовах незрозумілих або швидко змінних вимог [19].

Програмування відповідно до методик XP доводить використання загальноприйнятих принципів програмування до екстремальних рівнів:

- перегляд коду виконується постійно (з урахуванням того, що програмування ведеться парами);
- кожен учасник проекту тестує код програми постійно (тестування модулів), навіть замовники проводять функціональне тестування;
- проектування є складовою частиною повсякденної роботи кожного розробника (перероблення коду);
- розроблення виконується з урахуванням вимоги збереження в системі найбільш простого дизайну, що забезпечує поточний необхідний рівень функціональності (простіші речі надійніші в роботі);
- увага до архітектури системи на кожному етапі проекту;
- інтеграційне тестування виконується після кожної найменшої зміни у системі (триваюча інтеграція);
- ітерації невеликі – тривають години, кілька днів (постійне планування).

У розділі конспекту «Керування та організація робіт» наведені основні ризики розроблення ПЗ, виділені К.Беком. Для

зменшення кількості можливих ризиків та усунення їх негативного впливу і розроблена методологія ХР. У табл.5 наведені найпоширеніші ризики розроблення та методи їх усунення за допомогою екстремального програмування.

Таблиця 5 – Методи усунення ризиків розроблення ПЗ у методології ХР

Ризик	Метод подолання у ХР
Зміна графіка	Планується короткострокове розроблення версій ПЗ
Закриття проекту	Замовник визначає найменший перелік найважливіших функцій системи, які визначають якість системи
Система втрачає корисність	Постійне розроблення та виконання тестів над системою не дозволяють накопичуватися дефектам, що впливають на роботу системи
Велика кількість дефектів і недоліків	Тести постійно створюються і виконуються не лише розробниками, а й замовниками, які перевіряють функціональність системи
Невідповідність ПЗ розв'язуваній проблемі	Представник замовника є постійним членом команди розробників. Специфікація постйно перевіряється і змінюється за потреби
Зміна характеру бізнесу	Бізнес не встигає помітно змінити напрямок діяльності у ході короткострокового розроблення версії програми
Нестача функціональних можливостей	Задачі із найбільшим пріоритетом реалізуються у першу чергу
Плинність кадрів	Програмісти самостійно оцінюють обсяг робіт та термін виконання, що підвищує їх зацікавленість

Зміна графіка робіт є найпоширенішою проблемою під час створення ПЗ. Неуважність до розтягування робіт може

зірвати увесь проект і стати причиною усіх інших ризиків. *На кожну версію* програми при використанні ХР *виділяється один-два місяці*. У рамках кожної версії планується кілька *ітерацій* для отримання запланованої функціональності, кожна з яких *триває один-чотири тижні*. *Замовник обов'язково перевіряє функціональність*, отриману в ході чергової ітерації, тобто постійно забезпечується зв'язок із замовником, що отримує уявлення про поточний стан робіт. Кожна ітерація розділяється на кілька *задач*, які *тривають кілька днів*. *Найважливіші завдання реалізуються в першу чергу.* Тобто при зміщенні строків робіт можна бути впевненим, що не робробленими залишилися завдання низького пріоритету. Використання коротких етапів проекту для розроблення чергової версії програми дозволяє гнучко керувати строками робіт.

Для формування стилю розроблення, що дозволить досягти потрібної якості рішення, в ХР пронується керуватись чотирма цінностями [19]:

- ***комунікацією*** *(communication)* – дисципліна ХР, спрямована на забезпечення безперервної комунікації усіх учасників проекту. Під час тестування, програмування в парах та попереднього оцінювання робіт замовники, розробники та менеджери змушені постійно спілкуватись;
- ***простотою*** *(simplicity)* – неможливо заздалегідь точно визначити, як буде розвиватися проект, тому вирішувати необхідно лише завдання, що виникають сьогодні, у найбільш простий спосіб;
- ***зворотним зв'язком*** *(feedback)* – спосіб отримати точні та конкретні дані про стан проекту. Постійне виконання тестів для перевірки усіх змін у системі забезпечує програміста звороним зв'язком про якість роботи. Кожен новий опис замовником вимог до системи одразу ж оцінюється розробниками і забезпечує замовника зворотним зв'язком із інформацію про якість опису. Менеджер, який контролює строки виконання робіт, забезпечує усіх учісників проекту інформацією про виконання планових термнів розроблення. В ХР найбільш важливі функції системи реалізуються в

першу чергу у реально працюючий продукт, тому замовник досить швидко може оцінити хід виконання та якість розроблення, а також відповідність замовленню;

- ***хоробрість*** *(courage)* – для того, щоб розроблення не втратила своєї актуальності, в ХР рішення повинні прийматися з максимальною швидкістю, для чого потрібна певна сміливість.

Для визначення методів вирішення проблем розроблення ПЗ на основі обраних цінностей потрібно керуватися такими *фундаментальними принципами* оцінки методів вирішення поставленого завдання:

- *швидкий зворотний зв'язок* – інформація про стан системи повинна якомога швидше надходити до зацікавлених сторін проекту. У рамках дисципліни ХР ці дані повинні із максимальною швидкістю надходити, інтерпретуватися та на основі їх аналізу швидко виконуватися модифікації системи;
- *прийнятна простота* – кожна проблема повинна вирішуватись у найбільш простий спосіб. У рамках ХР значні зусилля (тестування, переробка коду, комунікації) покладаються для вирішення завдань сьогодення таким чином, щоб завтра внесення змін не потребувало великих зусиль;
- *поступова зміна* – кардинальні зміни часто приводять до провалу проекту, щоб заплановані модифікації мали успіх, їх потрібно реалізовувати як серію невеликих змін, після кожної з яких перевіряється працездатність системи;
- *прийнятна зміна* – найвигідніша стратегія – та, що дозволяє вирішити найбільш важливу проблему і залишає максимальну свободу дій;
- *якісна робота* – кожен учасник проекту повинен прагнути максимально якісно виконувати свої завдання і сприяти якісній роботі інших.

Кожен принцип втілює заявлені цінності. З-поміж альтернативних методів, які втілюють принципи, в ХР пропонується обирати метод рішення, який відповідає виконанню більшості принципів.

У рамках XP пропонуються методи для виконання чотирьох основних видів діяльності у ході розроблення ПЗ:

Кодування – від якості коду залежать робота та надійність системи. Якісне кодування потребує постійного навчання та удосконалення. Також код містить інформацію про стан проекту у найльш стислій та чітко зрозумілій формі;

Тестування – постійне тестування зменшує вартість внесення змін у розроблення та зменшує ризики, пов'язані із змінами. Виконання усіх запланованих тестів свідчить про успішне завершення чергової версії або усього проекту;

Слухання – відкрите чесне спілкування – як між розробниками, так і між розробниками і замовниками. Програмісти повинні почути бізнес-вимоги замовника, щоб виконати завдання, а замовник повинен чути зауваження розробників, щоб краще розуміти свої потреби;

Проектування – проект системи впливає на якість ПЗ і на легкість внесення змін. Правильний проект створює структуру, що організовує логіку системи. Логічно пов'язані фрагменти ПЗ повинні поєднуватись у незалежні частини системи. При правильному дизайні програми внесення змін в одному елементі системи не буде потребувати змін у інших.

Для розв'язання виділених проблем розроблення в методології XP використовуються такі методики [19]:

Гра в планування *(planning game)* – швидко визначає обсяг робіт, що необхідно виконати у наступній версії. Для цього розглядаються бізнес-пріоритети та технічні оцінки. Якщо з часом план перестає відповідати дійсності, відбувається оновлення плану. Замовники повинні приймати рішення стосовно обсягів робіт, пріоритетності завдань та строків випуску версій. Розробники повинні відповідати за оцінку часу, потрібного на реалізацію вимог, наслідки прийнятих рішень, організацію процесу розроблення та детальне планування робіт.

Невеликі версії *(small releases)* – найперша спрощена версія системи, що реалізує найбільш важливу функціональність, швидко вводиться в експлуатацію. Наступні версії випускаються через відносно короткі проміжки часу.

Метафора *(metaphor)* – проста загальнодоступна і загальновідома історія, що коротко описує, як працює вся система. Ця історія керує усім процесом розроблення.

Простий дизайн *(simple design)* – у кожен момент часу система повинна бути спроектована так просто, як це можливо. Виявлена надмірна складність усувається одразу.

Тестування *(testing)* – програмісти постійно пишуть тести для модулів, а замовники – тести, які демонструють працездатність і завершеність тієї чи іншої можливості системи. Умова продовження розроблення – усі тести спрацьовують без помилок.

Переробка *(refactoring)* – програмісти реструктурують систему, не змінюючи її поведінки. При цьому вони усувають дублювання коду, покращують комунікацію, спрощують код і підвищують його гнучкість.

Програмування парами *(pair programming)* – увесь розроблювальний код пишеться двома програмістами на одному комп'ютері.

Колективне володіння *(collective ownership)* – у будь-що момент часу будь-що член команди може змінити будь-що код у будь-якому місці системи.

Безперервна інтеграція *(continuous integration)* – система інтегрується і збирається кожного разу, коли завершується рішення чергового завдання (можливо кілька разів за день).

40-годинний тиждень *(40-hour week)* – програмісти працюють не більше 40 годин на тиждень. Ніколи не можна працювати понаднормово два тижні поспіль.

Замовник на місці розроблення *(on-site customer)* – до складу команди входить реальний користувач системи. Він доступний упродовж усього робочого дня і здатний відповідати на запитання про систему.

Стандарти кодування *(coding standards)* – програмісти пишуть увесь код у відповідності до правил, які забезпечують комунікацію за допомогою коду.

Гнучке розроблення ПЗ на основі Agile

Постійна зміна вимог під час розроблення ПЗ, необхідність забезпечення ефективної співпраці команди розробників потребували методів, які б забезпечили якість розроблення та супроводу програмних систем і уникнули недоліків занадто формалізованих методів, більшість з яких базувалась на каскадній моделі ЖЦ. У 90-х рр. XX ст. активно стали виникати різноманітні підходи до створення ПЗ, які забезпечували гнучку роботу програмістів. У результаті у 2001 році був написаний **Маніфест гнучкого розроблення** *(Agile manifesto),* що зафіксував цінності ефективних підходів до розроблення програмних продуктів, таких, як XP, Feature driven development, Scrum, Adaptive software development, Pragmatic Programming (прагматичне програмування). Текст маніфесту [35] підписали 17 найавторитетніших фахівців у цій галузі діяльності – Кент Бек (Kent Beck), Алістер Коуберн (Alistair Cockburn), Мартін Фаулер (Martin Fowler) та інші:

Agile-маніфест розроблення програмного забезпечення

Ми постійно відкриваємо для себе досконаліші методи розроблення програмного забезпечення, займаючись розробленням безпосередньо та допомагаючи у цьому іншим.

Завдяки цій роботі ми змогли зрозуміти, що:

***Люди та співпраця** важливіші за процеси та інструменти;*
***Працюючий продукт** важливіший за вичерпну документацію;*
***Співпраця із замовником** важливіша за обговорення умов контракту;*
***Готовність до змін** важливіша за дотримання плану.*

Хоча цінності, що справа, важливі, ми все ж цінуємо більше те, що зліва.

У результаті з'явився термін – ***Гнучке розроблення програмного забезпечення*** *(Agile software development)* – клас методологій розроблення програмного забезпечення, що

базуються на ітеративному розробленні, в якому вимоги та рішення еволюціонують через співпрацю між самоорганізовуваними багатофункціональними командами.

У маніфесті озвучені основні принципи гнучкого розроблення ПЗ:

- Найвищий пріоритет – *задоволення потреб замовника* шляхом завчасного та регулярного постачання програмного забезпечення.
- *Схвальне ставлення до змін*, навіть на завершальних стадіях розроблення. Agile-процеси використовують зміни для забезпечення конкурентоспроможності замовника.
- *Працюючий продукт* необхідно *випускати* якомога *частіше*, з періодичністю від двох тижнів до двох-трьох місяців.
- Упродовж усього проекту *розробники і представники бізнесу повинні працювати разом* щодня.
- Над проектом *повинні працювати вмотивовані професіонали*, які працюють у зручних умовах, із повною підтримкою та довірою менеджменту проекту.
- *Особиста комунікація* – найефективніший та найпрактичніший метод обміну інформацією в команді.
- *Працюючий продукт – головний показник прогресу.*
- Інвестори, розробники і користувачі повинні мати можливість підтримувати *постійний ритм роботи*. Agile допомагає налагодити такий сталий процес розроблення.
- Постійна увага до *технічної досконалості і якості проектування* підвищує гнучкість проекту.
- *Простота* – мистецтво мінімізації зайвої роботи – дуже необхідна.
- *Найкращі* вимоги, архітектурні та технічні *рішення виникають у командах, які здатні самоорганізовуватись*.
- Команда *постійно шукає способи підвищення ефективності* та відповідно коригує свою роботу.

Більшість гнучких методологій націлені на мінімізацію ризиків шляхом зведення розроблення до серії коротких циклів – ітерацій, які, як правило, тривають один-два тижні. Кожна ітерація є програмним проектом у мініатюрі і включає всі

завдання, необхідні для отримання мінімального приросту функціональності: планування, аналіз вимог, проектування, кодування, тестування та документування. Гнучкий програмний проект передбачає в кінці кожної ітерації отримання працездатного, готового до встановлення у реальному середовищі продукту. Після закінчення кожної ітерації команда виконує переоцінку пріоритетів розроблення.

Гнучкі методи орієнтовані на різні аспекти життєвого циклу розроблення програмного забезпечення. Деякі акцентують увагу на практичних завданнях (екстремальне програмування, прагматичне програмування, Agile-моделювання), інші зосереджені на управлінні програмними проектами (Scrum). Також серед agile-методів існують підходи, які забезпечують повне охоплення життєвого циклу розроблення – метод розроблення динамічних систем (dynamic systems development method, DSDM) та уніфікований процес розроблення (RUP), розглянутий вище. Більшість гнучких методів для використання упродовж життєвого циклу ПЗ потрібно доповнювати іншими підходами, окрім DSDM та RUP.

Особливістю гнучких методів є те, що на даний час відсутні дані про провальні agile-проекти, але є результати опитувань, які підтвердили їх успішне використання [36]. Тому і потужні компанії-розробники активно їх запроваджують у свою діяльність, наприклад, Oracle запроваджує гнучкі методи управління ЖЦ продуктів (Agile product lifecycle management), фірма IBM є власником продуктів, які підтримують використання методології RUP.

Потрібно виділити *фактори, які можуть негативно вплинути* на використання гнучких методів:

- масштабні зусилля у галузі розвитку (більше 20 розробників);
- розподілена у просторі команда;
- примусове впровадження гнучкого процесу усупереч вимогам команди розробників;

Небажане використання agile-методів у критично важливих системах (табл.6), де відмова ПЗ неможливий ні в якому разі (наприклад, управління повітряним рухом).

Таблиця 6 – Порівняння використання гнучких та формальних методів

Agile-методи	**Методи із чітким планом**	**Формалізовані методи**
Низька критичність	Висока критичність	Життєво важлива критичність
Досвідчені розробники	Молодші розробники	Досвідчені розробники
Вимоги часто змінюються	Вимоги змінюються не часто	Обмежені вимоги
Малі команди розробників	Великі групи розробників	Вимоги обов'язково моделюються
Культура, що залежить від змін	Стала культура розроблення	Екстремальна увага на якості розроблення

Agile акцентує увагу на безпосередньому спілкуванні між учасниками проекту. Більшість agile-команд розміщені в одному офісі (*bullpen*). До команди обов'язково включають представників замовника (замовники, які визначають продукт, менеджери продукту, бізнес-аналітики або користувачі). Також до команди входять тестувальники, дизайнери інтерфейсу, технічні автори та менеджери.

Із agile-методів потрібно виділити ***Scrum***, що встановлює правила керування процесом розроблення та дозволяє використовувати вже існуючі практики кодування, коригуючи вимоги або вносячи тактичні зміни. Використання цієї методології дає можливість виявляти і усувати відхилення від бажаного результату на найбільш ранніх етапах розроблення програмного продукту.

Scrum-методи ітераційні та інкрементні. Кожна ітерація (спринт) триває упродовж 15-30 днів і повинна закінчитися

приростом функціональних можливостей. Під час спринту обов'язково відбуваються зустрічі дійових осіб:

- ***Планування спринту*** *(Planning Meeting)* – відбувається на початку ітерації. Не більше 4-8 годин.
- ***Мітинг*** *(Daily Scrum)* – відбувається кожного дня упродовж спринту. Триває 15 хвилин.
- ***Демонстрація*** *(Demo Meeting)* **–** відбувається в кінці ітерації (спринту). Обмежена 4 годинами.
- ***Ретроспектива*** *(Retrospective Meeting).* Усі члени команди розповідають про своє ставлення до ходу спринту. Обмежено 1-3 годинами.

Дійові особи розподіляються на дві групи:

- *повністю задіяні* у процесі розроблення («свині»):
  - Власник Продукту (Product Owner);
  - Керівник (ScrumMaster);
  - Команда (Scrum Team);
- *причетні* до розроблення («кури»):
  - Користувачі (Users);
  - Клієнти, Продавці (Stakeholders);
  - Експерти-консультанти (Consulting Experts).

Віддаючи перевагу безпосередньому спілкуванню, agile-методи зменшують обсяг письмової документації порівняно з іншими методами, що викликає критику прихильників більш формалізованих підходів.

Патерни проектування

Створюючи об'єкт, у тому числі й програмний продукт, розробник часто стикається із завданнями, які вже хто-небудь вирішив. У 70-ті рр. XX ст архітектор Кристофер Александер (Christopher Alexander) [37] запропонував шаблони проектування будинків та міст. Через десятиліття ця ідея переросла у процесі розроблення ПЗ до шаблонів проектування інтерфейсу користувача, запропонованих Вардом Каннінгемом (Ward Cunningham) та Кентом Беком [38]. Далі ідея була

активно підхоплена та розвинута у вигляді каталогу патернів ООП. Цей каталог [39] став дуже популярним серед розробників і часто згадується як патерни GoF («Gang of Four», або «банда чотирьох», – за кількістю авторів). Ідея повторного використання не тільки коду, а й архітектурних та проектних рішень виявилася настільки успішною, що сьогодні патерни проектування широко застосовуються в різних методиках розроблення програмного забезпечення.

У роботі [39] під ***патернами проектування*** об'єктно-орієнтованих систем розуміють опис взаємодії об'єктів і класів, адаптованих для вирішення спільної задачі проектування в конкретному контексті.

Існує багато патернів розроблення програмних систем, які відмінні сферою застосуання, масштабом, змістом, стилем опису. Наприклад, залежно від області використання існують такі патерни, як патерни аналізу, проектування, кодування, рефакторингу, тестування, реалізації корпоративних застосувань, шаблони роботи з базами даних, шаблони розподілених додатків, шаблони роботи із багатопотоковістю, шаблони документування та ін.

В останнє десятиліття шаблони впроваджені навіть у роботу менеджера процесу розроблення ПЗ (Джеймса Коплі, Ніла Харрісона «Organizational Patterns of Agile Software Development», Тома Демарко, Тіма Лістера «Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior»).

У даний час найбільш популярними патернами є патерни проектування. Однією з найпоширеніших класифікацій таких патернів є класифікація за ступенем деталізації та рівнем абстракції розглянутих систем. Патерни проектування програмних систем поділяються на три категорії [40] (рис.44).

Архітектурні патерни, найбільш високорівневі, описують структурну схему програмної системи в цілому. У даній схемі зазначаються окремі функціональні складові системи (підсистеми), а також взаємовідносини між ними. Прикладом архітектурного патерну є добре відома програмна

парадигма "модель-вигляд-контролер" (model-view-controller - MVC).

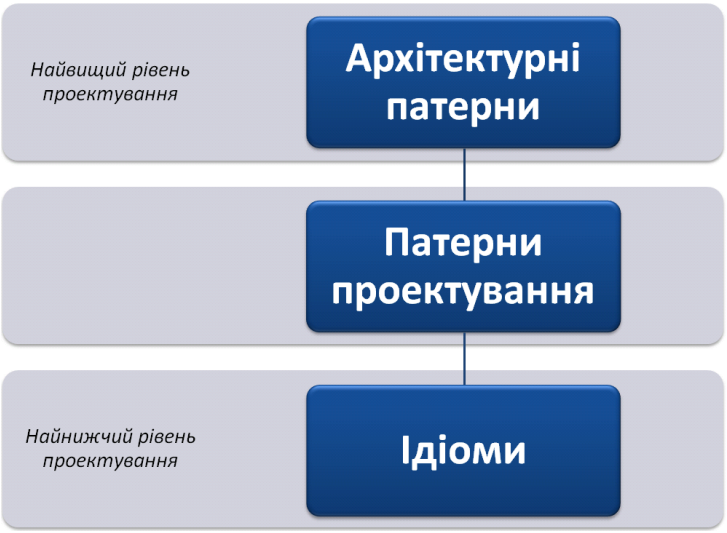


Рисунок 44 – Шаблони проектування програмних систем

Ідіоми, низькорівневі патерни, мають справу з питаннями реалізації певної проблеми з урахуванням особливостей мови програмування. При цьому часто одні й ті самі ідіоми для різних мов програмування мають різний вигляд або взагалі відсутні. Наприклад, в C++ для усунення можливих втрат пам'яті можуть використовуватися інтелектуальні покажчики. Інтелектуальний покажчик містить покажчик на ділянку динамічно виділеної пам'яті, що буде автоматично звільнений при виході із зони видимості. У середовищі Java такої проблеми просто не існує, оскільки там використовується автоматичне складання сміття. Як правило, для використання ідіом потрібно глибоко знати особливості застосовуваної мови програмування.

Завдання кожного патерну – дати чіткий опис проблеми та її вирішення у відповідній області. У загальному випадку опис патерну завжди містить такі елементи [38]:

- *Назва патерну* – унікальне змістове ім'я, що однозначно визначає дану задачу або проблему і її рішення.

- *Розв'язувана задача* – надається розуміння того, чому розв'язувана проблема дійсно є такою, чітко описує її межі.
- *Рішення* – зазначається, як саме дане рішення пов'язане з проблемою, наводяться шляхи її вирішення.
- *Результати використання патерну* – як правило, це переваги, недоліки та компроміси.

На рис.45 показана класифікація шаблонів проектування інформаційних систем [41].

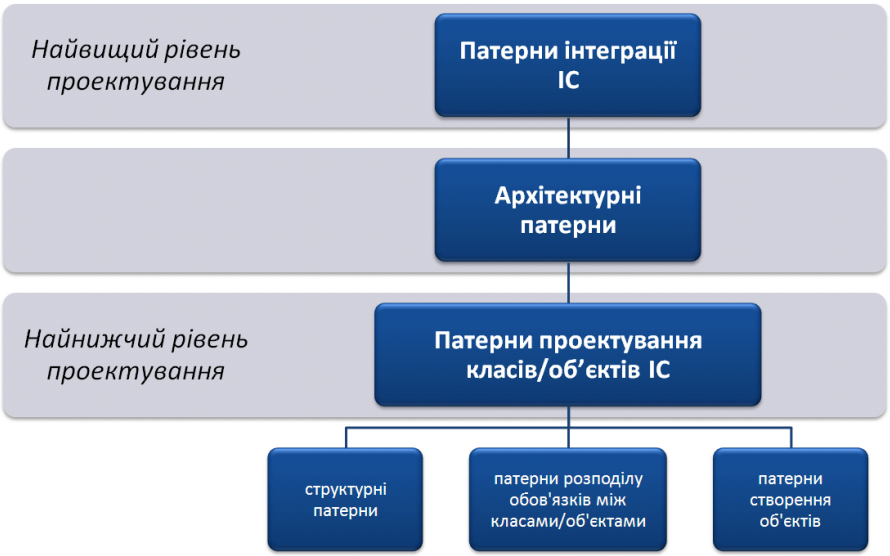


Рисунок 45 – Шаблони проектування інформаційних систем

Архітектурні патерни також об’єднуються у групи:

- структурні, архітектурні патерни – слугують для організації класів або об’єктів системи у базовій підструктурі;
- патерни управління – для забезпечення необхідного системного функціоналу.

У свою чергу, патерни управління розділені на патерни централізованого управління (патерни, в яких одна з підсистем повністю відповідає за управління, запускає і завершує роботу решти підсистем), патерни управління, які передбачають децентралізоване реагування на події (згідно з цим патернам на

зовнішні події відповідає відповідна підсистема), та патерни, які описують організацію зв'язку з базою даних

Патерни інтеграції інформаційних систем знаходяться на верхньому рівні класифікації патернів проектування. Аналогічно патернам більш низьких рівнів класифікації серед патернів інтеграції виділена група структурних патернів. Структурні патерни описують основні компоненти інтегрованої метасистеми. Для опису взаємодії окремих корпоративних систем, включених до інтегрованої метасистеми, використовується група патернів, об'єднаних відповідно до методу інтеграції. Інтеграція корпоративних інформаційних систем передбачає організований обмін даними між системами, для якого використовується відповідний патерн. Необхідно зазначити, що на відміну від патернів проектування класів/об’єктів і архітектурних системних патернів віднесення окремого патерну інтеграції до того чи іншого виду є менш умовним.

Про шаблони проектування знають розробники, проектувальники, архітектори, але про це явище абсолютно нічого не відомо користувачам,для яких і розроблялася мова шаблонів Александера [37]. На сьогоднішній день основна роль шаблонів - це повторне використання досвіду в різних областях розроблення ПЗ, усунення комунікаційного бар'єру всередині команди розробників і між ними, підвищення якості створюваного продукту за рахунок використання перевірених роками рішень.

Питання для самоконтролю

1. Перелічіть візуальні нотації для моделювання предметної області та визначте області їх застосування.
2. Опишіть основні класифікатори мови UML.
3. Які діаграми використовуються в мові UML? Для чого вони використовуються?
4. Які методології розроблення вам відомі? Наведіть приклади.
5. Перелічіть переваги гнучкого розроблення ПЗ та зазначте її недоліки.
6. Перелічіть та поясніть основні характеристики уніфікованого процесу розроблення ПЗ.
7. Які моделі програмної системи розробляються в уніфікованому процесі RUP? Порівняйте їх.
8. Визначте особливості методології Microsoft Solution Framework.
9. Які цінності та принципи покладені в основу методів eXtreme Programming?
10. Що є особливостями методів Scrum?
11. Порівняйте методології RUP, MSF та XP.

СПИСОК ВИКОРИСТАНОЇ ТА РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Computing Curricula 2005. The Overview Report. [Електронний ресурс]. – Режим доступу: www.acm.org/education/curricula.html.
2. Обеспечение систем обработки информации программное. Термины и определения. ГОСТ 19781-90. – [Чинний від 1992-02-01 до 2007-12-10] – 16 с.– (Міждержавний стандарт).
3. Systems and software engineering – Software Life Cycle Processes. ISO 12207:2008. – [Чинний від 2008-02-01] – II, 122 с.– (Міжнародний стандарт).
4. Бьярне Страуструп. Программирование. Принципы и практика использования C++; пер. с англ. Д.Клюшин. Москва: Вильям,. 2011. – 1248 с.
5. Alistair Cockburn. Methodology per project. Humans and Technology Technical Report, TR 99.04, Oct.1999 7691 Dell Rd, Salt Lake City, UT 84121 USA. [Електронний ресурс]. Режим доступу: http://alistair.cockburn.us/Methodology+per+project.
6. IEEE Standard Glossary of Software Engineering Terminology, Глосарій. IEEE Std 610.12-1990. – (Галузевий стандарт).
7. Автоматизированные системы. Стадии создания. ГОСТ 34.601-90. – [Чинний від 1991-01-01] – 10 с.– (Міждержавний стандарт).
8. Техническое задание на создание автоматизированной системы. ГОСТ 34.602-89 – [Чинний від 1990-01-01] – 12 с.– (Міждержавний стандарт).
9. Виды, комплектность и обозначение документов при создании автоматизированных систем. ГОСТ 34.201-89. – [Чинний від 1990-01-01] – 8 с.– (Міждержавний стандарт).
10. Модели жизненного цикла программного обеспечения. [Електронний ресурс]. Режим доступу: http://swebok.sorlik.ru/software_lifecycle_models.html.

11. Iterative and incremental development. [Електронний ресурс]. Режим доступу: http://en.wikipedia.org/wiki/ Iterative_and_incremental_development.
12. Spiral model. [Електронний ресурс]. Режим доступу: http://en.wikipedia.org/wiki/Spiral_model.
13. Standard Glossary of terms used in Software Testing. Version 1.2, ISTQB, 2006. [Електронний ресурс]. – Режим доступу: www.istqb.org/downloads/glossary.
14. 1061-1998 IEEE Standard for Software Quality Metrics Methodology. – (Галузевий стандарт).
15. Проектный треугольник. [Електронний ресурс]. Режим доступу: http://office.microsoft.com/ru-ru/project-help/HA010351692.aspx.
16. Терехов А.М. Вступ до технологій програмування. [Електронний ресурс]. Режим доступу: http://www.intuit.ru/department/se/introprogteach/2.
17. Oracle® Unified Method (OUM). [Електронний ресурс]. Режим доступу: www.oracle.com/us/products/ consulting/resource-library/oracle-unified-method-069204.pdf.
18. Спиральная модель. [Електронний ресурс]. Режим доступу: http://ru.wikipedia.org/wiki Спиральная_модель.
19. Кент Бек. Экстремальное программирование. – СПб.: Изд-во «Питер», 2002. – 224 с.
20. Брукс Ф. Как проектируются и создаются программные комплексы. Мифический человеко-месяц. Очерки по системному программированию. – СПб.: Изд-во «Символ-Плюс», 2000. – 304 с.
21. ДСТУ ISO 9000:2007. Системи управління якістю. Основні положення та словник термінів. – К.: Держспоживстандарт, 2008. – [Чинний від 2008-01-01] – 35 с.– (Державний стандарт).
22. Alistair Cockburn. Methodology per project. Humans and Technology Technical Report, TR 00.04, Jan.00. [Електронний ресурс]. Режим доступу: http://alistair.cockburn.us.
23. Alistair Cockburn, Laurie Williams. The Costs and Benefits of Pair Programming. Proceedings of the First International

Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000). [Електронний ресурс]. Режим доступу: http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF.
24. Сэм Канер. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бізнес-приложений/ Сэм Канер, Джек Фолк, Енг Кек Нгуен; пер. с англ. – К.: Изд-во «ДиаСофт», 2001. – 544 с.
25. IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK), 2004. – (Галузевий стандарт). [Електронний ресурс]. – Режим доступу: http://www.computer.org/portal/web/swebok/htmlformat.
26. ISO/IEC 15288 Systems and software engineering - System life cycle processes. – [Чинний від 2008-03-18] – 70 с.– (Міжнародний стандарт).
27. ДСТУ ISO 9001:2009. Системи управління якістю. Вимоги. – К.: Держспоживстандарт, 2009. – [Чинний від 2009-06-22] – 80 с.– (Державний стандарт).
28. M.C. Paulk, C.V. Weber, B. Curtis, M.B. Chrissis et al The Capability Maturity Model: Guidelines for Improving the Software Process. Addison-Wesley, Boston. 1995. – 456 с.
29. OMG Unified Modeling Language Specification Version 1.5, March 2003 formal/03-03-01. [Електронний ресурс]. Режим доступу: www.omg.org.
30. Буч Г., Рамбо Дж., Джекобсон А. UML. Руководство пользователя. – СПб.: Изд-во «ДМК-Пресс», «Питер», 2001. – 432 с.
31. Patricia Griffith Friel and Thomas M. Blinn (1989). "Automated IDEF3 and IDEF4 Systems Design Specification Document". Technical report. NASA Johnson Space Center. [Електронний ресурс]. Режим доступу: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19910023489_1991023489.pdf.
32. Шеер А.-В. ARIS - моделирование бизнес-процессов. – Москва: Вильямс, 2009. – 224 с.

33. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – СПб.: Изд-во «Питер», 2002. – 492 с.
34. Introduction to the Microsoft Solutions Framework. [Електронний ресурс]. Режим доступу: http://technet.microsoft.com/en-us/library/bb497060.aspx.
35. Agile manifesto. [Електронний ресурс]. Режим доступу: http://agilemanifesto.org.
36. Agile software development. [Електронний ресурс]. Режим доступу: http://en.wikipedia.org/wiki/ Agile_software_development.
37. C. Alexander et al. A pattern language: towns, buildings, construction. New York: Oxford University Press, 1977. – 120 с.
38. Введение в паттерны проектирования. [Електронний ресурс]. Режим доступу: http://cpp-reference.ru/patterns/ introduction.
39. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Изд-во «Питер», 2007. – 366 с.
40. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. Wiley, 1996. – 476 с.
41. Ольга Дубинина. Обзор паттернов проектирования. [Електронний ресурс]. Режим доступу: http://citforum.ru/SE/project/pattern/p_4.shtml#lit_2.