МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ І ВЕРИФІКАЦІЇ ПРОГРАМ

КОНСПЕКТ ЛЕКЦІЙ

Рекомендовано Методичною радою ФІОТ КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 121 «Інженерія програмного забезпечення»,
спеціалізацією «Інженерія програмного забезпечення комп'ютерних систем»

Київ
КПІ ім. Ігоря Сікорського
2021

Сучасні технології автоматизованого проектування і верифікації програм: Конспект лекцій [Електронний ресурс] : навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення», спеціалізації «Інженерія програмного забезпечення комп'ютерних систем» / КПІ ім. Ігоря Сікорського; уклад.: Я. Ю. Дорогий, О. О. Дорога-Іванюк. – Електронні текстові дані (1 файл: 3,9 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 89 с.

Гриф надано Методичною радою ФІОТ НТУУ «КПІ ім. Ігоря Сікорського» (протокол № 1 від 30.08.2021 р.)

Електронне мережне навчальне видання

СУЧАСНІ ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ І ВЕРИФІКАЦІЇ ПРОГРАМ

КОНСПЕКТ ЛЕКЦІЙ

Укладачі: *Дорогий Ярослав Юрійович*, канд. техн. наук, доц.
Дорога-Іванюк Олена Олександрівна

Відповідальний редактор *Полторак В. П.*, канд. техн. наук, доц.

Рецензент *Стиренко С. Г.*, д-р. техн. наук, проф. зав. каф. ОТ

Конспект лекцій з навчальної дисципліни «Сучасні технології автоматизованого проектування і верифікації програм» у компактній формі відображає матеріал курсу, допомагає сформувати загальне уявлення про предмет вивчення, правильно зорієнтуватися в даній галузі знань. Конспект лекцій з названої дисципліни сприятиме більш успішному вивченню дисципліни, причому більшою мірою для студентів очно-заочної форми, екстернату та дистанційного навчання.

ЗМІСТ

Лекція 1. Сучасні технології автоматизованого проектування і верифікації програм (СТАПВП) як інженерна дисципліна

1.1 Загальна характеристика дисципліни

СТАПВП як інженерна дисципліна – це сукупність прийомів виконання діяльності, пов'язаної з автоматизованим проектуванням, виготовленням та верифікацією програмного продукту для різних видів цільових об'єктів із автоматизованих застосуванням методів, засобів і інструментів. Основу інженерії становлять наступні базові елементи процесу виготовлення програмного продукту:

1) ядро знань SWEBOK, як набір теоретичних концепцій і формальних визначень методів і засобів розробки та керування програмними проектами, які можуть застосовуватися в СТАПВП;

2) базовий процес СТАПВП, як стрижень процесної діяльності в організації-розробнику програмного продукту;

3) стандарти, як набір регламентованих правил конструювання проміжних артефактів у процесах ЖЦ;

4) інфраструктура – умови середовища та методичне забезпечення базового процесу створення програмної системи (ПС) і підтримка дій його виконавців, що займаються виробництвом програмного продукту;

5) менеджмент проекту (PMBOK) – ядро знань з керування промисловими проектами – набір стандартних процесів, а також принципів і методів планування і контролювання роботами у проекті;

6) засоби та інструменти проектування, розробки та тестування програмних продуктів.

З інженерної точки зору в СТАПВП розв'язуються задачі автоматизованого виготовлення ПП, подані як технологічні процеси формування вимог, проектування і супроводу продукту, а також перевірки операцій базового процесу щодо правильності виконання різних функціональних задач та виконання робіт за проектом у заданий замовником строк.

Дисципліну будемо розглядати з двох точок зору:

– як інженерну діяльність, у якій інженери різних категорій виконують роботи в рамках проекту, використовуючи відповідні теоретичні методи і засоби ПІ, що рекомендовані у ядрі знань SWEBOK, а також стандарти процесів проектування цільових об'єктів за вибраними методами;

– як систему керування проектом, якістю і ризиками з використанням правилі положень стандартів ЖЦ, якості та менеджменту проекту.

Інженерна діяльність обов'язково планується та ґрунтується на розподілі робіт у проекті між різними категоріями виконавців. Менеджер проекту – це головна діюча особа проекту, відповідальна за проектування і контроль виконання робіт спеціальними службами інфраструктури проекту в організації, зокрема служби верифікації, тестування, якості тощо. Продукт колективного виготовлення передається замовнику для супроводу. В ньому можуть бути

виявлені різні помилки і недоліки, які усувають розробники. Ця діяльність у програмній інженерії практично вже відпрацьована і за своєю суттю близька до інженерної діяльності у промисловості, де *інженерія* – це спосіб застосування наукових результатів для виготовлення технічних виробів на основі технологічних правил і процедур, методик виміру, оцінки і сертифікації з метою отримання користі від виготовленого продукту або товару.

Далі (у пп. 1–5) наведено загальну характеристику базових елементів інженерної дисципліни СТАПВП.

1. Ядро знань SWEBOK – стислий опис концептуальних основ програмної інженерії. Структурно ділиться на 10 розділів (knowledge areas), які умовно можна розкласти за двома категоріями: проектування продукту і інженерна діяльність. Перша категорія – це методи і засоби розробки (формування вимог, проектування, конструювання, тестування, супровід), друга категорія – методи керування проектом, конфігурацією і якістю та базовим процесом організації-розробника(детальніше див. у п.1.2). Методи ядра знань програмної інженерії менеджер проекту зіставляє з відповідними стандартними процесами ЖЦ, виконання яких забезпечує послідовне розроблення програмного продукту через наповнення базового процесу програмної інженерії методами з ядра знань SWEBOK, а також задачами і діями стандартного ЖЦ, що обумовлює його застосовність до потреб конкретної організації-розробника щодо певної регламентованої послідовності розробки і супроводу програмного продукту. Все це створює технологічний базис інженерії виготовлення конкретного продукту (або низки однотипних продуктів) в організації. На початкових стадіях розробки виконуються процеси визначення вимог до продукту, вироблення проектних рішень і каркасу (абстрактної архітектури) майбутнього продукту. На основі вимог і каркасу розробляються або вибираються готові прості об’єкти для «наповнення» цього каркасу змістом для подальшого його доведення до стану готового продукту.

2. Базовий процес (БП) – є метарівнем для забезпечення «процесного продукування» продукту. Він містить у собі опис понять щодо оснастки організаційної структури колективу розробників та методології оцінки, вимірювання, керування змінами та вдосконалення самого процесу. В цілому базовий процес складається з множини логічно пов’язаних видів інженерної діяльності організації-розробника та набору засобів і інструментів щодо виготовлення програмного продукту.

3. Інфраструктура – це набір технічних, технологічних, програмних (методичних) та людських ресурсів організації-розробника, необхідних для виконання підпроцесів базового процесу програмної інженерії, орієнтованого на виконання договору з замовником програмного проекту. До технічних ресурсів належать: комп’ютери, пристрої (принтери, сканери тощо), сервери і т.п., до програмних – загальносистемне ПЗ середовища розробки, напрацювання колективу, оформлені у вигляді компонентів повторного використання, та інформаційне забезпечення. Технологічні та методичні ресурси складають методики, процедури, правила, рекомендації стандартів з процесу і керування

персоналом разом з комплектом документів, що встановлює регламент виконання і регулювання процесів ЖЦ, застосовуваних для розв'язання конкретних задач проекту. Людські ресурси – це групи розробників і служб керування проектом, планами, якістю, ризиком, конфігурацією, а також перевірки правильності виконання проекту розробниками.

Засоби, проміжні результати розробки за процесами ЖЦ, а також методики керування різними ресурсами, виконання БП і застосування методів програмування, зберігаються у базі знань проекту (рисунок 1.1).

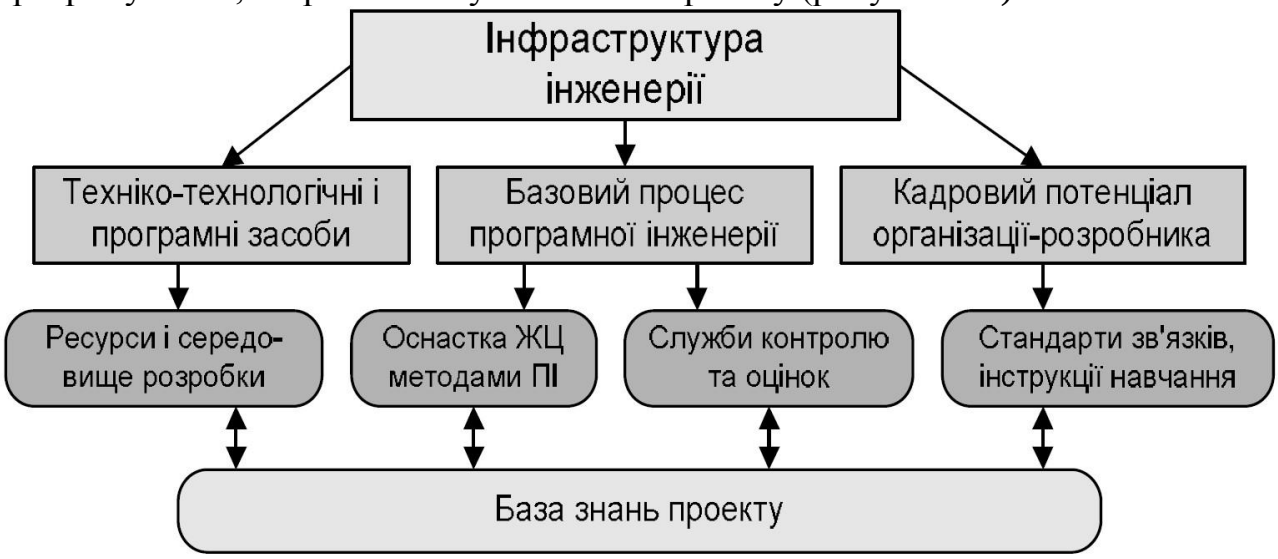

Рисунок 1.1 – Загальна інфраструктура проекту

Після виконання проекту і отримання досвіду побудови конкретного продукту, базовий процес і його окремі елементи, подані на рисунку 1.1, можуть удосконалюватися (доопрацюванням або зміною прийомів, доробкою, змінюванням, додаванням нових засобів) відповідно до вимог стандарту ДСТУISO/IEC 15504-7 («Оцінювання процесів ЖЦ ПЗ. Настанови з удосконалення процесу») з метою підвищення рівня можливостей і оцінки потужності процесу.

Готовність всіх видів забезпечення організації-розробника, досконаліть виконуваних процесів і якість створеного в ній продукту надають підстави для оцінки зрілості організації або сертифікації процесів виробництва ПЗ. Для оцінювання зрілості може застосовуватися модель зрілості CMM (Capability22 Розділ 1Maturity Models), запропонована Інститутом програмної інженерії SEI США, або інша модель, наприклад, Bootstrap, Trillium тощо. Модель CMM встановлює рівні зрілості організації щодо створення програмних продуктів. Рівень зрілості визначається наявністю в організації базового процесу усіх необхідних видів ресурсів (у тому числі і фінансових), відповідних стандартів і методик, а також професійних здібностей (зрілості) членів колективу організації, здатних виготовляти програмні продукти в заданий строк і встановленої вартості.

4. Стандарти ПІ встановлюють технологічно відпрацьований набір процесів зі строго визначеним і регламентованим порядком проведення різних

видів робіт з програмної інженерії, зв'язаних з розробленням програмного продукту і оцінюванням його якості, ризику тощо. Стандарти у галузі програмної інженерії регламентують різні напрями діяльності щодо проектування програмних продуктів. Вони стандартизують термінологію і поняття, життєвий цикл, якість, вимірювання, оцінювання продуктів і процесів. Найбільш важливими серед них є стандарт ISO/IEC 12207 «Процеси життєвого циклу програмного забезпечення» (та його дещо застарілий вітчизняний еквівалент ДСТУ 3918–99), серія стандартів ДСТУISO/IEC 14598 «Оцінювання програмного продукту», стандарт ДСТУ ISO 15939«Процес вимірювання», серія стандартів ISO/IEC 15504 «Оцінювання процесів ЖЦПЗ», базові стандарти з якості – ISO 9001 «Системи керування якістю. Вимоги», ДСТУ 2844–94, ДСТУ 2850–94, що регламентують різні аспекти забезпечення якості ПП. Серед стандартів, що безпосередньо пов'язані з якістю ПЗ, слід також назвати проект нової серії стандартів ISO/IEC TR 9126 «Програмна інженерія. Якість продукту». У цих стандартах узагальнені знання спеціалістів з технології проектування і інженерних методів керування розробкою, починаючи від встановлення вимог і закінчуючи оцінкою якості продукту і можливою його подальшою сертифікацією.

Процеси ЖЦ в стандарті ISO/IEC 12207 подають загальні положення, задачі і регламентовані дії з проектування, а також рекомендації щодо застосування цих процесів для розроблення і контролю проміжних результатів. У стандарті містяться також описи організаційних процесів – планування, керування і супроводження. *Процес планування* призначений для складання планів, графіків робіт з виконання проекту і розподілу робіт між різними категоріями фахівців, а також для контролю планів і виконаних робіт. *Процес керування проектом* визначає задачі та дії з керування роботами фахівців проекту, які володіють теорією керування, а також відстеження планових строків, що встановлені замовником проекту. *Процес супроводження* містить у собі дії з виявлення й усунення знайдених недоліків і внесення нових або видалення деяких функцій у продукті.

Ядро знань SWEBOK і стандарти з ЖЦ мають взаємопов'язані складові. Процесам ЖЦ зіставляються необхідні методи ядра і тим самим визначається базовий процес створення проекту, що доповнюється методиками і обмеженнями щодо вироблення продукту. Діючі фундаментальні моделі ЖЦ (каскадна, спіральна тощо), що широко використовуються на практиці, пропонують привнесення в них стилю проектування і реалізації деяких видів продуктів.

5. Менеджмент проекту – це керування виконанням проекту з використанням теорії керування та процесів ядра знань PMBOK (Project Management body of knowledge). У серії настанов до PMBOK, розроблених американським Інститутом керування проектами (www.pmi.org), подано положення і правила керування часовим виробничим циклом побудови унікального продукту в рамках проекту, спочатку без урахування рівня комп'ютеризації промисловості(1987р.), а потом і з його врахуванням (2000р.).

Ядро знань PMBOK містить у собі опис лексики, структури процесів і областей знань, відображаючи сучасну практику керування проектами в різних областях промисловості. У ньому визначені процеси ЖЦ проекту і головні області знань, згруповані за задачами: ініціація, планування, використання, моніторинг і керування, завершення. Крім того, область знань «інтеграція» визначає прийняття рішень про використання ресурсів у кожний момент виконання проекту і керування загальними задачами проекту.

У PMBOK визначено три головні області знань. Область знань *керування змістом проекту* містить у собі процеси, які необхідні для виконання робіт за проектом, а також для його планування з розподілом робіт на простіші для спрощення процесу керування. Область *керування якістю* містить у собі процеси й операції досягнення цілей проекту щодо якості, правила і процедури для полегшення процесу досягнення цілей і забезпечення якості відповідно до заданих вимог, а також контролю результату на відповідність стандартам якості. Область *керування людськими ресурсами* організації і розподілу робіт між виконавцями відповідно до їх кваліфікації і професіоналізму містить у собі процедури регламентування виконання робіт з розроблення програмного продукту. Сфера менеджменту проекту охоплює виконавців, усі види забезпечення (інформаційне, програмне, технічне тощо), і, що головне, роботи, розподілені між виконавцями. Кожній роботі відповідає завдання і вхідні дані, які задаються менеджером проекту для виконання робіт.

На теперішній час настанови до PMBOK та SWEBOK введені в статус стандартів, а саме: ISO/IEC TR 19759 («Guide to the Software Engineering Body ofKnowledge (SWEBOK)) та IEEE Std.1490 «IEEE Guide adoption of PMI Standard. AGuide to the Project Management Body of Knowledge).

6. Засоби та інструменти ПІ. Проектування об’єктів виконується за допомогою сучасних візуальних мов, наприклад UML, мов програмування (C++,Java, Object Pascal тощо) з використанням відповідних інструментальних середовищ, що містять у собі необхідні мовні перетворювачі і інструменти підтримки різних артефактів ПП, що розробляються. Як засоби їх проектування застосовують діаграми використання, потоків даних, класів, поведінки, а також шаблони, каркаси тощо.

Перевірка правильності цих об’єктів здійснюється за допомогою вказаних методів і відповідних інструментів, пристосованих для розроблення різних задач проекту у середовищі проектування. Готовий продукт перевіряється щодо відповідності реалізованих функцій заданим вимогам, тестується за спеціальними методиками, а також піддається вимірюванню та оцінюванню щодо отримання показників якості, точності, відмовостійкості, захищеності тощо.

У середовищі проектування цільових об’єктів застосовуються сучасні технології і відповідні інструментально-технологічні пакети інструментів (наприклад, технології RUP, MSF та інструменти Rational Rose, Microsoft VisualStudio тощо). Вони містять не тільки інструменти проектування різних

типів цільових об’єктів проектів, а й засоби і інструменти керування проектом, зокрема персоналом, планами та якістю продуктів.

Засоби і інструменти забезпечують автоматизовану підтримку базового процесу виготовлення програмного продукту в організації-розробнику.

Класифікацію загальних інструментів, рекомендованих для застосування до всіх видів об’єктів у процесах ЖЦ, подано в ядрі знань SWEBOK.

Лекція 2. Характеристика областей знань з інженерії програмного забезпечення – SWEBOK

У лекції розглядається теоретичний і інтелектуальний базис проектування – методи, принципи, засоби і методології, представлені областями в ядрі знань програмної інженерії SWEBOK.

Ядро знань SWEBOK – основний науково-технічний документ, що відображає знання та досвід багатьох іноземних і вітчизняних фахівців з програмної інженерії і узгоджується з регламентованими процесами ЖЦ стандарту ISO/IEC 12207.

Документ містить у собі опис 10 областей, кожна з яких представлена відповідно до прийнятої всіма учасниками формування ядра SWEBOK загальної схеми опису, що містить у собі визначення понятійного апарату, методів і засобів, а також інструментів підтримки інженерної діяльності. Стосовно кожної області визначено коло знань, які повинні практично використовуватися при виконанні процесів життєвого циклу.

Для подання понятійного апарату областей знань SWEBOK проведемо умовне розділення областей на *головні* (п'ять областей для розроблення ПС, рисунок 2.1) і допоміжні *організаційні області* (п'ять областей, що забезпечують інженерію керування розробкою ПС, рисунок 2.2).

У кожній області наведені ключові поняття, підходи і методи проектування різних типів ПС. Розподіл областей на основні і допоміжні відповідає структурі розподілу процесів стандарту ISO/IEC 12207 (див. розділ 2), виконання яких визначається методами і засобами, запропонованими в ядрі знань SWEBOK.

Далі подається огляд областей цього ядра, що є основним для дисципліни СТАПВП.

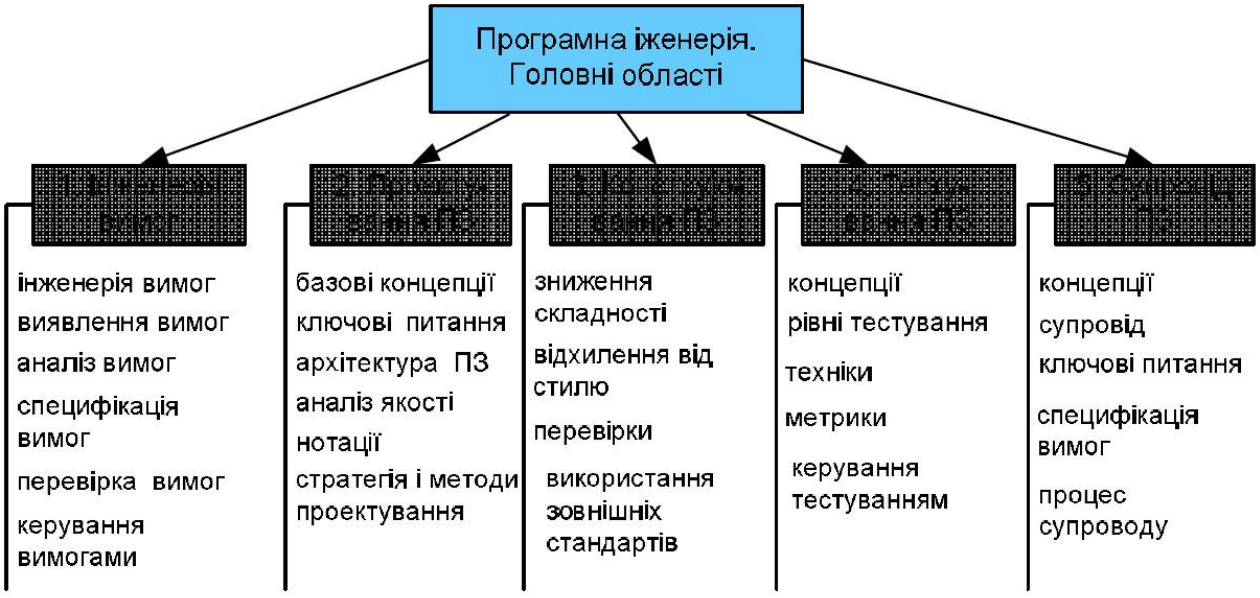

Рисунок 2.1 – Головні області знань SWEBOK

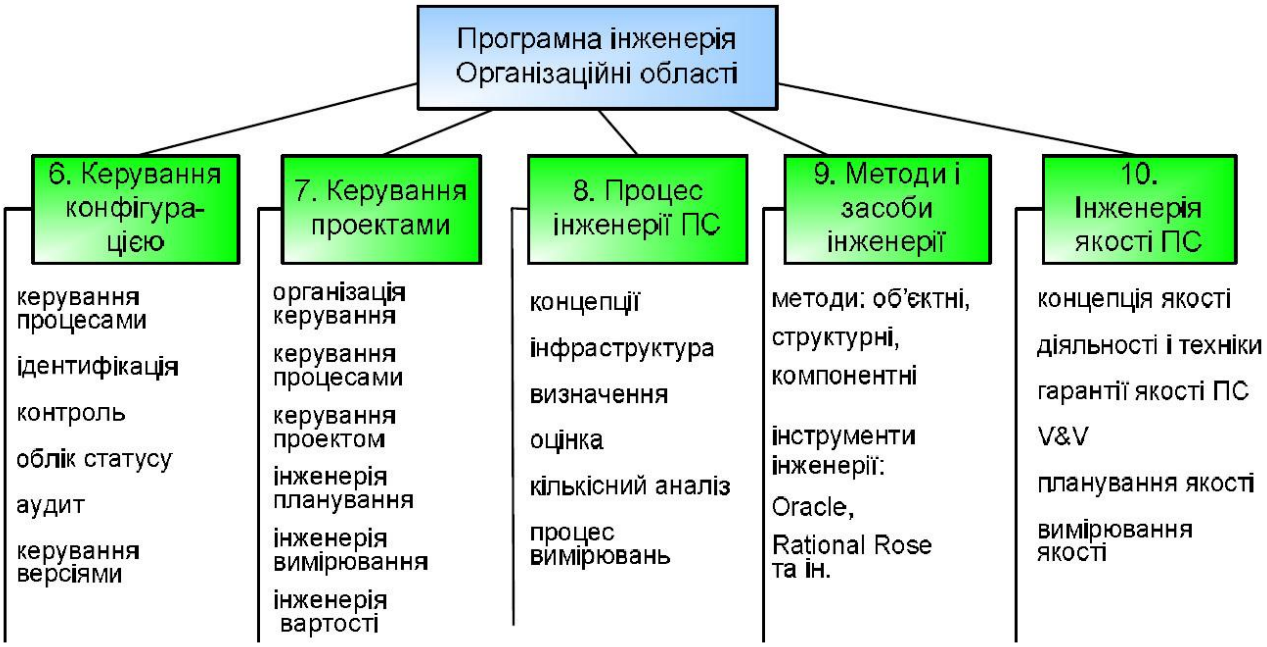

Рисунок 2.2 – Організаційні області знань SWEBOK

2.1. Інженерія вимог

Вимоги до ПЗ – сукупність властивостей, які повинно мати ПЗ. Призначені для адекватного визначення функцій, умов і обмежень виконання ПЗ, а також обсягів даних, технічного забезпечення і середовища його виконання.

Вимоги відбивають потреби людей (замовників, користувачів, розробників), зацікавлених у створенні ПЗ. Замовник і розробник спільно виявляють вимоги, аналізують, переглядають, визначають необхідні обмеження і умови, а також описують їх. Розрізняють вимоги до продукту і до процесу, а також функціональні, не функціональні і системні вимоги. Вимоги до продукту і до процесу визначають умови виконання і режими роботи ПЗ в операційному середовищі, обмеження на структуру і пам'ять комп'ютерів та принципи взаємодії програм.

Функціональні вимоги визначають призначення і функції системи, а нефункціональні – умови стосовно виконання ПЗ, його переносності і доступу доданих. Системні вимоги описують вимоги до програмної системи, яка складається з взаємозалежних програмних і апаратних підсистем і різних застосувань. Вимоги можуть бути кількісні (наприклад, кількість оброблених запитів на секунду, середній показник помилок і т.п.). Значна частина вимог стосується атрибутів якості: безвідмовність, надійність і ін., а також захисту і безпеки як ПЗ, так і даних.

Область знань «Вимоги до ПЗ (Software Requirements)» складається з таких розділів:

– інженерія вимог (Requirement Engineering),
– виявлення вимог (Requirement Elicitation),
– аналіз вимог (Requirement Analysis),
– специфікація вимог (Requirement Specification).

– валідація вимог (Requirement validation),

– керування вимогами (Requirement Management).

Інженерія вимог до ПЗ – це дисципліна аналізу і документування вимог до ПЗ, що полягає в перетворенні запропонованих замовником вимог до системи на опис вимог до ПЗ і їх валідації. Інженерія базується на моделі процесу визначення вимог і діяльності осіб, що забезпечують керування і формування вимог, а також на методах досягнення показників якості.

Модель процесу визначення вимог – це схема процесів ЖЦ, що виконуються від початку проекту і доти, поки не будуть визначені і погоджені вимоги. Таким процесом може бути маркетинг і перевірка виконання вимог у даному проекті.

Керування вимогами до ПЗ полягає в контролі за виконанням вимог і плануванні використання ресурсів (людських, програмних, технічних, часових, вартісних) у процесі розроблення проміжних робочих продуктів на процесах ЖЦ і продукту в цілому.

Якість і процес поліпшення вимог – це процес формулювання характеристик і атрибутів якості (надійність, реактивність і ін.), які повинно мати ПЗ, методи їх досягнення на процесах ЖЦ і оцінювання отриманих результатів.

Виявлення вимог – це процес витягування інформації з різних джерел (договорів, матеріалів аналітиків з декомпозиції задач і функцій системи й ін.), проведення технічних заходів (співбесід, збирання пропозицій і ін.) для формування окремих вимог до продукту і до процесу розроблення. Вимоги погоджуються з замовником.

Аналіз вимог – процес вивчення потреб і цілей користувачів, класифікація і перетворення їх на вимоги до системи, апаратури і ПЗ, встановлення і вирішення конфліктів між вимогами, визначення пріоритетів, меж системи і принципів взаємодії із середовищем функціонування.

Специфікація вимог до ПЗ – процес формалізованого опису функціональних і нефункціональних вимог, вимог до характеристик якості відповідно до стандарту якості ISO/IEC 9126, які будуть відпрацьовуватися на процесах ЖЦ ПЗ. У специфікації вимог відбивається структура ПЗ, вимоги до функцій, якості і документації, а також задається архітектура системи і ПЗ, алгоритми, логіка керування і структура даних. Специфікуються також системні вимоги, нефункціональні вимоги і вимоги до взаємодії з іншими компонентами і платформами (БД, СКБД, маршаллінг даних, мережа й ін.).

Валідація вимог – це перевірка викладених у специфікації вимог, що виконується для того, щоб шляхом відстеження джерел вимог переконатися, що вони визначають саме дану систему. Замовник і розробник ПЗ проводять експертизу сформованого варіанта вимог для того, щоб розробник міг далі продовжувати проектування ПЗ. Один з методів валідації – прототипування, тобто швидке відпрацьовування окремих вимог на конкретному інструменті і дослідження масштабів зміни вимог, вимірювання обсягу функціональності і вартості, а також створення моделей оцінки зрілості вимог.

Верифікація вимог – це процес перевірки правильності специфікацій вимог щодо їх відповідності потребам, несуперечності, повноти і можливості реалізації, а також узгодженості зі стандартами. Як результат перевірки вимог складається погоджений вихідний документ, що встановлює повноту і коректність вимог до ПЗ, а також можливість продовження його проектування.

Керування вимогами – це керування процесами формування вимог на всіх процесах ЖЦ, а також змінами й атрибутами вимог, проведення моніторингу – відновлення джерела вимог. Керування змінами виникає після того, як ПЗ починає працювати в заданому середовищі і виявляє помилки щодо трактування вимог, невиконання деякої окремої вимоги тощо. Невід'ємною складовою процесу керування є *трасування вимог* для відстеження правильності встановлення і реалізації вимог до системи і ПЗ на процесах ЖЦ, а також зворотний процес відстеження в отриманому продукті реалізованих вимог. Для уточнення деяких вимог або додавання нової вимоги складається план зміни вимог, що погоджується з замовником. Внесені зміни спричиняють і зміни в створеному продукті або вокремих його компонентах.

2.2. Проектування програмного забезпечення

Проектування ПЗ – це процес визначення архітектури, набору компонентів, їх інтерфейсів, інших характеристик системи і кінцевого складу програмного продукту.

Область знань «Проектування ПЗ (Software Design)» складається з таких розділів:

– базові концепції проектування ПЗ (Software Design Basic Concepts),– ключові питання проектування ПЗ (Key Issue in Software Design),

– структура й архітектура ПЗ (Software Structure and Architecture),

– аналіз і оцінка якості проектування ПЗ (Software Design Quality Analysis and Evaluation),

– нотації проектування ПЗ (Software Design Notations),

– стратегія і методи проектування ПЗ (Software Design Strategies and Methods).

Базова концепція проектування ПЗ – це методологія проектування архітектури за допомогою різних методів (об'єктного, компонентного й ін.), процеси ЖЦ (стандарт ISO/IEC 12207) і техніки – декомпозиція, абстракція, інкапсуляція й ін. На початкових стадіях проектування предметна область декомпозується на окремі об'єкти (при об’єктно-орієнтованому проектуванні) або на компоненти (при компонентному проектуванні). Для подання архітектури програмного забезпечення вибираються відповідні артефакти (нотації, діаграми, блок-схеми і методи).

Ключові питання проектування – це декомпозиція програм на функціональні компоненти для незалежного і одночасного їхнього виконання, розподіл компонентів у середовищі функціонування і їх взаємодія між собою, забезпечення якості і живучості системи й ін.

Проектування архітектури ПЗ проводиться архітектурним стилем, заснованим на визначенні основних елементів структури – підсистем, компонентів, об'єктів і зв'язків між ними.

Архітектура проекту – високорівневе подання структури системи і специфікація її компонентів. Архітектура визначає логіку системи через окремі компоненти системи настільки детально, наскільки це необхідно для написання коду, а також визначає зв'язки між компонентами. Існують і інші види подання структур, засновані на проектуванні зразків, шаблонів, сімейств програм і каркасів програм.

Один з інструментів проектування архітектури – *патерн (шаблон)*. Це типовий конструктивний елемент ПЗ, що задає взаємодію об'єктів (компонентів) проектованої системи, а також ролі і відповідальності виконавців. Основна мова опису – UML. Патерн може бути *структурним*, що містить у собі структуру типової композиції з об'єктів і класів, об'єктів, зв'язків і ін.; *поведінковим*, що визначає схеми взаємодії класів об'єктів і їх поведінку, задається діаграмами діяльності, взаємодії, потоків керування й ін.; *погоджувальним*, що відображає типові схеми розподілу ролей екземплярів об'єктів і способи динамічної генерації структур об'єктів і класів.

Аналіз і оцінка якості проектування ПЗ – це заходи щодо аналізу сформульованих у вимогах атрибутів якості, функцій, структури ПЗ, з перевірки якості результатів проектування за допомогою метрик (функціональних, структурних і ін.) і методів моделювання і прототипування.

Нотації проектування дозволяють представити опис об'єкта (елемента) ПЗ і його структуру, а також поведінку системи за цим об’єктом. Існує два типи нотацій: структурна, поведінкова, та множина їх різних представлень.

Структурні нотації – це структурне, блок-схемне або текстове подання аспектів проектування структури ПЗ з об'єктів, компонентів, їх інтерфейсів і взаємозв'язків. До нотацій відносять формальні мови специфікацій і проектування: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD(Entity–Relation Diagrams), IDL (Interface Description Language) тощо. Нотації містять у собі мовний опис архітектури й інтерфейсу, діаграм класів і об'єктів, діаграм сутність–зв'язок, конфігурації компонентів, схем розгортання, а також структурні діаграми, що задають у наочному вигляді оператори циклу, розгалуження, вибору і послідовності.

Поведінкові нотації відбивають динамічний аспект роботи системи та її компонентів. Ними можуть бути діаграми потоків даних (Data Flow), діяльності (Activity), кооперації (Colloboration), послідовності (Sequence), таблиці прийняття рішень (Decision Tables), передумови і постумови (Pre-Post Conditions), формальні мови специфікації (Z, VDM, RAISE) і проектування.

Стратегія і методи проектування ПЗ. До стратегій відносять: проектування вгору, вниз, абстрагування, використання каркасів і ін. Методи є функціонально-орієнтовані, структурні, які базуються на структурному аналізі, структурних картах, діаграмах потоків даних й ін. Вони орієнтовані на

ідентифікацію функцій і їх уточнення знизу-вгору, після цього уточнюються діаграми потоків даних і проводиться опис процесів.

В об’єктно-орієнтованому проектуванні ключову роль відіграє спадкування, поліморфізм й інкапсуляція, а також абстрактні структури даних і відображення об'єктів. Підходи, орієнтовані на структури даних, базуються на методі Джексона і використовуються для подання вхідних і вихідних даних структурними діаграмами. Метод UML призначений для опису сценаріїв роботи проекту у наочному діаграмному вигляді. Компонентне проектування ґрунтується на використанні готових компонентів (reuse) з визначеними інтерфейсами і їх інтеграції в конфігурацію, як основи розгортання компонентної системи для її функціонування в операційному середовищі.

Формальні методи опису програм ґрунтуються на специфікаціях, аксіомах, описах деяких попередніх умов, твердженнях і постумовах, що визначають заключну умову одержання програмою правильного результату. Специфікація функцій і даних, якими ці функції оперують, а також умови і твердження – основа доведення правильності програми.

2.3. Конструювання програмного забезпечення

Конструювання ПЗ – створення ПЗ з конструкцій (блоків, операторів, функцій) і його перевірка методами верифікації і тестування. До інструментів конструювання ПЗ віднесені мови конструювання, програмні методи й інструментальні системи (компілятори, СКБД, генератори звітів, системи керування версіями, конфігурацією, тестуванням й ін.). До формальних засобів опису процесу конструювання ПЗ, взаємозв'язків між людиною і комп'ютером з урахуванням середовища оточення віднесені структурні діаграми Джексона.

Область знань «Конструювання ПЗ (Software Construction)» містить у собі такі розділи:

– зниження складності (Reduction in Complexity),

– попередження відхилень від стилю (Anticipation of Diversity),– структуризація перевірок (Structuring for Validation),

– використання стандартів (Use of External Standards).

Зниження складності – це мінімізація, зменшення і локалізація складності конструювання.

Мінімізація складності – це обмеження на обробку складних структур і великих обсягів інформації протягом тривалого періоду часу. Вона досягається, зокрема, використанням у процесі конструювання простих елементів, а також рекомендацій стандартів.

Зменшення складності в конструюванні ПЗ досягається шляхом створення простого коду, що легко читається і спрощує тестування, підвищує продуктивність і впливає на досягнення інших характеристик і обмежень проекту. Зменшення складності спрощує процеси верифікації і тестування результатів конструювання елементів ПС.

Локалізація складності – це спосіб конструювання з застосуванням об’єктно-орієнтованого підходу, що лімітує інтерфейс об'єктів, спрощує їхню

взаємодію, перевірку правильності самих об’єктів і зв'язків між ними. Локалізація призначена для внесення змін, пов'язаних з виявленими помилками в коді, або коли джерелом помилок є середовище, у якому виконується код.

Попередження відхилень від стилю. Для розв’язання різних задач конструювання застосовуються різні стилі конструювання (лінгвістичний, формальний, візуальний).

Лінгвістичний стиль заснований на використанні словесних інструкцій і виразів для подання окремих елементів (конструкцій) програм. Він призначений для конструювання нескладних конструкцій і приводиться до вигляду традиційних функцій і процедур або реалізується методами логічного і функціонального програмування й ін.

Формальний стиль використовується для точного й однозначного визначення компонентів системи, мінімальної кількості помилок, що можуть виникнути в зв'язку з неоднозначністю визначень або невдалих узагальнень об'єктів конструювання ПЗ.

Візуальний стиль – найбільш універсальний для конструювання прикладного ПЗ. Він дозволяє представляти елемент конструювання у наочному вигляді. Візуальна мова проектування UML надає розробнику набір діаграм для подання статичної і динамічної структур ПЗ. При його застосуванні створюється текстовий і діаграмний опис конструктивних елементів ПЗ, який виводиться на екран дисплея для перегляду і коригування.

Структуризація перевірок припускає, що побудова ПС структурована таким чином, що спрощується пошук помилок, дефектів і різних збоїв у процесі перевірок як на стадії незалежного тестування, так і в процесі експлуатації. Структуризації перевірок сприяють огляд, інспектування, спільний перегляд, модульне тестування із застосуванням автоматизованих засобів тестування й ін.

Використання зовнішніх стандартів. Конструювання ПЗ залежить від застосовних зовнішніх стандартів, пов'язаних з мовами програмування, інструментальними засобами й інтерфейсами. При конструюванні має бути визначений достатній набір стандартів для керування і забезпечення координації між визначеними видами діяльності і групами операцій, мінімізації складності, внесення змін, аналізу ризиків тощо.

До таких стандартів відносять: мови програмування (Java, Ада 95, C++ і ін.), інтерфейси мов програмування (МП) і прикладні інтерфейси платформ Windows (COM, DCOM), CORBA і ін. При конструюванні використовують стандарти мовопису даних (XML, SQL і ін.), засобів комунікації (COM, CORBA і ін.), інтерфейсних мов (POSIX, IDL, APL), UML і ін.

Перелічені вище розділи області знань «Конструювання ПЗ» у ядрі знань SWEBOK об’єднуються в групу «Основи конструювання». Крім того, розглядаються групи розділів «Керування конструюванням» та «Практичні міркування». Опишемо першу з них детальніше.

Керування конструюванням – це керування процесом конструювання ПЗ, планування, оцінка виконання плану і розроблення заходів щодо внесення змін.

Моделі конструювання містять у собі набір операцій, послідовність дій і результатів. Види моделей визначаються стандартом ЖЦ, методологіями і практиками. Деякі стандарти ЖЦ за своєю природою орієнтовані на конструювання засобами екстремального програмування і раціонального уніфікованого процесу – RUP (Rational Unified Process) [13].

Планування – це визначення порядку операцій, термінів і рівня виконання заданих умов у процесі конструкторської діяльності за моделлю ЖЦ, що містить у собі задачі і дії зі створення, перевірки й оцінки показників якості. Виконавці розподіляються за процесами і виконують відповідні задачі з реалізації проміжного і кінцевого продукту. Остаточний результат виміряється за обсягом коду, ступенем повторного використання, кількістю помилок і дефектів, а також оцінюванням показників якості ПЗ.

Внесення змін пов'язане з помилками, виявленими при перевірці і тестуванні, проводиться з метою збереження функціональної цілісності системи. У випадку виявлення помилок на процесі супроводження приймається рішення про внесення змін або заміну коду у цілому.

Лекція 3. Характеристика областей знань з інженерії програмного забезпечення – SWEBOK (продовження)

3.1. Тестування програмного забезпечення

Тестування ПЗ – це процес перевірки готової програми в статиці (перегляди, інспекції, налагодження вихідного коду) і в динаміці (прогін на наборі тестових даних) з метою перевірки різних шляхів виконання програми і порівняння отриманих результатів із заздалегідь заданими.

Існує дві форми перевірки коду – модульна й інтеграційна. Спочатку використовують стандарти (IEEE 829:1996 і IEEE 1008:1987) з перевірки і тестування модулів. Потім проводиться інтеграційне тестування модулів системи і їх інтерфейсів у динаміці виконання. Під час різних видів перевірок збираються дані про помилки, дефекти, відмови тощо і оформляється відповідна документація (таблиці типів помилок, частоти і часу виявлення відмов і ін.). Зібрані дані використовуються при оцінюванні характеристик якості готового ПЗ, наприклад, надійності.

Область знань «Тестування ПЗ (Software Testing)» містить у собі такі розділи:

– основні концепції і визначення тестування (Testing Basic Concepts and definitions),

– рівні тестування (Test Levels),

– техніки тестування (Test Techniques),

– метрики тестування (Test Related Measures),

– керування процесом тестування (Managing the Test Process).

Дана область знань SWEBOK визначає методи перевірки правильності ПЗ: верифікація, валідація, тестування. Наводяться типи, рівні і техніки тестування ПЗ, методи планування процесу тестування, розроблення тестових наборів даних для прогону ПЗ в режимі випробування модулів або системи в цілому і наступною оцінкою результатів тестування.

Основна концепція тестування – це базові терміни, ключові проблеми і їхній зв'язок з іншими областями знань. Тестування визначається як процес перевірки правильності програми в динаміці її виконання за тестовими даними. При тестуванні виявляються недоліки: відмови (faults) і дефекти (defects) як причини порушення роботи програми, збої (failures) як небажані ситуації, помилки (errors) як наслідки збоїв і ін. Базове поняття тестування – тест, що виконується в заданих умовах і за наборами даних. Тестування вважається успішним, якщо знайдено дефект або помилка, і вони відразу усуваються. Ступінь тестованості визначається критерієм покриття системи тестами, перевірки всіх можливих шляхів виконання програм і імовірності припущення стосовно того, що може з'явитися збій або помилкова ситуація в системі.

Рівні тестування:

– *тестування окремих елементів* – це перевірка окремих, ізольованих і незалежних одна від одної частин ПЗ;

– *інтеграційне тестування* орієнтоване на перевірку зв'язків і взаємодії компонентів (інтерфейсів), що можуть розміщуватися на різних архітектурних платформах розподіленого середовища;

– *тестування системи* – це перевірка правильності функціонування системи, пошук і виявлення відмов і дефектів у системі і їхнє усунення. При цьому контролюється виконання сформульованих не функціональних вимог (безпека, надійність і ін.) у системі, правильність подання і здійснення зовнішніх інтерфейсів системи з зовнішнім середовищем.

На всіх рівнях тестування застосовуються методи:

– *функціонального тестування*, які забезпечують перевірку реалізації функцій, що визначені у вимогах, а також правильності їх виконання;

– *регресійного тестування*, що орієнтоване на повторне вибіркове тестування системи або її компонентів після внесення в них змін на тих самих тестах, що і до модифікації;

– *тестування ефективності* – це перевірка продуктивності, пропускної здатності, максимального обсягу даних і системних обмежень відповідно до вимог;

– *стрес-тестування* – це перевірка поведінки системи при максимально припустимому навантаженні або в разі його перевищення;

– *альфа- і бета-тестування* – це тестування системи (альфа) групою тестувальників організації-розробника і тестування системи «зовнішніми» користувачами (бета);

– *конфігураційного тестування* – перевірка структури й ідентифікації системи, а також роботи системи на різних конфігураціях апаратури й устаткування.

Тестуванню підлягають також перевірка реалізації вимог і забезпечення параметрів настроювання і розміщення компонентів ПЗ на заданій кількості і типах комп'ютерів і середовища.

Техніки тестування базуються на певних теоретичних і практичних положеннях щодо проектування (компонентного, об'єктно-орієнтованого, сервісного і т.п.), а також на таких даних:

– інформація про структуру ПЗ або системи в документації («біла скринька»);

– підбір тестових наборів даних для перевірки правильності роботи компонентів і системи в цілому без знання їх структури («чорна скринька»);

– аналіз граничних значень, таблиць прийняття рішень, потоків даних, статистики відмов і ін.;

– блок-схеми побудови програм і складання наборів тестів для покриття системи цими тестами;

– виявлені і зафіксовані в таблицях системи дефекти, перед- і посту мови виконання, структурні характеристики системи (кількість модулів, обсяг даних тощо).

Метрики тестування. Для вимірювання результатів тестування ПЗ й оцінки якості використовуються метрики. Вимір як частина планування і

розробки тестів базується на розмірі програм, їх структурі і кількості виявлених помилок і дефектів. Метрики тестування – це вимірювання процесу планування, проектування і тестування, а також результатів тестування на основі таксономії відмов і дефектів, покриття границь тестування, перевірки потоків даних і ін.

Процес тестування документується і, відповідно до стандарту IEEE 829:1995, містить у собі опис тестових документів, їх зв'язку між собою і з задачами тестування. Без документації на процес тестування неможливо провести сертифікацію продукту за моделями зрілості, зокрема, моделлю CMM [11]. Після завершення тестування оцінюється вартість і ризики ПЗ, викликані збоями або недостатньо надійною роботою системи. Вартість тестування – одне з обмежень, на основі якого приймається рішення про його припинення або продовження.

Керування тестуванням:

– планування процесу тестування (складання планів, тестів, наборів даних) і оцінювання показників якості готового продукту;

– проведення тестування компонентів повторного використання і патернів як основних об'єктів складання ПЗ;

– генерація необхідних тестових сценаріїв, що відповідають середовищу виконання ПЗ;

– верифікація правильності реалізації системи і валідація реалізації вимог до ПЗ;

– збирання даних про відмови, помилки і виявлені непередбачені ситуації при виконанні програмного продукту;

– підготовка звітів за результатами тестування й оцінка характеристик системи.

Відповідно до стандарту ISO/IEC 12207 тестування ПЗ розглядається як невід'ємна частина ЖЦ.

3.2. Методи і інструменти

Методи забезпечують проектування, реалізацію і виконання ПЗ. Вони накладають деякі обмеження на інженерію ПЗ у зв'язку з особливостями застосування їхніх нотацій і процедур, а також забезпечують оцінку і перевірку процесів і продуктів. Інструменти забезпечують програмну підтримку окремих методів інженерії ПЗ для автоматизованого виконання задач процесів ЖЦ.

Область знань «Методи та інструменти інженерії ПЗ (Software Engineering Tools and Methods)» складається з розділів:

– інструменти інженерії ПЗ (Software Engineering Tools),

– методи інженерії ПЗ (Software Engineering Methods).

Методи інженерії ПЗ – це евристичні методи (heuristic methods), формальні методи (formal methods) і методи прототипування (prototyping methods).

Евристичні методи містять у собі: структурні методи, засновані на функціональній парадигмі; методи, орієнтовані на структури даних, якими

маніпулює ПЗ; об’єктно-орієнтовані методи, що розглядають предметну область як колекцію об'єктів; методи, орієнтовані на конкретну область застосування, наприклад, на системи реального часу, безпеки та ін.

Формальні методи засновані на формальних специфікаціях, аналізі, доведенні і верифікації програм. Специфікація записується мовою, синтаксис і семантика якої визначені формально і засновані на математичних концепціях(алгебрі, теорії множин, логіці). Розрізняються наступні категорії формальних методів:

– *мови і нотації специфікації* (specification languages and notations), орієнтовані на модель, властивості і поведінку;

– *уточнення специфікації* (refinement specification) шляхом трансформації вкінцевий результат, близький до кінцевого програмного продукту, що виконується;

– *методи верифікації/доведення* (verification/proving properties), що використовують твердження (теореми), перед- і постумови, формально описуються і застосовуються для встановлення правильності специфікації програм.

Методи доведення застосовувалися в основному в теоретичних експериментах. Понад 25 років їх застосування було обмежено через трудомісткість і економічну невигідність. У 2005 р. проблема верифікації знову набула актуальності у запропонованому новому міжнародному проекті «Цілісний автоматизований набір інструментів для перевірки коректності ПС» (Т. Хоар, «Открытые системы», 2006, № 6), який поставив наступні перспективні задачі:

– розробка єдиної теорії побудови й аналізу програм;– побудова багатостороннього інтегрованого набору інструментів верифікації на усіх виробничих процесах

– розроблення формальних специфікацій, їх доведення і перевірка правильності, генерація програм і тестових прикладів, уточнення, аналіз і оцінка;

– створення репозитарію формальних специфікацій, верифікованих програмних об'єктів різних типів і видів.

Формальні методи верифікації будуть охоплювати всі аспекти створення і перевірки правильності програм. Це приведе до створення потужної верифікованої виробничої основи і сприятиме значному зменшенню помилок у ПЗ.

Методи прототипування (Prototyping Methods) засновані на використанні прототипу ПЗ для моделювання на ньому завдань нової системи і базуються на:

– стилях прототипування, що уособлюють тривалість використання прототипів, наприклад, стиль створення тимчасово використовуваних прототипів (throw away),

– моделях еволюційного прототипування – перетворення прототипу вкінцевий продукт і розроблення специфікацій, відповідно до якої він виконується;

– техніках оцінки/дослідження (evaluation) результатів прототипування.

Інструменти інженерії ПЗ забезпечують автоматизовану підтримку процесів розроблення ПЗ і містять у собі множину інструментів, що охоплюють усі процеси ЖЦ.

Інструменти роботи з вимогами (Software Requirements Tools) – це:

– інструменти розробки (Requirement Development) і керування вимогами (Requirement Management), орієнтовані на аналіз, збирання, специфікування і перевірку вимог;

– інструменти трасування вимог (Requirement traceability tools) є невід'ємною частиною роботи з вимогами, їх функціональний зміст залежить від складності проектів і рівня зрілості процесів.

Інструменти проектування (Software Design Tools) – це інструменти для створення ПЗ із застосуванням базових нотацій (структурної SADT/IDEF, моделювання UML і т.п.).

Інструменти конструювання ПЗ (Software Construction Tools) – це інструменти для трансляції і об’єднання програм. До них належать:

– редактори програм (program editors) і програми редагування загального призначення;

– компілятори і генератори коду (compilers and code generators) як самостійні засоби об'єднання програмних компонентів в інтегрованому середовищі для одержання вихідного продукту з використанням препроцесорів, складальників, завантажників і ін.;

– інтерпретатори (interpreters), які забезпечують контрольоване виконання програм за їх описом. Намітилася тенденція злиття інтерпретаторів і компіляторів(наприклад, Java, в .NET);

– відлагоджувачі (debuggers), призначені для перевірки правильності опису вихідних програм і усунення помилок;– інтегроване середовище розробки (IDE – integrated development environment) та бібліотеки компонентів (libraries components), що є утворюють середовище виконання процесу розроблення ПС;

– програмні платформи (Java, J2EE і Microsoft .NET) і платформи для розподілених обчислень (CORBA і WebServices, тощо).

Інструменти тестування (Software Testing Tools) – це:

– генератори тестів (test generators), що допомагають у розробці сценаріїв тестування;

– засоби виконання тестів (test execution frameworks), які забезпечують виконання тестових сценаріїв і відслідковують поведінку об'єктів тестування;

– інструменти оцінки тестів (test evaluation tools), які підтримують оцінювання результатів виконання тестів і ступеня відповідності поведінки тестованого об'єкта очікуваній поведінки;

– засоби керування тестами (test management tools), які забезпечують інженерне керування процесом тестування ПЗ;

– інструменти аналізу продуктивності (performance analysis tools), кількісної її оцінки та оцінки поводження програм у процесі виконання.

Інструменти супроводу (Software Maintenance Tools) містять у собі:

– інструменти полегшення розуміння (comprehension tools) програм, наприклад, різні засоби візуалізації;Розділ 1 45– інструменти реінженерії (reengineering tools) підтримують діяльність з перетворення програм і зворотної інженерії (reverse engineering) для відновлення (артефактів, специфікації, архітектури) застарілого ПЗ або генерації нового продукту.

Інструменти конфігураційного керування (Software Configuration Management Tools) – це:

– інструменти відстеження (tracking) дефектів;

– інструменти керування версіями;– інструменти керування складанням, випуском версії (конфігурації) продукту та його інсталяції.

Інструменти керування інженерною діяльністю (Software Engineering Management Tools) підрозділяються на:

– інструменти планування і відстеження ходу проектів, кількісної оцінки зусиль і вартості робіт у проекті (наприклад, Microsoft Project 2003);

– інструменти керування ризиками, які використовуються для ідентифікації, моніторингу ризиків і оцінки нанесеного ушкодження;

– інструменти кількісної оцінки властивостей ПЗ шляхом вимірювань і розрахунків остаточного значення надійності і якості.

Інструменти підтримки процесів (Software Engineering Process Tools) розділені на:

– інструменти моделювання та опису моделей ПЗ (наприклад, UML і його інструменти);

– і нструменти керування програмними проектами (наприклад, Microsoft Project);

– інструменти керування конфігурацією для підтримки версій і всіх артефактів проекту.

Інструменти забезпечення якості (Software Quality Tools) діляться на дві категорії:

– інструменти інспектування для підтримки перегляду (review) і аудиту;

– інструменти статичного аналізу артефактів, даних, потоків робіт і перевірки їх властивостей на відповідність показникам.

Додаткові аспекти інструментального забезпечення (Miscellaneous Tool Issues) стосуються:

– техніки інтеграції інструментів (платформ, представлень, процесів, даних) для їх природного сполучення в інтегрованому середовищі;

– метаінструментів для генерації інших інструментів для ПЗ;

– оцінки інструментів при їх еволюції.

3.3. Якість програмного забезпечення

Якість ПЗ – набір властивостей продукту (сервісу або програм), що характеризують його здатність задовольнити встановлені або передбачувані потреби замовника. Поняття якості має різні інтерпретації залежно від

конкретної програмної системи і вимог до неї. Крім того, у різних джерелах таксономія(класифікація) характеристик у моделі якості розрізняється.

Моделі мають різну кількість рівнів і повністю або частково збігаються щодо набору характеристик якості. Наприклад, модель якості МакКолла на найвищому рівні має три характеристики: функціональність, модифікованість і переносність, а на нижчих рівнях моделі – 11 підхарактеристик якості і 18 критеріїв (атрибутів) якості.

Стандарт ISO 9126:2001 регламентує *зовнішні і внутрішні характеристики* якості. Перші відображають вимоги до функціонування програмного продукту. Для кількісного встановлення критеріїв якості, за якими буде здійснюватися перевірка і підтвердження відповідності ПЗ заданим вимогам, визначаються відповідні зовнішні вимірювані властивості (зовнішні атрибути*)* ПЗ, метрики (наприклад, час виконання окремих компонентів), діапазони зміни значень і моделі їх оцінки. Метрики використовуються на стадії тестування або функціонування і називаються *зовнішніми метриками.* Вони являють собою моделі оцінки атрибутів.

Внутрішні характеристики якості і внутрішні атрибути ПЗ використовуються для складання плану досягнення необхідних зовнішніх характеристик якості продукту. Для квантифікації внутрішніх характеристик якості застосовують внутрішні метрики, як інструмент перевірки відповідності проміжних продуктів внутрішнім вимогам до якості, які формулюються на процесах, що передують тестуванню.

Зовнішні і внутрішні характеристики якості відображають властивості самого ПЗ (працюючого або не працюючого), а також погляд замовника і розробника на таке ПЗ. Безпосереднього кінцевого користувача ПЗ цікавить експлуатаційна якість ПЗ – сукупний ефект від досягнення характеристик якості, що виміряється строком результату, а не властивістю самого ПЗ. Це поняття ширше, ніж будь-яка окрема характеристика (наприклад, зручність використання або надійність).

Остаточна оцінка якості проводиться відповідно до стандарту ISO/IEC 14598. Якість може підвищуватися за рахунок постійного поліпшення використовуваного продукту виявленням, усуненням дефектів у ПЗ і їх запобіганням.

Область знань «Якість ПЗ (Software Quality)» складається з наступних розділів:

– концепції якості ПЗ (Software Quality Concepts);

– визначення і планування якості (Definition & Planning for Quality);

– техніки й види діяльності, що забезпечують гарантію якості, валідацію і верифікацію (Activities and Techniques for Software Quality Assurance, Validation &Verification –V&V);

– вимірювання при аналізі якості ПЗ (Measurement in Software Quality Analysis).

Концепції якості ПЗ – це зовнішні і внутрішні характеристики якості, їхні метрики, а також моделі якості, визначені на множині цих характеристик, що

наведені в стандартах з якості і в [8, 9] – це шість характеристик і кожна з них має кілька атрибутів. До характеристик якості належать:

– функціональність (functionality);
– надійність (realibility);
– зручність застосування (usability);
– ефективність (efficiency);
– супровід (maitainnability);
– переносність (portability).

Базова модель якості містить у собі ці характеристики і вони притаманні будь-якому типу програмних продуктів. При розробці вимог замовник формулює такі вимоги до якості, які найбільшою мірою підходять для програмного продукту, який замовляється.

Визначення і планування якості ПЗ ґрунтується на положеннях стандартів у цій області, складанні планів і графіків робіт, процедурах перевірки і ін. План забезпечення якості містить у собі набір дій для перевірки процесів забезпечення якості (верифікація, валідація і ін.) і формування документа з керування якістю.

Планування якості призначено для підтримки керування процесами досягнення якості продуктів проекту (зокрема проміжних робочих) і ресурсів – програмних, технічних, виконавських і ін. Воно також передбачає керування вимогами до процесів і продуктів і полягає в наступному:

– визначення продукту термінами заданими характеристиками якості;
– планування процесів для гарантії одержання необхідної якості;
– вибір методів оцінки запланованих характеристик якості і встановлення відповідності продукту сформульованим вимогам.

У стандарті ISO/IEC 12207 визначені спеціальні процеси забезпечення якості – верифікація, валідація (атестація), спільний аналіз і аудит.

Види діяльності і техніки гарантії якості містять у собі, зокрема: інспекцію, верифікацію і валідацію ПЗ.

Інспекція ПЗ – аналіз і перевірка різних видів подання системи і ПЗ (специфікації, архітектурної схеми, діаграм, початкового коду тощо). Виконується на всіх процесах ЖЦ розробки ПЗ.

Верифікація ПЗ – процес забезпечення правильної реалізації ПЗ відповідно до специфікацій, виконується протягом усього життєвого циклу. Верифікація дає відповідь на питання, чи правильно створюється система.

Валідація – процес перевірки відповідності ПЗ функціональним і нефункціональним вимог і очікуваним потребам замовника.

Верифікація і валідація (V&V) можуть виконуватися, починаючи з ранніх стадій ЖЦ. Вони орієнтовані на отримання правильних функцій ПЗ, плануються і забезпечуються визначеними ресурсами з чітким розподілом ролей. Перевірка ґрунтується на використанні відповідних технік тестування для виявлення тих або інших дефектів і збирання статистики. Після зібрання даних оцінюється правильність реалізації вимог і роботи ПЗ у заданих умовах.

Вимірювання при аналізі якості ПЗ ґрунтується на метриках продукту і даних, зібраних у процесі створення продукту при заданих ресурсах: оцінок процесів, ПЗ і його моделей, і передбачає документування вимірів. Оцінювання якості продукту полягає у вимірюванні і оцінюванні якісних показників за допомогою даних про різні типи помилок і відмов під час тестування ПЗ і виконання коду на тестових даних. Ці дані аналізуються, перевіряються і використовуються при якісній і кількісній оцінки ПЗ.

Для імітації роботи системи в режимі тестування розробляються тести з реальними вхідними даними для перевірки правильності роботи ПЗ на різних фрагментах програми і шляхах проходження в них операторів. У процесі тестування ПЗ виявляються різного роду помилки (відмови, дефекти, помилки тощо), кількість яких значною мірою може вплинути на одержання правильного і якісного результату.

З урахуванням типів виявлених помилок можна встановити наявність (або відсутність) відповідності реалізованих і нереалізованих функцій, заданих у вимогах до системи, а також оцінити способи реалізації нефункціональних вимог (продуктивності, надійності та ін.). Оцінюються процеси керування планами, інспекціями, прогонами і т.п. За цими оцінками приймаються рішення про завершення розробки продукту проекту і передачі його замовнику в експлуатацію, під час якої можуть бути внесені зміни щодо усунення помилок, визначення адекватності планів і вимог, оцінки ризиків перероблення ПЗ тощо.

Мета інспекцій – виявлення різних аномальних станів у ПЗ незалежними фахівцями команди експертів із залученням авторів проміжного або кінцевого продукту. Експерти інспектують виконання вимог, інтерфейси, вхідні дані і т.п., а потім документують виявлені відхилення в проекті. Призначення аудита – незалежна оцінка продуктів і процесів на відповідність регулюючим і регламентуючим документам (планам, стандартам і ін.), формулювання звіту про випадки невідповідності і пропозицій для їх коригування.

Таким чином, розгляд розділів SWEBOK свідчить по те, що це ядро містить весь необхідний набір знань з програмної інженерії, який повинні мати фахівці різних профілів (аналітики, інженери, програмісти, оцінювачі, контролери тощо), що виробляють програмний продукт.

Зазначимо, що ядро знань SWEBOK не позбавлено недоліків тактичного характеру. Так, між областями знань у цьому ядрі існують перетини за методами, концепціями і стратегіями, а деякі важливі напрями програмної інженерії взагалі не відбиті у ньому наявними областями знань. Це стосується, наприклад, методів доведення правильності програм, еволюції програм, розподілених і неоднорідних середовищ, взаємодії систем, таких методів програмування, як аспектне, агентне, сервісне й інші, а також аспектів захисту, безпеки тощо.

Список літератури до Лекцій 2-3.

1. *Рекомендации* по преподаванию программной инженерии и информатики в университетах. – Computing Curricula-2001: Computer Science.– Пер. с англ. – Интернет – Ун. информац. технологий, М.: 2007.–462с.

2. *Бабенко Л.П., Лавріщева К.М.* Основи програмної інженерії. – Навч.посібник. – К.: Знання, 2001. – 269 с.

3. *Лаврищева Е.М., Грищенко В.Н*. Области знаний программной инженерии – SWEBOK и подход к обучению этой дисциплине // Управляющие системы и машины. –2005. – №1.–С.38–4.

4. *Pfleeger S.L.* Software Engineering. Theory and practice. – Printice Hall: UpperSaddenle River, New Jersey, 1998. – 576 p.

5. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing. – New York: Addison-Wesley Publ. Co., 1994. –529 p.

6. *Иан Соммервил.* Инженерия программного обеспечения. 6-е издание. – М.;Спб. – Киев, 2002. – 623 с.

7. *Лавріщева К.М.* Основні напрямки досліджень в програмній інженерії і шляхи їхнього розвитку // Проблеми програмування. – 2003. – № 3–4. –С. 44–58.

8. *Лаврищева Е.М.* Методы программирования. Теория, инженерия, практика.– К.: Наук. думка, 2006.–450с.

9. *Основы инженерии* качества программных систем / Ф.И.Андон, Г.И.Коваль,Т.М. Коротун, Е.М.Лаврищева, В.Ю. Суслов – К.: Академпериодика.–2007. – 678 с.

10. *Лаврищева Е.М., Коваль Г.И., Коротун Т.М.* Подход к управлению качеством программных систем обработки данных // Кибернетика и системный анализ. – 2006.–№ 5.–С.174–185.

11. *Capability Maturity* Model for Software, Version 1.1 / M.Paulk, B.Curtis et al.// CMU–SEI–TR–024, Soft. Engin. Institute, Pittsburg PA 15213, Feb. –Pittsburg, 1993. – 82 p.

12. *Thayer R.H., ed.* Software Engineering Project Management, 2 nd. ed., IEEE CSPress, Los Alamitos, Calif., 1997. – 391 p.

13. *Кендалл С.* Унифицированный процесс. Основные концепции.–М.;– СПб.–Киев.–2002.– 157с.

Лекція 4. СТАНДАРТ І МОДЕЛІ ЖИТТЄВОГО ЦИКЛУ

За десятки років розробки програмного забезпечення і програмних систем створено низку типових схем упорядкування виконання робіт з проектування і розроблення. Такі схеми одержали назву життєвого циклу і узагальнені в стандарті ISO/IEC 12207 і основоположних моделях ЖЦ, що застосовуються на практиці.

4.1 Характеристика життєвого циклу стандарта ISO/IEC 12207

Стандарт ISO/IEC 12207:2002 визначає загальну структуру і зміст ЖЦ ПС, починаючи з розробки концепції до утилізації системи. Структурно він складається з опису багатьох процесів, взаємозв'язків між ними, а також сформульованих дій і задач, виконуваних у цих процесах. Іншими словами, стандартний життєвий цикл визначає лише схему робіт за процесами розробки ПС, а не те, як саме виконувати і або інші процеси (табл.4.1).

Таблиця 4.1 – Процеси життєвого циклу в стандарті ISO/IEC 12207

№ п/п		Процес (підпроцес)
1. Категорія «Основні процеси»		
1.1	Замовлення (договір)	
	1.1.1	Підготовка замовлення, вибір постачальника
	1.1.2	Моніторинг діяльності постачальника, приймання споживачем
1.2	Постачання (придбання)	
	Розроблення	
	1.3.1	Виявлення вимог
	1.3.2	Аналіз вимог до системи
	1.3.3	Проектування архітектури системи
	1.3.4	Аналіз вимог до ПЗ системи
	1.3.5	Проектування ПЗ
	1.3.6	Конструювання (кодування) ПЗ
	1.3.7	Інтеграція ПЗ
	1.3.8	Тестування ПЗ
	1.3.9	Системна інтеграція
	1.3.10	Системне тестування
	1.3.11	Інсталяція ПЗ
1.4	Експлуатація	
	1.4.1	Функціональне використання
	1.4.2	Підтримка споживача
1.5	Супроводження	
2. Категорія «Процеси підтримки»		
2.1	Документування	
2.2	Керування конфігурацією	
2.3	Забезпечення гарантії якості	
2.4	Верифікація	
2.5	Валідація	

№ п/п		Процес (підпроцес)
2.6	Загальний огляд	
2.7	Аудит	
2.8	Вирішення проблем	
2.9	Забезпечення застосовності продукту	
2.10	Оцінювання продукту	
3. Категорія «Організаційні процеси»		
3.1	Керування	
	3.1.1	Керування на рівні організації
	3.1.2	Керування проектом
	3.1.3	Керування якістю
	3.1.4	Керування ризиком
	3.1.5	Організаційне забезпечення
	3.1.6	Вимірювання
	3.1.7	Керування знаннями
3.2	Удосконалення	
	3.2.1	Упровадження процесів
	3.2.2	Оцінювання процесів
	3.2.3	Удосконалення процесів

Стандарт не зобов'язує використовувати всі процеси ЖЦ одночасно і не ставить особливих вимог до формату і змісту розроблених документів. Тому організація-користувач стандарту під час розроблення конкретного програмного продукту може створити стандарти підприємства, методики і процедури, що деталізуватимуть вибрані для конкретних потреб процеси ЖЦ. Міжнародна організація зі стандартизації ISO (International Organization for Standardization) випускає також посібники і настанови, що доповнюють стандарт ISO/IEC 12207.

Як видно з табл. 4.1, усі процеси в даному стандарті поділяються на три категорії:

– основні процеси;

– процеси підтримки;

– організаційні процеси.

Для кожного з процесів визначені види діяльності (дії – activity), задачі, сукупність результатів (виходів) діяльності і розв’язання задач, а також деякі специфічні вимоги. У стандарті наведено перелік робіт для основних, організаційних процесів і процесів підтримки, але не спосіб їх виконання і не форма подання результатів.

В стандарті до основних процесів належать:

– *процес придбання*, який ініціює ЖЦ ПС і визначає дії організації-покупця (або замовника), що отримує автоматизовану систему, програмний продукт або сервіс. Цей процес містить у собі такі види діяльності: ініціювання і підготовка запиту, оформлення контракту і його актуалізація; моніторинг користувачів, приймання і завершення;

– *процес постачання*, який визначає дії з передачі покупцю програмного продукту або сервісу і містить у собі такі види діяльності: підготовку пропозицій(відповідей на запити); оформлення контракту; планування, виконання і контроль продукту, що постачається; аналіз і оцінку продукту; постачання і завершення робіт з постачання. Процес постачання починається тоді, коли встановлені договірні відношення між замовником і постачальником. Залежно від умов договору процес постачання може містити у собі процес розроблення ПЗ, процес експлуатації і супроводження для виправлення і поліпшення ПС;

– *процес розроблення*, який визначає дії підприємства-розробника програмного продукту: аналіз вимог до системи; проектування архітектури системи, детальне проектування компонентів ПС, кодування і тестування ПС, інтеграцію системи, кваліфікаційне тестування, установку ПС і забезпечення приймання ПС (рисунок 4.1);

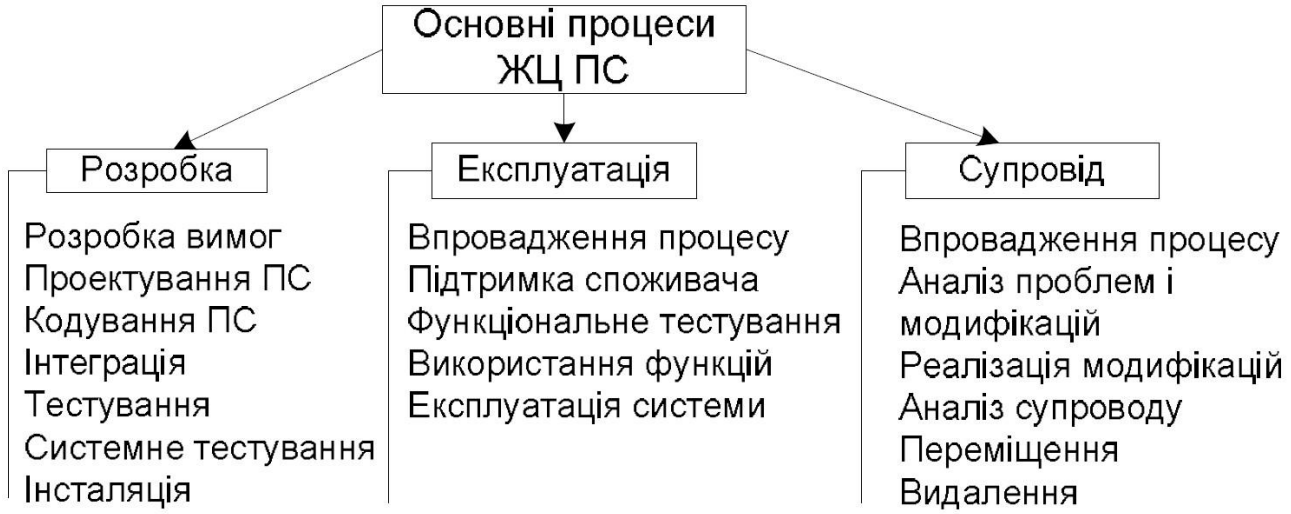

Рисунок 4.1 – Схема основних процесів ЖЦ ПС

– *процес експлуатації*, який визначає дії підприємства-оператора, що забезпечує обслуговування системи в ході її експлуатації користувачами (консультування користувачів, вивчення потреб операторів, задоволеності споживачів системою тощо). Цей процес регламентує задачі і дії з функціонального тестування, перевірки правильності експлуатації системи; дотримання інструкцій і настанов з її запуску;

– *процес супроводження*, який визначає дії організації, що виконує супровід програмного продукту (керування модифікаціями, підтримку поточного стану і функціональної придатності, інсталяцію програмного продукту на обчислювальній системі користувача та її вилучення при списанні). Даний процес містить у собі завдання і дії щодо аналізу питань супроводження і модифікації, розробки планів і реалізації модифікації системи, аналізу результатів супроводження після змін системи, міграції ПС в інше середовище або її виведення з експлуатації.

До категорії основних процесів належать також «первинні» процеси, що визначають порядок підготовки договору на розробку ПС, моніторинг діяльності постачальників ПС тощо (див. таблицю 4.1).

Стандарт містить у собі опис допоміжних процесів, що регламентують додаткові дії з перевірки продукту, керування проектом та його якістю (рисунок 4.2).

До процесів підтримки розроблення ПС належать: документування, керування версіями, верифікація і валідація, перегляди, аудити, оцінювання продукту та ін. Процес керування версіями за змістом відповідає керуванню конфігурацією системи, що так само, як і продукти процесів, повинні перевірятися на правильність реалізації цілей проекту і відповідність вимогам замовника. Завдання з перевірки рекомендується виконувати спеціальним контролерам, які знаються на методах і процесах проектування ПС.

До організаційних процесів належать: керування проектом (менеджмент розробки), якістю, ризиками тощо.

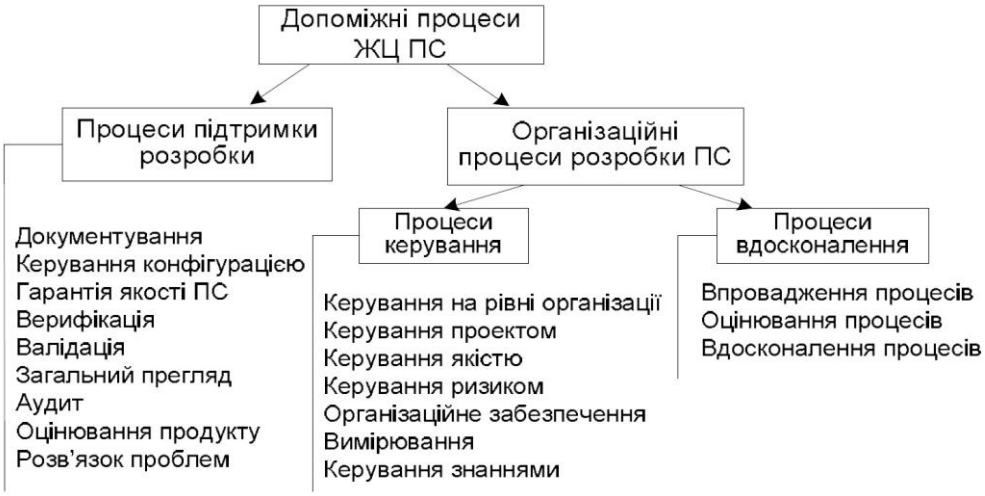

Рисунок 4.2 – Схема допоміжних процесів ЖЦ ПС

Ці процеси виконуються спеціальними службами, що здійснюють планування робіт у проекті, контроль процесів, визначення метрик для вимірювання продуктів, перевірку показників якості, дотримання стандартних положень та ін.

Процеси, дії і задачі наведені в стандарті в найбільш загальній природній послідовності. Залежно від цілей конкретного проекту головний розробник і менеджер вибирають процеси, дії і задачі, вибудовують визначену схему ЖЦ для застосування в цьому проекті.

Процеси керування в стандарті структуровані за рівнями і напрямами, вони жодним чином не зв’язані з існуючими методами і засобами програмної інженерії з розроблення ПС. Це дає можливість при їх виборі для застосування у ЖЦ зіставляти з ними звичні парадигми і методи розроблення (об'єктні, компонентні, сервісні й ін.) та засоби ядра знань SWEBOK.

Таким чином, між стандартом ISO/IEC 12207 і ядром знань SWEBOK існує зв'язок: вони взаємодоповнюють та збагачують один одного, більше у розробці

відповідних документів брали участь одні ті самі висококваліфіковані фахівці в галузі програмування й інформатики. Інженерна дисципліна проектування ПС використовує теоретичні, прикладні методи і засоби розробки ПС і стандарти (ISO/IEC 12207, ISO/IEC 15404, ISO/IEC 9126 та ін.), а також рекомендації і методики керування розробкою, до яких відносять методи оцінки на процесах ЖЦ, якості ПС, витрачених ресурсів і вартості виконаних робіт. При цьому ядро знань SWEBOK, а також численні монографії і статті рекомендують до застосування методи і засоби програмної інженерії, а стандарт дає настанови до побудови процесів на стандартизованій інженерній основі.

Природно, що в невеликих програмних проектах завжди можна буде застосовувати творчі і неформальні підходи, запропоновані фахівцями для створення різного роду унікальних продуктів, процес розроблення яких не завжди відповідає загальному стандарту.

4.2. Формування прикладних моделей життєвого циклу

Кожна ПС протягом свого існування проходить визначену послідовність *процесів*, починаючи від постановки задачі до її втілення в готову програму, наступної експлуатації і остаточного виведення з експлуатації та списання. Така послідовність процесів називається життєвим циклом розробки ПС. На кожному процесі ЖЦ виконується визначена сукупність процесів і/або підпроцесів, кожний з яких породжує відповідний проміжний продукт, використовуючи при цьому результати попереднього процесу і доробок.

Модель життєвого циклу – це схема виконання робіт і задач у рамках процесів, що забезпечують розробку, експлуатацію і супровід програмного продукту. Ця схема відображає еволюцію ПС, починаючи від формулювання вимог і закінчуючи припиненням користування нею.

Історично така схема робіт містить у собі:

– розробку вимог або технічного завдання;

– розробку ескізного або технічного проекту;

– програмування або робоче проектування;

– пробну експлуатацію;

– супровід і поліпшення;

– зняття з експлуатації.

Основне призначення моделей ЖЦ є таким:

– планування і розподіл робіт і ресурсів між розробниками, а також керування програмним проектом;

– забезпечення взаємодії між розробниками проекту і замовником;

– спостереження і контроль робіт, оцінка проміжних продуктів ЖЦ на дотримання специфікацій вимог, правильне їх виконання, оцінка продукту і реальних витрат, у тому числі і щодо застосовуваних програмних засобів і інструментів;

– узгодження проміжних результатів із замовником;

– перевірка правильності кінцевого продукту шляхом його тестування на запланованих і погоджених із замовником наборах тестів;

– оцінка відповідності характеристик якості отриманого продукту заданим вимогам;

– обговорення використовуваних процесів ЖЦ з метою оцінки їх потенційних можливостей і недоліків, що виявлялися при їх застосуванні, а також визначення напрямів удосконалення або модернізації ЖЦ.

Отже, для побудови конкретної моделі ЖЦ ПС, що задовольняє концептуальній ідеї проектованої системи з урахуванням її складності і масштабу робіт, необхідно зробити правильний вибір процесів, їх завдань і дій відповідно до стандарту. На сьогодні основою формування нової моделі ЖЦ для конкретної прикладної системи є стандарт ISO/IEC 12207, що задає повний набір процесів (більш 40), охоплює всі можливі види робіт і завдань, пов'язаних з побудовою ПС, починаючи з аналізу предметної області і закінчуючи виготовленням відповідного продукту. Стандарт ISO/IEC 12207 надає загальний опис процесів на найвищому рівні, проте він не покликаний деталізувати виконання дій або задач, з яких складаються процеси. Він також не ставить вимоги до формату і змісту документів, що випускаються на різних процесах.

Процеси, дії і задачі наведені в стандарті в найбільш загальній природній послідовності, але це не означає, що в такій самій послідовності вони повинні бути застосовані в конкретній моделі ЖЦ ПС. Залежно від проекту процеси, дії і задачі стандарту вибираються, упорядковуються і включаються в модель ЖЦ. При застосуванні вони можуть перекривати, переривати один одного, виконуватися ітераційне або рекурсивно. Це визначає «динамічний» характер стандарту і дозволяє реалізувати з його допомогою довільну модель ЖЦ ПС.

Тому організаціям, що будуть застосовувати стандарт у своїй роботі, знадобляться додаткові внутрішні стандарти або процедури, які визначатимуть різні деталі застосування вибраних елементів ЖЦ залежно від типу ПС. Крім цього, існують міжнародні стандарти з керування конфігурацією ПЗ, супроводу, документування, оцінювання якості, верифікації і валідації, тестування тощо.

Зі стандарту ISO/IEC 12207 можна вибирати тільки ті процеси, що найбільше підходять для реалізації конкретної ПС. Обов'язковими є основні процеси, що є у всіх відомих моделях ЖЦ. Залежно від цілей і задач предметної області вони можуть бути поповнені процесами з категорії організаційних процесів даного стандарту. Розробник приймає рішення щодо необхідності вміщення в нову модель ЖЦ засобів забезпечення якості компонентів, системи керування проектом або визначення набору процедур перевірки для забезпечення правильності продукту і відповідності його заданим вимогам.

Процеси, що включені в модель ЖЦ, призначені для реалізації стандартних задач процесів ЖЦ, вони можуть залучати інші процеси, що пов'язані із забезпеченням захисту даних. Інтерфейси (входи і виходи) будь-яких двох процесів ЖЦ повинні бути мінімальними і кожний з них повинен відповідати таким правилам:

– якщо процес А викликається процесом В і тільки процесом В, то А належить В;

– якщо функція викликається більше ніж одним процесом, то вона стає окремим процесом;

– перевірка будь-якої функції в ЖЦ є обов'язковою.

Іншими словами, якщо вирішення певної задачі потребує більше ніж один процес, то вона може сама набути статусу процесу, що використовується одно- або багаторазово протягом ЖЦ конкретної системи. Кожен процес повинен мати внутрішню структуру, встановлену відповідно до особливостей його виконання.

Процеси конкретної моделі ЖЦ орієнтовані безпосередньо на розробника даної системи. Розробник може виконувати один або кілька процесів і процес, у свою чергу, може бути виконаний одним або кількома розробниками. При цьому один з розробників має відповідати за один процес або за всі процеси моделі, навіть якщо окремі роботи виконує інший розробник.

Створювана модель ЖЦ узгоджується з конкретними методиками розробки систем і відповідними стандартами в області програмної інженерії, які існують або розробляються самостійно для проекту з урахуванням можливостей і особливостей ПС. Іншими словами, кожен процес ЖЦ підкріплюється вибраними для реалізації ПС засобами і методами програмування, а також методикою їх застосування і виконання.

При формуванні моделі ЖЦ важливу роль відіграють організаційні аспекти:

– структура колективу і зв’язків між ними;

– планування послідовності робіт і термінів їх виконання;

– підбір і підготовка ресурсів (людських, програмних і технічних) для виконання робіт;

– оцінка можливостей реалізації проекту в заданий термін, вартість і ресурси.

Впровадження моделі ЖЦ у практичну діяльність зі створення програмного продукту дозволить впорядкувати взаємини між суб'єктами процесу розроблення ПС і враховувати динамічні модифікації вимог до проекту і до системи.

Приклад. ЖЦ розробки ПС із задачами і діями процесу тестування. Головне призначення процесу тестування ЖЦ – виконання задач процесу на основі входів (вхідні дані для виконання задач процесу) і виходів при завершенні задач, а також ролей і дій виконавців цих задач.

Відповідно до стандарту ISO/IEC 12207 задачі тестування розглянуті і розподілені за процесами ЖЦ ПС. Як результат, отримано єдиний безперервний процес тестування різних продуктів ПС, задачами якого є *підготовка*, *проведення* й *оцінювання* результатів тестування. Ці задачі розподілилися між 20 діями (кроками) процесу розроблення. Даний підхід до поглибленого тестування ПС доцільно застосовувати, наприклад, для систем реального часу.

На кроці *підготовки* здійснюється аналіз робочих продуктів процесу розроблення ПС (вхідних для даного кроку процесу тестування) для визначення

цілей, об'єктів, сценаріїв і ресурсів тестування, адекватних кроку тестування. Результати виконання кроків підготовки тестування повинні фіксуватися в планах тестування (рисунок 4.3).

Рисунок 4.3 – ЖЦ з конкретними задачами на підпроцесах тестування ПС

На кроці *виконання* здійснюється фіксація результатів виконання тестів, їх порівняння з очікуваними результатами, визначення поточного стану робочого продукту ПС і прийняття рішення про достатність тестування.

Кожен крок процесу *розроблення* складається з набору розв'язуваних задач, їх розподілу за процесами і підпроцесами ЖЦ. Кроки процесу й окремих задач можуть виконуватися циклічно для різних об'єктів ПС при їх тестуванні.

Опис семантики задач і кроків процесу тестування наведено в таблиці 4.2.

Таблиця 4.2 – Зміст задач процесу тестування

Крок процесу	**Задачі процесу тестування**
1. Створення групи тестування	1.1. Визначення учасників процесу тестування
	Розподіл обов'язків у групі і формування плану тестування
2. Аналіз ризику	2.1. Ідентифікація ризиків
	2.2. Упорядкування ризиків
	2.3. Розподіл ресурсів

Крок процесу	Задачі процесу тестування
3. Визначення цілей тестування	3.1. Ідентифікація цілей тестування
	3.2. Визначення критеріїв проходження тестів
	3.3. Упорядкування цілей тестування за оцінками ризику
4. Розроблення планів тестування	4.1. Розроблення плану тестування ПС
	4.2. Розроблення плану інтеграційного тестування
	4.3. Розроблення плану автономного тестування
	4.4. Розроблення плану комплексного тестування
5. Розроблення тестів	5.1. Проектування і розроблення тестів
	5.2. Підготовка тестових даних
	5.3. Перевірка тестових документів
6. Автономне й інтеграційне тестування	6.1. Автономне тестування модулів і аналіз результатів
	6.2. Інтеграційне тестування
	6.3. Повторне тестування після усунення дефектів
	6.4. Аналіз результатів інтеграційного тестування
7. Тестування ПС	7.1. Затвердження середовища і ресурсів тестування
	7.2. Тестування ПС
	7.3. Повторне тестування ПС після усунення дефектів
	7.4. Аналіз результатів завершення тестування ПС
	7.5. Тестування інсталяції ПС
8. Складання документа за тестуванням ПС і підготовка звіту	8.1. Збирання і аналіз даних про результати тестування
	8.2. Підготовка розв’язків і рекомендацій з використання ПС
	8.3. Підготовка кінцевого документа про результати тестування
	8.4. Перевірка рішень і підготовка документа звіту

Для підключення задач тестування до всіх процесів ЖЦ проводиться:

– розподіл обов'язків між учасниками процесу з урахуванням вимог щодо їх професійної підготовки;

– визначення стандартів на подання остаточних документів, метрик процесу, критеріїв початку і завершення задач і переходу до наступного кроку процесу;

– підбір методів тестування для вибраного класу ПС і перевірки правильності виконання задач тестування;

– розроблення спеціальних шаблонів для документування процесу тестування щодо кожного його кроку.

При завершенні тестування ПС з метою визначення часу тестування та вартості робіт враховуються результати і дані процесу розроблення, а також оформляється звітний документ на виготовлення ПС. Оцінювання ризику відмов проводиться на процесі підготовки тестування і на кроках аналізу, а оцінювання якості виконується після завершення тестування.

4.3. Типи моделей життєвого циклу

Розглянуті підходи щодо побудови різних видів моделей ЖЦ базуються на процесному підході до виконання програмних проектів. Вони використовувалися на практиці під час створення різних типів моделей ЖЦ, до яких належать такі моделі: каскадна, спіральна, інкрементна, еволюційна та ін.

4.3.1. Каскадна модель

Однією з перших почала застосовуватися каскадна модель, де кожна робота виконується один раз і в такому порядку, який подано на рисунку 4.4.

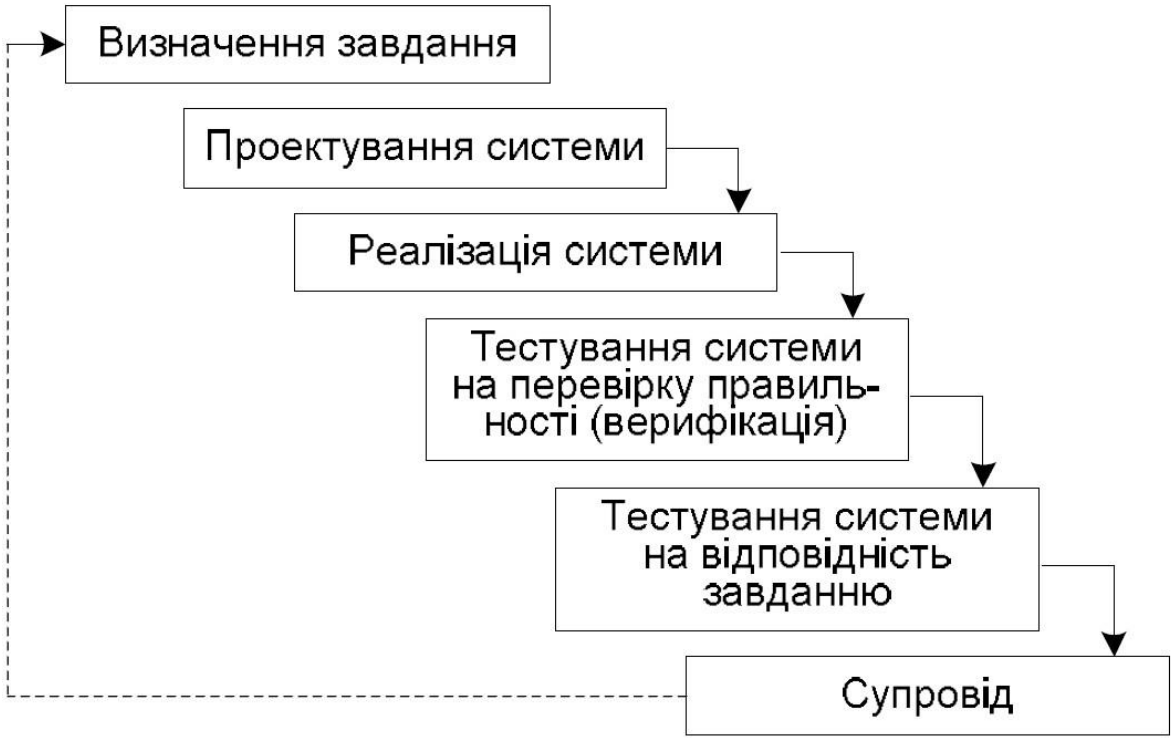

Рисунок 4.4 – Каскадна модель ЖЦ програмних систем

Тобто вважається, що кожна робота має бути виконана настільки ретельно, що після її закінчення і переходу до наступного етапу, повертатися до попереднього не буде потреби. Розробник перевіряє проміжний результат відомими методами верифікації і фіксує його як готовий еталон для наступного процесу.

Згідно з даною моделлю роботи і завдання процесу розроблення зазвичай виконуються послідовно, як це наведено у схемі.

Проте допоміжні і організаційні процеси (контроль вимог, керування якістю і ін.), як правило, виконуються разом з процесами розробки ПЗ. У даній моделі повернення до початкового процесу передбачається після супроводження і виправлення помилок.

Особливість такої моделі полягає у фіксації послідовних процесів розроблення програмного продукту. В її основу покладена модель фабрики, де продукт проходить стадії від задуму до виробництва, потім його передають

замовнику у вигляді готового виробу, де заміна не передбачена, хоча можна подати інший подібний виріб. Недоліки цієї моделі такі:

– процес створення ПС не завжди вкладається в таку жорстку форму і послідовність дій;

– не враховуються змінювані потреби користувачів, нестабільні умови зовнішнього середовища, які впливають на зміни вимог до ПС під час її розроблення;

– значний розрив між часом внесення помилки (наприклад, на процесі проектування) і часом її виявлення (при супроводі), що призводить до суттєвої переробки ПС.

При застосуванні каскадної моделі можуть спостерігатися такі чинники ризику:

– вимоги до ПС недостатньо чітко сформульовані, або не враховують перспективи розвитку ОС, середовищ і т.п.;

– громіздка система, що не допускає компонентної декомпозиції, може викликати проблеми щодо розміщення її в пам'яті або на платформах, непередбачених у вимогах;

– внесення швидких змін до технології і у вимоги може погіршити процес розроблення окремих частин системи або системи в цілому;

– обмеження на ресурси (людські, програмні, технічні і ін.) в ході розробки можуть звузити окремі можливості реалізації системи;

– отриманий продукт може виявитися не придатним для застосування внаслідок нерозуміння розробниками вимог або функцій системи або недостатньо проведеного тестування.

Переваги реалізації системи за допомогою каскадної моделі такі:

– всі завдання підсистем і системи реалізуються одночасно, завдяки чому неможна забути жодного завдання, а це сприяє встановленню стабільних зв'язків між ними;

– повністю розроблену систему з документацією на неї легше супроводжувати, тестувати, фіксувати помилки і вносити зміни не хаотично, а цілеспрямовано, починаючи з вимог, наприклад, додавати або замінювати деякі функції і повторювати процес.

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

4.3.2. Інкрементна модель

Цю модель (incremental) ще називають моделлю з нарощуванням або з приростом. Її суть полягає в розробці продукту ітераціями, і кожна ітерація закінчується випуском працездатної версії. У кожній новій версії додаються деякі функціональні можливості. Розробка системи починається з визначення

набору всіх вимог до ПС і ділення процесу розроблення на ітерації. Кожна ітерація реалізується послідовно з використанням процесів ЖЦ і фіксації робочої версії системи, системи, що поступово наближається до остаточної версії (рисунок 4.5).

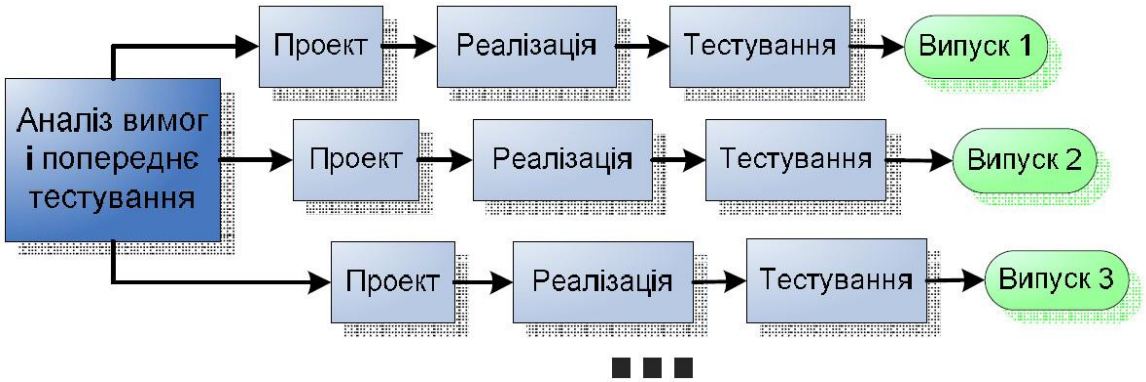

Рисунок 4.5 – Інкрементна модель ЖЦ

Перша проміжна версія системи, що створюється (випуск 1), реалізує частину вимог, у подальшу версію (випуск 2) додають додаткові вимоги до тих пір, поки не будуть остаточно виконані всі вимоги і розв’язані задачі розробки системи.

Для кожної проміжної версії на процесах ЖЦ виконуються необхідні процеси, роботи і завдання, зокрема, аналіз вимог і створення нової архітектури, які можуть бути виконані одночасно.

Процеси розроблення технічного проекту програмної системи, її програмування і тестування, збирання і кваліфікаційні випробування ПС виконуються при створенні кожної подальшої версії.

Відповідно до даної моделі ЖЦ, процеси якої практично такі самі, як і в каскадній моделі, наголос робиться на побудову деякої закінченої проміжної версії, а завдання процесу розроблення виконуються послідовно або частково паралельно для ряду окремих проміжних структур версії.

Роботи і завдання процесу розроблення наступної версії системи з додатковими вимогами або функціями можуть виконуватися неодноразово в одній тій же послідовності для всіх проміжних версій системи. Процеси супроводження і експлуатації можуть бути реалізовані разом з процесом розроблення версії шляхом перевірки частково реалізованих вимог в кожній проміжній версії і так до отримання кінцевого варіанта системи. Допоміжні і організаційні процеси ЖЦ зазвичай виконуються разом з процесом розроблення версії системи і до кінця розробки збираються дані, на підставі яких можна встановити рівень завершеності і якості виготовленої системи.

При застосуванні даної моделі необхідно враховувати такі чинники ризику:

– вимоги складені з урахуванням можливості їх зміни при реалізації продукту;

– всі можливості системи потрібно реалізувати одразу;

– швидка зміна технології і вимог до системи може призвести до порушення отриманої структури системи;

– обмеження в ресурсному забезпеченні (виконавці, фінанси) можуть призвести до несвоєчасного введення системи в експлуатацію.

Дану модель ЖЦ доцільно використовувати, у випадках, коли:

– бажано реалізувати деякі можливості системи швидко за рахунок створення проміжної версії продукту;

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

– можливе збільшення фінансування на розробку окремих частин системи.

4.3.3. Спіральна модель

Виходячи з можливості внесення змін, як в процес, так і в проміжний продукт було створено спіральну модель ЖЦ (рисунок 4.6).

Внесення змін орієнтоване на задоволення потреби користувачів одразу, як тільки буде встановлено, що створені артефакти або елементи документації не відповідають дійсному стану розробки.

Дана модель ЖЦ допускає аналіз продукту на витку розробки, його перевірку, оцінку правильності та прийняття рішення про перехід на наступний виток або повернення на попередній виток для доопрацювання на ньому проміжного продукту.

Відмінність цієї моделі від каскадної полягає в можливості багато разів повертатися до процесу формулювання вимог і до повторної розробки версії системи з будь-якого процесу моделі.

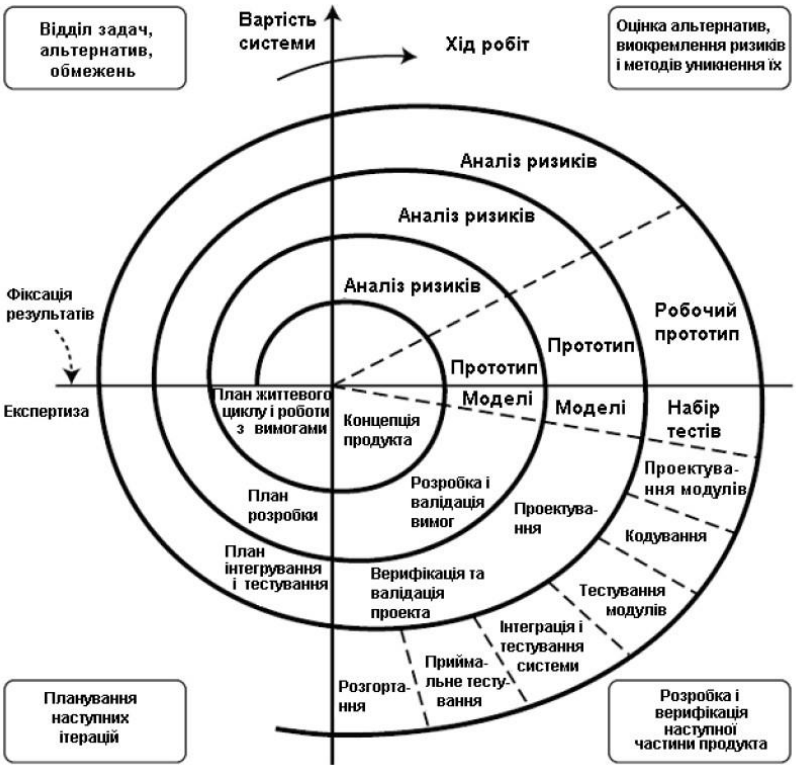

Рисунок 4.6 – Спіральна модель ЖЦ розробки програмних систем

Для програмного продукту така модель не дуже підходить з декількох причин. По-перше, висловлення вимог замовником носить суб'єктивний характер, вимоги можуть багаторазово уточнюватися протягом розробки ПС і навіть після завершення та випробовування, і часом може з'ясуватися, що замовник «хотів зовсім інше». По-друге, змінюються обставини та умови використання системи, тому загальновизнаним законом програмної інженерії є закон еволюції, який сформулюємо так: кожна діюча ПС з часом потребує внесення змін або виводиться з експлуатації.

При необхідності внесення змін до системи на кожному витку з метою отримання нової версії системи обов'язково вносяться зміни в заздалегідь зафіксовані вимоги, після чого повертаються на попередній виток спіралі для продовження реалізації нової версії системи з урахуванням усіх змін.

4.3.4. Еволюційна модель

У разі еволюційної моделі система послідовно розробляється з блоків конструкцій. На відміну від інкрементної моделі в еволюційній моделі вимоги встановлюються частково і уточнюються в кожному наступному проміжному блоці структури системи.

Використання еволюційної моделі припускає проведення дослідження предметної області для вивчення потреб її замовника і аналізу можливості застосування цієї моделі для реалізації. Модель використовується для розробки нескладних і некритичних систем, де головною вимогою є реалізація функцій системи. При цьому вимоги не можуть бути визначені відразу і повністю. Тому розробка системи здійснюється ітераційним шляхом її еволюційного розвитку з отриманням деякого варіанта системи–прототипу, на якому перевіряється реалізація вимог. Іншими словами, такий процес за своєю суттю є ітераційним, з етапами розробки, що повторюються, починаючи від змінених вимог і закінчуючи отриманням готового продукту. В деякому розумінні до цього типу моделі можна віднести спіральну модель.

Розвитком цієї моделі є модель еволюційного прототипування в рамках усього ЖЦ розробки ПС (рисунок 4.7).

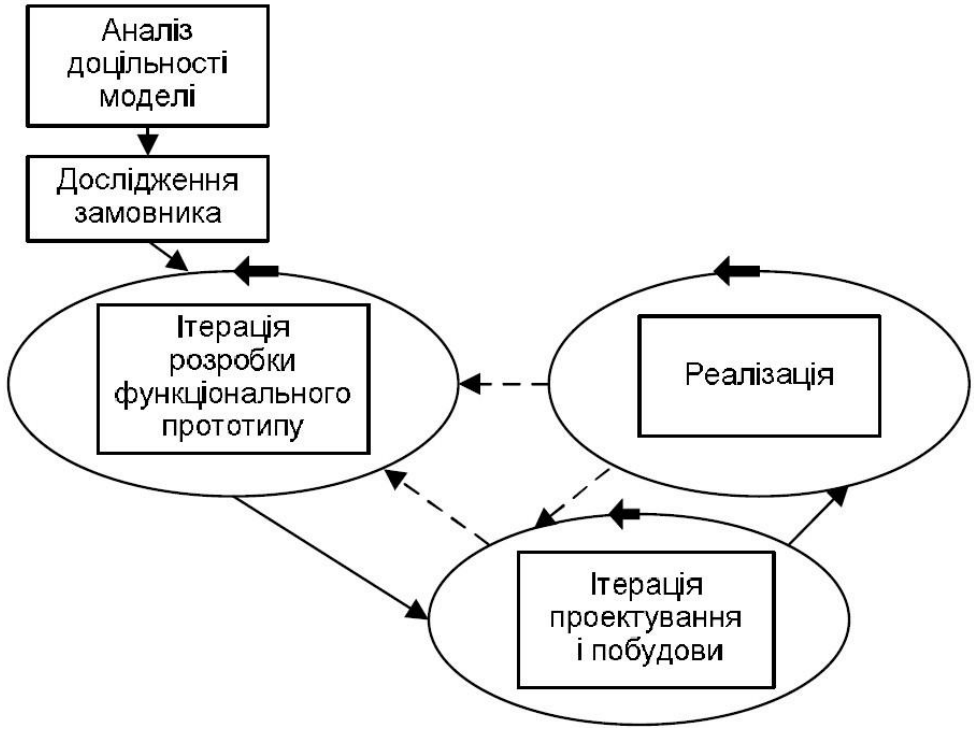

Рисунок 4.7 – Модель еволюційного прототипування

У літературі вона часто називається моделлю швидкої розробки програм RAD (Rapid Application Development).

У даній моделі наведені дії, які пов'язані з аналізом її застосовності для конкретного виду системи, а також обстеженням замовника для визначення потреб користувача при розробці плану створення прототипу.

У моделі є дві головні ітерації розробки функціонального прототипу, проектування і реалізації системи з метою перевірки, чи задовольняє вона всі функціональні і нефункціональні вимоги. Основною ідеєю цієї моделі є моделювання окремих функцій системи в прототипі і поступове еволюційне його доопрацювання до виконання всіх заданих функціональних вимог.

3Ітерацій з отримання проміжних варіантів прототипу може бути декілька, в кожній з яких додається функція і повторно моделюється робота прототипу. І так до тих пір, поки не будуть промодельовані всі функції, задані у вимогах до системи. Після цього виконується ще одна ітерація – остаточне програмування для отримання готової системи.

Ця модель застосовується для систем, в яких найбільш важливими є функціональні можливості, і які необхідно швидко продемонструвати на CASE-засобах.

Оскільки проміжні прототипи системи відповідають реалізації деяких функціональних вимог, їх можна перевіряти і під час супроводу і експлуатації, тобто разом з процесом розробки чергових прототипів системи. При цьому допоміжні і організаційні процеси можуть виконуватися разом з процесом розроблення і накопичувати відомості за даними кількісних і якісних оцінок на процесах розроблення.

При цьому враховуються такі чинники ризику:

– реалізація всіх функцій системи одночасно може призвести до громіздкості;

– обмежені людські ресурси зайняті розробкою протягом тривалого часу.

Переваги застосування даної моделі ЖЦ такі:

– швидка реалізація деяких функціональних можливостей системи і їх апробація;

– використання проміжного продукту в наступному прототипі;

– виділення окремих функціональних частин для реалізації їх у вигляді прототипу;

– можливість збільшення фінансування системи;

– зворотний зв'язок із замовником для уточнення функціональних вимог;

– спрощення внесення змін у зв'язку із заміною окремих функції.

Модель розвивається у напряму додавання нефункціональних вимог до системи, пов'язаних із захистом і безпекою даних, несанкціонованим доступом даних і ін.

Список літератури до Лекції 4

1. *ISO/IEC 12207*: 1995–0801: Informational Technology – Software life cycle processes. // ГОСТ Р ИСО/МЭК 12207–99. Информационная технология. Процессы жизненного цикла программных средств.

2. *Лаврищева Е.М.*, *Грищенко В.Н.* Области знаний программной инженерии –SWEBOK и подход к обучению этой дисциплины// Управляющие системы и машины.–2005. – №1.– С.38–54.

3. *Васютович В.*, *Самотохин С., Никифоров Г.* Регламентация жизненного цикла программных средств

4. *Вендеров А.М.* Проектирование программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2000. – 347 с.

5. *Орлов С.А.* Технологии разработки программного обеспечения. – Спб.: Питер, 2002. – 463 с.

6. *Соммервил И.* Инженерия программного обеспечения. 6-е издание. – М.– Спб.–Киев,– 2002. –623 с.

7. *Лаврищева Е.М.*, *Коротун Т.М.* Построение процесса тестирования программных систем // Проблемы программирования. – 2002. – № 1–2. – С. 272–281.

8. *Андон Ф.И.*, *Коваль Г.И., Коротун Т.М., Лаврищева Е.М., Суслов В.Ю.* Основы инженерии качества программных систем. – Киев: Академпериодика, 2007.– 680 с.

Лекція 5. Agile розробка ПС

Ітеративна розробка – це технічний підхід до створення програмних систем, покладений в основу опису об'єктно-орієнтованого аналізу та проектування. У рамках цього підходу розроблення виконується у вигляді декількох короткострокових міні-проектів фіксованої тривалості, що називаються *ітераціями*. Кожна ітерація містить свої власні етапи аналізу вимог, проектування, реалізації та завершується тестуванням, інтеграцією та створенням робочої системи. Перші ідеї ітеративного процесу називалися «проектування по спіралі й еволюційною розробкою». Переваги ітеративного розроблення:– своєчасне усвідомлення можливих технічних ризиків, осмислення вимог, завдань проекту та зручності використання системи;– швидкий та помітний прогрес;– ранній зворотний зв'язок, можливість обліку побажань користувачів і адаптації системи. У результаті система більш задовольняє реальні вимоги керівників і споживачів;– керована складність;– отриманий при реалізації кожної ітерації досвід можна методично використовувати для поліпшення самого процесу розроблення.

Agile – гнучка методологія розроблення, – це концептуальний каркас, в рамках якого виконуються проекти з розроблення програмного забезпечення. Основна ідея всіх гнучких моделей полягає в тому, що процес, який застосовується у розробці ПЗ, повинен бути адаптивним. Вони декларують своєю вищою цінністю орієнтованість на людей та їх взаємодію, а не на процеси і засоби.

5.1 Модель керування програмними проектами SCRUM

Методологія *Scrum* була вперше озвучена в роботі Хіротака Такеучи й Ікуджіро Нонаки, опублікованій в *Harvard Business Review*. Джеф Сазерленд використовував цю роботу при створенні методології для корпорації *Easel* у 1993 році, яку за аналогією і назвав *Scrum*, а Кен Швабер формалізував процес для використання в індустрії розроблення програмного забезпечення. Мета цієї методології – виявити й усунути відхилення від бажаного результату на більш ранніх етапах розроблення програмного продукту. Методологія *Scrum* визначає:

1. Правила, за якими повинен плануватися і управлятися список вимог до продукту, з метою досягнення максимальної прибутковості від реалізованої функціональності;
2. Правила планування ітерацій для забезпечення максимальної зацікавленості команди в отриманні результату;
3. Основні правила взаємодії учасників команди для максимально швидкої реакції на непередбачені робочі ситуації;
4. Правила аналізу і коригування процесу розроблення для вдосконалення взаємодії всередині команди.

Кожну ітерацію можна описати так: «Плануємо – Фіксуємо– Реалізуємо – Аналізуємо». За рахунок фіксування вимог на протязі однієї ітерації та зміни

довжини ітерації можна керувати балансом між гнучкістю та плановим керуванням розроблення. *Scrum* фокусується на постійному визначенні пріоритетних завдань, ґрунтуючись на бізнес цілях, що збільшує корисність і прибутковість проекту на його ранніх стадіях. Зважаючи на те, що при ініціації проекту його прибутковість визначити майже неможливо, *Scrum* пропонує концентруватися на якості розроблення і до кінця кожної ітерації мати проміжний продукт, який можна використовувати. Наприклад, результатом ітерації може бути каркас сайту, який можна показати на презентації. Методологія *Scrum* орієнтована на те, щоб оперативно пристосовуватися до змін у вимогах, що дозволяє команді швидко адаптувати продукт до потреб замовника. Девіз *Scrum* – «аналізуй та адаптуй»: аналізуйте те, що отримали, адаптуйте те, що вже розроблено до реальних потреб, а потім аналізуйте знову. Чим менше формалізму, тим більш гнучко та ефективно можна працювати, – це основний принцип даної методології. Але це не означає, що формальних процесів не повинно бути зовсім, їх має бути достатньо для організації ефективної взаємодії і керування проектом. Формальна частина *Scrum* складається з трьох ролей, трьох практик і трьох основних документів. В *Scrum*-проектах відділяють наступні ролі:

1. **Власник продукту** (*Product Owner*) – людина, що формулює вимоги програмістам. Зазвичай власник продукту є представником або довіреною особою замовника, а для компаній, що випускають коробкові продукти, він являє собою ринок, на якому реалізується продукт. Власник продукту повинен скласти бізнес план, який показує очікувану дохідність і план розвитку до вимог, відсортованими за коефіцієнтом окупності інвестицій. Виходячи з наявної інформації, власник продукту готує перелік вимог, відсортований за значимістю;

2. *Scrum*-**майстер** (*Scrum Master*) – забезпечує максимальну працездатність і продуктивність команди, чітку взаємодію між усіма учасниками проекту, своєчасне вирішення всіх проблем, що гальмують або зупиняють роботу будь-якого члена команди, відгороджує ко проходження процесу всіх учасників проекту. Ця людина має бути одним з членів команди розроблення і брати участь у проекті як розробник. Він відповідає за своєчасне вирішення поточних проблем.

3. *Scrum*-**команда** (*Scrum Team*) – група, що складається з п’яти-дев’яти самостійних, ініціативних програмістів. Перше завдання цієї команди – поставити реально досяжну, прогнозовану, цікаву і значущу мету для ітерації. Друге завдання – зробити все для того, щоб ця мета була досягнута у відведені терміни і з заявленою якістю. Мета ітерації вважається досягнутою тільки в тому випадку, якщо всі поставлені завдання реалізовані, весь код написаний за певними проектом «стандартам кодування» (*coding guidelines*), програма протестована повністю, а всі знайдені дефекти усунені. Програмісти цієї команди повинні вміти оцінювати та планувати свою роботу, працювати в команді, постійно аналізувати та поліпшувати якість взаємодії та роботи. В обов’язки всіх членів *Scrum*-команди входить участь у виборі мети ітерації і визначення результату роботи. Вони повинні робити все можливе і неможливе

для досягнення мети ітерації в рамках, визначених проектом, ефективно взаємодіяти з усіма учасниками команди, самостійно організовувати свою роботу, надавати власнику робочий продукт в кінці кожного циклу. *Scrum* містить наступні документи: **журнал продукту** (*Product Backlog*), **журнал спринту** (*SprintBacklog*) та **графік спринту** (*Burndown Chart*).

На початку проекту власник продукту готує **журнал продукту** (табл. 5.1) – перелік вимог, відсортований за значимістю, а *Scrum*-команда доповнює цей журнал оцінками вартості реалізації вимог. Перелік повинен містити функціональні та технічні вимоги, необхідні для реалізації продукту. Найбільш пріоритетні з них повинні бути досить детально прописані, щоб їх можна було оцінити і протестувати. Про своєчасну деталізацію вимог має дбати власник продукту і надавати необхідний обсяг у потрібний час. У цьому сенсі програмісти є замовниками вимог для власника продукту. Надалі інші вимоги повинні поступово уточнюватися та деталізуватися до такого ж рівня. Головне, щоб у команди завжди був достатній обсяг підготовлених до реалізації вимог. Графік спринту дозволяє також власнику продукту спостерігати за ходом ітерації – якщо загальний обсяг робіт не зменшується щодня, значить, щось йде не так. Під час сесії планування команда знаходить і оцінює завдання, які треба виконати для успішного завершення ітерації. Сума оцінок всіх завдань в журналі спринту є загальним обсягом роботи, який треба виконати за ітерацію. Піс ля завершення кожного завдання *Scrum*-майстер перераховує обсяг роботи, що залишилася, і зазначає це на графіку спринту. Тільки в тому випадку, якщо по закінченні ітерації у журналі спринту не залишилося незавершених завдань, ітерація вважається успішною. Графік спринту використовується як допоміжний інструмент, що дозволяє коригувати роботу для завершення ітерації вчасно, з працюючим кодом і необхідною якістю.

Таблиця 5.1 – Приклад журналу продукту

Номер вимоги	**Опис вимоги**	**Цінність бізнесу (ум. од.)**	**Пріоритет**	**Високорівнева оцінка(часи)**
FE1	Покупець може зареєструватися на сайті	10.000	1	20
FE2	Покупець може ввести свої персональні дані	12.000	6	36
FE3	Покупець може побачити список доступних виробів	16.000	10	30
FE4	Покупець може купити виріб	100.000	20	48

FE5	Покупець може робити пошук виробів	80.000	30	32
FE6	Покупець може підписатись на новини	30.000	40	66К

До *Scrum*-**практик** відносяться: спринт (*Sprint*), скрам (*Daily Scrum Meeting*) та демонстраційне засідання (*Sprint Review Meeting*).

1. Підготовка до першої ітерації, так званого **спринту** (*Sprint*), починається після того, як власник продукту розробив **журнал продукту**. При плануванні ітерації відбувається детальне розроблення сесій планування спринту (*Sprint Planning Meeting*), яке починається з того, що власник продукту, *Scrum*-команда і *Scrum*-майстер перевіряють план розвитку продукту, план релізу та перелік вимог, *Scrum*-команда перевіряє оцінки вимог, переконується, що вони досить точні, щоб почати працювати, вирішує, який обсяг роботи вона може успішно виконати за спринт, ґрунтуючись на розмірі команди, доступному часі та продуктивності. *Scrum*-команда повинна вибирати перші за пріоритетом вимоги з журналу продукту. Після того як *Scrum*-команда зобов'язується реалізувати обрані вимоги, *Scrum*-майстер починає планування спринту. *Scrum*-команда розбиває вибрані вимоги на завдання, необхідні для його реалізації. Ця активність в ідеалі не повинна займати більше чотирьох годин, і її результатом є **журнал спринту**. Необхідно, щоб всі учасники команди взяли на себе зобов'язання з реалізації обраної мети.

2. Після закінчення планування починається ітерація. Кожен день *Scrum*-майстер проводить «**скрам**» (*Daily Scrum Meeting*) – п'ятнадцятихвилинну нараду, мета якої – досягти розуміння того, що сталося з часу попередньої наради, скоригувати робочий план до реалій сьогодення і позначити шляхи вирішення існуючих проблем. Кожен учасник *Scrum*-команди відповідає на три питання: що я зробив з часу попереднього скраму, що мене гальмує або зупиняє в роботі, що я буду робити до наступного скрама? У цій нараді може брати участь будь-яка зацікавлена особа, але тільки учасники *Scrum*-команди мають право приймати рішення. Правило обґрунтовано тим, що вони давали зобов'язання реалізувати мету ітерації, і тільки це дає впевненість у тому, що вона буде досягнута. На них лежить відповідальність за їх власні слова, і, якщо хтось з боку втручається і приймає рішення за них, тим самим він знімає відповідальність за результат з учасників команди. В кінці кожного спринту проводиться демонстраційне засідання (Sprint Review Meeting) тривалістю не більше чотирьох годин. Спочатку Scrum-команда демонструє власнику продукту зроблену протягом спринту роботу, а той у свою чергу веде цю частину наради і може запросити до участі всіх зацікавлених осіб. Власник продукту визначає, які вимоги з журналу спринту були виконані, і обговорює з командою і замовниками, як краще розставити пріоритети в журналі продукту для наступної ітерації. У другій частині мітингу проводиться аналіз минулого спринту, який

веде Scrum-майстер. Scrum-команда шукає використані в останньому спринті позитивні і негативні способи спільної роботи, аналізує їх, робить висновки і приймає важливі для подальшої роботи рішення. Scrum-команда також визначає програми, які можуть працювати краще, і шукає шляхи для збільшення ефективності подальшої роботи. Потім цикл замикається, і починається планування наступного спринту. Час між ітераціями – це час прийняття основоположних рішень, що впливають на хід всього проекту. Під час спринту ніякі зміни ззовні не можуть бути зроблені. Після того як команда дала зобов'язання реалізувати журнал спринту, він фіксується, і зміни в ньому можуть бути зроблені тільки з таких причин:

• Scrum-команда протягом ітерації отримала кращу уяву про вимоги і потребує додаткових завдань для успішного завершення ітерації;

• знайдені дефекти, які треба обов'язково виправити для успішного завершення ітерації;

• Scrum-майстер і Scrum-команда можуть вирішити, що невеликі зміни, які не впливають на загальний обсяг робіт, можуть бути реалізовані в зв'язку з виниклою у власника продукту необхідністю.

Виходячи з того, що журнал спринту не може бути змінений ззовні під час ітерації, потрібно вибирати його довжину, ґрунтуючись на стабільності вимог. Якщо вимоги стабільні, змінюються або доповнюються рідко, то можна вибрати шеститижневий цикл. У цьому випадку заощаджується час на перемикання команди з активного розроблення на планування та демонстраційні мітинги. Якщо вимоги часто змінюються і доповнюються, потрібно відштовхуватися від двотижневого циклу, в будь-якому випадку довжина ітерації – це величина експериментальна.

Недоліки *Scrum*:

1. Складно домогтися активної участі від кожного розробника і злагодженої колективної роботи в команді.

2. Складно залучити постачальника вимог до активної участі в проекті, зацікавити його динамікою розвитку продукту, дати можливість бути активним вболівальником і спонсором команди. Проте, незважаючи на це, використання методології *Scrum* в проектах дозволяє:

– використовувати весь технічний багаж, накопичений компанією, тому що головний напрям методології *Scrum* направлений на керування проектами і не задає ніяких технічних практик;

– з високою якістю та в рамках бюджету реалізувати великий обсяг функціональності та специфікації, які були відсутні на момент початку проекту;

– забезпечити максимальну бізнес-цінність виробленого продукту, за рахунок того, що практично всі реалізовані функції активно використовуються відвідувачами;– досягнути високу супровідність коду (можливість внесення змін з мінімальними трудовитратами) – вартість змін, внесених до продукту, практично еквівалентна вартості розроблення аналогічних функцій продукту на початку проекту, що рідко буває у RUP чи MSF моделях виробництва, для яких характерний експонентний ріст вартості змін по мірі виконання проекту.

5.2 Модель керування програмними проектами Kanban

З початку 70-х років у Японії, а потім і в інших країнах набула великого поширення система «*Канбан*», що є механізмом організації безперервного гнучкого виробничого потоку і функціонує практично без страхових запасів. Традиційна концепція організації виробництва, як відомо, спрямована на запобігання простоям та організацію безперервного потоку з обов'язковим створенням страхового запасу. Японська концепція ґрунтується на практично повній відмові від страхових запасів. Більше того, менеджери навмисне дають робітникам змогу повністю випробувати на собі наслідки простоїв. В результаті весь персонал постійно зайнятий виявленням причин збоїв у виробництві та пошуком шляхів підвищення надійності та запасу міцності системи управління. Після виявлення та усунення причин простоїв керівники ще більше скорочують страховий запас, породжуючи додаткові зусилля з покращення організації виробництва з боку всього персоналу. В умовах системи «*Канбан*», на відміну від традиційного підходу, виробник не має завершеного плану й графіка виробництва, а жорстко пов'язаний конкретним замовленням споживача. Він не взагалі оптимізує свою роботу, а в межах замовлення. Конкретного графіка роботи на декаду, місяць він не має. Кожен зайнятий у технологічному ланцюгу робітник знає, що він вироблятиме продукцію тільки тоді, коли карта «*Канбан*» з його продукції відкріплена від контейнера на складі, тобто коли продукція фактично надійшла на наступну стадію обробки. Конкретний графік послідовності праці одержують лінії кінцевого складання, вони розкручують клубок інформації у зворотному напрямі. Іншими словами, графіки виробництва не переглядаються, а тільки формуються рухом карток «*Канбан*». Виробництво постійно перебуває в стані надбудови, відбувається його системне коригування під зміни ринкової кон'юнктури. Методика розроблення програмного забезпечення ***Канбан*** була введена у практичне використання Девідом Андерсеном у 2007 році. Багато з цих практик та підходів використовувалися різними *Agile* командами, перш ніж були описані як єдине ціле. Нововведення полягали в тому, що було введено завдання «в процесі». Це робилося і раніше іншими *Agile*-командами, але в Канбан існує всім відоме обмеження на кількість робочих завдань, які можуть виконуватися в один час. Ця межа зазвичай досить низька – ліміт приблизно дорівнює числу розробників в команді або трохи менший. Основні положення Канбан наступні:

1. Візуалізація потоку робіт:

– розбиття роботи на частини, кожна частина виписується на картку і прикріплюється до стіни;

– розбиття усіх завдань на стовпчики для розділення по стадіям розробки.

2. Обмеження незавершеної роботи (НЗР) (*work*-in progress) визначається можлива кількість незавершених пунктів на кожній стадії робочого процесу.

3. Вимірювання часу виконання завдання (*lead* time) (середньої тривалості часу для завершення одного пункту, іноді звану «оперативним часом» (*cycle*

time)), оптимізація процесу, щоб звести час виконання завдання до мінімуму і зробити його настільки прогнозованим, наскільки це можливо.

Канбан – це не конкретний процес, а система цінностей. Його можна описати однією простою фразою – «Зменшення виконується в певний момент роботи (*work in progress*)».

Різниця між *Канбан* і Scrum:

– немає таймбоксів (ні на завдання, ні на спринти);

– завдання більші і їх менше;

– оцінки термінів на задачу опціональні або взагалі їх немає;

– «швидкість роботи команди» відсутня і враховується тільки середній час на повну реалізацію завдання.

Команда для роботи використовує дошку. Наприклад, вона може виглядати класично, як на рисунку 5.1., де дошка розділена на 3 складові.

Наприклад, вона може мати і більше стовпців (стовпці зліва направо):

Цілі проекту. Сюди можна помістити високорівневі цілі проекту.

Черга завдань. Тут зберігаються завдання, які готові до того, щоб почати їх виконувати. Завжди для виконання дошки береться зверху, сама пріоритетна задача і її картка переміщується в наступний стовпець.

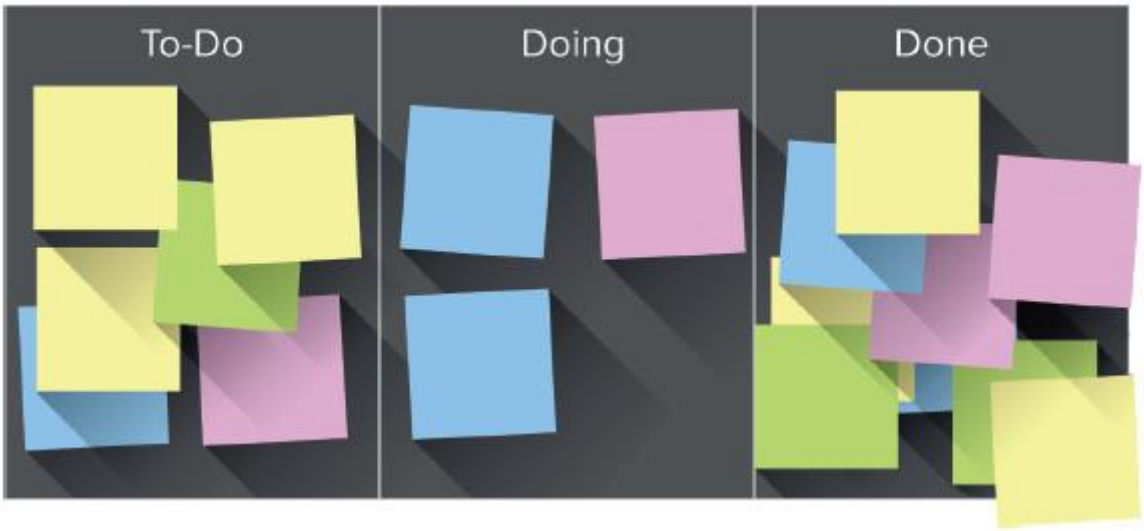

Рисунок 5.1 – Приклад Канбан

Опрацювання дизайну. Цей та інші стовпці до «Закінчено» можуть змінюватися, тому що саме команда вирішує, які кроки проходить завдання до стану «Закінчено». Наприклад, в цьому стовпці можуть перебувати завдання, для яких дизайн коду або інтерфейсу ще не ясний і обговорюється. Коли обговорення закінчені, завдання пересувається в наступний стовпець.

Розробка. Тут завдання перебуває до тих пір, поки розробка не завершена. Після завершення вона пересувається в наступний стовпець. Або, якщо архітектура не вірна або не точна – завдання можна повернути в попередній стовпець.

Тестування. У цьому стовпці завдання знаходиться, поки воно тестується. Якщо знайдені помилки – повертається в Розробку. Якщо ні – пересувається далі.

Деплоймент. У всіх проектів різне поняття деплоймент, наприклад, викласти нову версію продукту на сервер або помістити код в репозиторій.

Закінчено. Сюди стікер потрапляє лише тоді, коли всі роботи по завданню закінчені повністю.

В будь-якій роботі трапляються термінові завдання. Для таких можна виділити спеціальне місце (наприклад «Прискорити»). В Прискорити можна помістити один рядок до завдання і команда повинна почати її виконувати негайно і завершити якомога швидше. У такій черзі може бути тільки одна така задача, якщо з'являється ще одна – вона повинна бути додана в «Черга завдань». Під кожним стовпцем вказуються цифри, які позначають кількість завдань, які можуть бути одночасно в цих стовпцях. Цифри підбираються експериментально, але вважається, що вони повинні залежати від числа розробників в команді. Наприклад, якщо є 8 програмістів в команді, то в рядок «Розробка» можна помістити цифру 4. Це означає, що одночасно програмісти будуть робити не більше 4-х завдань, а значить у них буде багато причин для спілкування та обміну досвідом. Якщо поставити туди цифру 2, то 8 програмістів, що займаються двома завданнями, можуть простоювати або втрачати занадто багато часу на обговореннях. Якщо поставити 8, то кожен буде займатися своїм завданням і деякі завдання будуть затримуватися на дошці надовго, а головне завдання Канбан – це зменшення часу проходження завдання від початку до стадії готовності.

Підсумовуючи, переваги *Канбан* полягають у тому, що:

1. Легко визначаються проблемні області.

2. Зростає продуктивність роботи за рахунок того, що члени команди техпідтримки самостійно організують свою роботу, використовуючи канбан-дошку, а менеджер лише спрямовує зусилля на приорітезацію великих проектів і вирішення виникаючих проблем.

Недоліки *Канбан*:

1. Зі зменшенням НЗР з'являються обмеження – для менш пріоритетних проектів не вистачає ресурсів, що призводить до скорочення кількості проектів на команду.

2. Канбан накладає менше обмежень, ніж *Scrum*, тобто керівництво отримує більше параметрів для налаштування. Це може бути як недоліком, так і перевагою, залежно від ситуації.

3. Канбан – це ще більш «гнучка» методологія, ніж *Scrum* і *XP*, тому вона не підійде командам та проектам, не готовим до гнучкої роботи.

Список літератури до Лекції 5

1. *Брауде Э*. Технология разработки программного обеспечения [Текст] / Брауде Э. – СПб.: Питер, 2004. – 655с. ил.

2. *Пышкин Е. В*. Основные концепции и механизмы объектно-ориентированного программирования [Текст] / Пышкин Е. В. – СПб.: БХВ-Петербург, 2005. – 640 с.: ил.

3. *Соммервилл И.* Инженерия программного обеспечения; пер. с англ. [Текст] / Соммервилл И. – М. : Изд. дом «Вильямс», 2002. – 624 с.

4. *Константайн Л.* Разработка программного обеспечения; пер. с англ. [Текст] / Л. Константайн, Л. Локвуд.– СПб.: Питер, 2004. – 592 с.

5. *Коплиен Дж.* Мультипарадигменное проектирование для C++ [Текст] / Коплиен Дж. – СПб. : Питер, 2005. – 235 с.

6. *Bahrami, A.* Object Oriented Systems Developoment? Irwin McGrawHill, 1999. – 412 pp.

7. Object Management Group: [Електрон. ресурс]. – Режим доступу: http://www.omg.org/.

8. *Буч Г.* Язык UML. Руководство пользователя [Текст] / Г. Буч, Д. Рамбо, А. Джекобсон. – М.: ДМК Пресс, 2001. – 257 с.

9. *Грэхем И.* Объектно-ориентированные методы. Принципы и практика; пер. с англ. [Текст] / Грэхем И. – М.: Издательский дом «Вильямс», 2004. – 880 с.

10. *Рамбо Дж.* UML: специальный справочник. [Текст] / Дж. Рамбо, А. Якобсон, Г. Буч. – СПб.: Питер, 2002. – 656 с.

11. *Элиенс А.* Принципы объектно-ориентированной разработки программ; пер. с англ. [Текст] /Элиенс А. – М.: Издательский дом «Вильямс», 2002. – 496 с.

12. *Брагина, Т.И.* Сравнительный анализ итеративных моделей разработки программного обеспечения [Текст] / Т.И. Брагина, Г.В. Табунщик // Радиоелектроніка. Інформатика. Управління. – 2010. – № 2. – С. 130 – 139.

13. *Брагина, Т.И.* Анализ подходов к управлению рисками в программных проектах с итеративным жизненным циклом [Текст] / Т.И. Брагина, Г.В. Табунщик // Радіоелектроніка. Інформатика. Управління. – 2011. – No2 – С. 120-124.

14. *Брагіна, Т.І.* Розробка засобів інтеграції / Т.І. Брагіна // Системи обробки інформації. Вип.8 (106). – 2012. – С. 127-130.

15. *Брагина, Т.И.* Оценка прогресса разработки программных проектов в JIRA / Т.И. Брагина, К.В. Задорожная // Тиждень науки – 2013: зб. тез доп. щоріч. наук.-практ. конф. викладачів, науковців, молодих учених, аспірантів, студентів ЗНТУ (Запоріжжя, 20–25 квіт. 2013 р.). – Запоріжжя: ЗНТУ, 2013. – С. 270-271.

16. *Табунщик, Г.В.* Інженерія якості програмного забезпечення: навч. посіб. / Г.В. Табунщик, Р.К. Кудерметов, Т.І. Брагіна. – Запоріжжя: ЗНТУ, 2013. – 176 с.

Лекція 6. Методи доведення, верифікації та тестування програм. Мови специфікації

Сучасні напрями перевірки правильності програм – це формальні специфікації, методи доведення, верифікація і тестування. Наведемо їхній зміст.

1. Формальні специфікації з'явилися у програмуванні в 70-х роках минулого сторіччя. Вони подібні МП і надають засоби, що полегшують опис міркування про властивості і особливості програм у математичній нотації. Під *специфікацією* розуміють формальний опис функцій і даних програм, якими ці функції оперують. На формальних специфікаціях базуються методи доведення програм, які були започатковані працями з теорії алгоритмів А.А. Маркова [1], А.А. Ляпунова [2], схемами Ю.І.Янова [3] та формальними нотаціями опису взаємодіючих процесів К.А. Хоара [4] і ін. Для перевірки формальної специфікації програми застосовують математичний апарат для опису правильного розв'язку деякої задачі, для якої вона розроблена. Разом з специфікацією розробляються додаткові аксіоми [5–10], твердження про опис операторів і умов, так звані попередні умови або передумови, і постумови, що визначають заключні правили одержання правильного результату.

2. Доведення програм проводиться за допомогою *тверджень*, що складаються у формальній мові і слугують для перевірки правильності програми в заданих її точках. Набір тверджень, перед- і постумов використовується для перевірки отриманого результату у деякій точці програми. Якщо твердження відповідає скінченому оператору програми, то за допомогою постумови робиться остаточний висновок про часткову або повну правильність роботи програм.

3. Верифікація і валідація – це методи забезпечення перевірки й аналізу правильності виконання заданих функцій програми відповідно до заданих вимог замовника до них і системи у цілому.

4. Тестування – це метод виявлення помилок у ПС шляхом виконання вихідного коду на тестових даних, збирання робочих характеристик у динаміці виконання в конкретному операційному середовищі, виявлення різних помилок, дефектів, відмов і збоїв, викликаних нерегулярними, аномальними ситуаціями або аварійним припиненням роботи системи.

Теоретичні засоби реалізуються як процеси програмування і перевірки правильності програмного продукту. На даний час процеси верифікації, валідації і тестування ПС регламентовані стандартом ISO/IEC–12207 [7] з життєвого циклу ПС. У деякій зарубіжній літературі процеси верифікації і тестування на практиці ототожнюються, вони орієнтовані на досягнення правильності програми.

Наведені методи доведення, верифікації і тестування при перевірці правильності програм кваліфікуються такими загальними поняттями і діями.

Доведення та верифікація коректності (правильності) виконуються за формальною специфікацією програми та за допомогою тверджень, передумов (обмежень вхідних параметрів програми) і постумов (обмежень вихідних параметрів програми), які утворюють незалежну від програми частину механізму її доведення. Ця частина специфікується, як правило, на тій же мові, що і програма. У ній застосовуються математичні операції (диз'юнкції, кон'юнкції, імплікації тощо), квантори існування і загальності та інші.

Передумова – це обмеження на сукупність вхідних параметрів і постумови як обмеження на вихідні параметри. Передумова і постумова задаються предикатами, результатом яких є булева величина (true/false). Передумова істинна тоді, коли вхідні параметри входять в область припустимих значень даної функції. Постумова істинна тоді, коли сукупність значень задовольняє вимоги, щодо формального визначення критерію правильності одержання результату.

Твердження формулюються на формальній математичній мові у вигляді додаткової доказової частини, що перевіряє правильність виконання програми в початковій, проміжній або кінцевій точках.

Постумова – це обмеження з умов про кінцевий результат програми, відповідно до якого формулюється висновок про правильне завершення цієї програми.

Під *доведенням* часткової правильності розуміють перевірку виконання програми за допомогою тверджень, які описують те, що повинна одержати ця програма, коли закінчиться її виконання відповідно до умов заключного твердження. Повністю правильною програмою щодо її опису і заданих тверджень буде така програма, яка частково правильна і її виконання завершується при *всіх* даних, що відповідають їй.

Перевірка правильності методом тестування базується на функціональних тестах або наборах тестів, які створюються шляхом опису функцій і проектної інформації на процесах ЖЦ з урахуванням вимог, сформульованих на процесі аналізу предметної області.

5. Організаційна інфраструктура якісного проектування – це різні служби і діяльність груп фахівців, що планують процеси досягнення правильності програм (доказ, верифікація, тестування) з використанням описів тверджень, різних умов, а також тестових даних для спостереження за процесом доведення правильності програм, зокрема тестування, і збирання даних про відмови і помилки програм для їхнього використання при оцінюванні показників якості.

Далі зазначені напрями досягнення правильності програм розглядаються більш детально.

6.1. Мови специфікації програм і їхня класифікація

Мови формальної специфікації, які використовуються для формального опису властивостей виконання програм шляхом завдання тверджень та перед і постумов, є мовами вищого рівня щодо мов алгоритмічного програмування, які можуть використовуватися для опису програми, для якої створюється доказ.

У загальному випадку *формальна специфікація програми* – це однозначний специфікований опис програми за допомогою математичних понять, термінів, правил синтаксису і семантики формальної мови.

Опис деякої задачі являє собою сукупність її формальної специфікації та необхідних для доведення аксіом, тверджень, перед- і постумов та інших формалізмів. Всі ці описи при реалізації вимагають не систему програмування з МП, а спеціальний програмно реалізований математично орієнтований апарат доведення програм, зокрема інтерпретатори або метасистеми.

Існують різні підходи до класифікації мов специфікації, що наведені на рис.6.1.

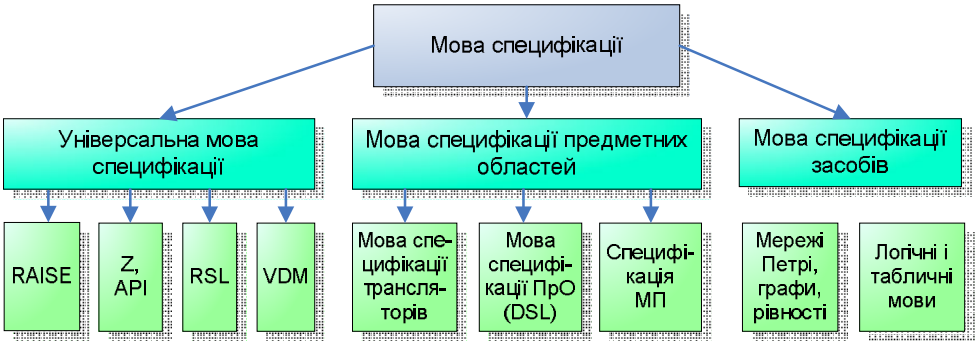

Рис.6.1. Категорії формальних мов специфікації

Нижче розглянуто основні мови специфікації, класифіковані за сферою застосування.

Універсальні мови специфікації – VDM, Z, RAISE, Spec# мають загально математичну основу з такими засобами:

1) логіки першого порядку, включаючи квантори;
2) арифметичні операції;
3) множини і операції над множинами;
4) описи послідовностей (кортежів, списків) і операції над ними;
5) описи функцій і операцій над ними;
6) описи деревоподібних структур;
7) засоби побудови моделей областей;
8) процедурні засоби мов програмування (оператори присвоювання, циклу, вибору, виходу);
9) операції композиції, аргументами і результатами яких можуть бути функції, вирази, оператори;
10) механізм конструювання нових структур даних.

Мови специфікації предметних областей (доменів) у програмуванні:

1) специфікації доменів;
2) описи взаємодій і паралельного виконання;
3) специфікації мов програмування і трансляторів;
4) специфікації баз даних і знань;
5) специфікації пакетів прикладних програм тощо.

Мови специфікації специфіки доменів DSL (Domain Specific Language) призначені для формалізованого опису задач в термінах предметної області, що підлягає моделюванню. Ці мови можна підрозділити на зовнішні і внутрішні. Зовнішні мови (типу UML, OWL і ін.) за рівнем вищі мов програмування і відповідають, наприклад, предметно-орієнтованої мові DSL, яка використовується для подання абстрактних понять і задач ПрО. Їхній опис трансформується до понять деякої внутрішній мові або мови програмування спеціальними генераторами або текстовими редакторами. Внутрішні мови – мови опису специфічних задач обмеженим синтаксисом і семантикою потребують препроцесорів для перебудови цього опису до базової мови програмування.

Специфікації опису взаємодій і паралельного виконання окремих процесів систем ПрО також добре подаються мовами DSL, наприклад, подібно – діаграм UML.

Мови програмування предметної області, доповнені засобами і механізмами технологій. Метапрограмування є ефективним засобом автоматизації специфікацій розроблених програм і в даний час знаходять широке застосування у галузі інформаційних технологій.

Формальні мови специфікації мов програмування спочатку застосовувалися при розробленні трансляторів. Так зазвичай синтаксис мови програмування описувався КС-граматиками у формі Бекуса–Наура. Такого типу мови є метамовами. Для специфікації семантики мов програмування використовуються формалізми рівностей. Техніка опису мов програмування базується на атрибутних граматиках і абстрактних типах даних з використанням денотаційних, алгебраїчних і атрибутних підходів. Як мови специфікації трансляторів, а також систем реального часу, де правильність і точність виконання програм є головними, використовують мови Z, VDM, RAISE.

Мови специфікації з орієнтацією на засоби програмування базуються на рівностях і підстановках з операційною семантикою (Лісп, Рефал); логічні мови; мови операцій (APL) над послідовностями і матрицями; табличні мови; мережі, графи та ін. Мова логіки предикатів використовується для запису передумов і постумов, інваріантів і процесу верифікації (наприклад, Пролог).

Для визначення семантики рівності застосовують денотаційний, операційний і аксіоматичний опис. Операційна семантика пов'язана з підстановками (заміна, продукція) і визначається в термінах операцій, що призводять до обчислень алгоритмів. При цьому фіксується порядок і динаміка виконання операцій програми.

У денотаційному підході до семантики надається перевага статичному опису об'єктів у термінах математичних властивостей, а у аксіоматичному – специфікації властивості об'єктів у рамках деякої логічної системи, що містить у собі правила виведення формул та/або інтерпретацій.

Окрім наведеної класифікації мов специфікацій, існують інші. Наприклад, можлива класифікація специфікацій за *способом виконання*:

– виконувана (executable),
– алгебраїчна (algebraic),
– сценарна (use case or scenarios),
– в обмеженнях (constraints).

Виконувані специфікації припускають розроблення прототипів систем для досягнення встановленої мети (VDM, SDL, RSL).

Алгебраїчні специфікації та мови SDL, RSL містять у собі механізми опису аксіом і тверджень, які призначені для доведення специфікованих програм.

Сценарні специфікації (UML) дозволяють описувати різні способи можливого застосування системи.

Програмування в обмеженнях використовують перед- і постумови для опису даних, операцій, інваріантів даних програм, що доводяться.

6.1.1. Мова формальної специфікацій – VDM

Мова специфікації VDM (Vienna Development Method) була розроблена у віденській лабораторії компанії IBM і призначалася для опису мов типу ПЛ/1,

трансляторів і систем із складними структурами даних. У мові специфікується правильна програма і набір тверджень для її доведення [7, 8].

Мова містить у собі такі типи даних:

X – натуральні числа з нулем;

N – натуральні числа без нуля;

Int – цілі числа;

Bool – булеві;

Qout – рядки символів;

Token – знаки і спеціальні позначення операцій.

Функція специфікує властивості структур даних і операцій над ними – апплікативно (функціонально) або імперативно (алгоритмічно). Наприклад, функції *min* у мові VDM описують двома способами:

$min\ N1\ N2 \to N3,$

$min\ (N1\ N2) = if\ N1 < N2\ then\ N3.$

Об'єкти мови VDM: множини, дерева, послідовності, відображення, сконструйовані складні структури.

Множина: X–set і операції $\in$, $\subseteq$, $\cup$, $\cap$ і ін. Правило: $x \in A$ буде коректним тільки тоді, коли A є підмножиною множини, якій належить x. Дистрибутивне об'єднання підмножин покажемо на прикладі:

union {(1, 2), (0, 2), (3, 1) } = (0, 1, 2, 3).

Списки (послідовності): X – операція, *len* – довжина списку, *inds* – номери елемента в списку. Наприклад, $inds\ lst = (i \in X\ [f \le i \le len])$.

Узяття першого (голови) елемента списку – *hd* і залишку (хвоста) після видалення першого елемента із списку – *tl*. Наприклад, *hd (a, b, c, d) = (a), tl (a, b, c, d) = (b, c, d).*

Дерево: mk об'єднує послідовності, множини і відображення. Елементи дерев можуть конструюватися. Наприклад, час 10.30 – конструкція *let mk*, час *(h, m) = t*, *tin* визначає значення $h = 10$, а $m = 30$.

Відображення: map – таблиця з ключів і значень. Операція *dom* будує множину ключів, *rng* – множину його значень.

Специфікація програми у VDM – це опис інваріантних властивостей, наприклад, *inv* – функція, аргументи і опис її операцій. Перевірки специфікації – це і перед- і постумови, твердження, які специфікуються засобами VDM, і мають таку семантику тлумачення у ньому.

Передумова – це предикат з операцією, до якої звертається інваріант програми після отримання початкового стану для визначення правильності виконання або фіксації помилкової ситуації. *Твердження* – це опис операцій перевірки правильності інваріанта програми в різних її точках. *Постумова* – це предикат, який є істинним після виконання передумови, завершення поточних операції в заданих точках при виконанні інваріантних властивостей програми.

Нижче наведено покрокову деталізацію специфікації програм мовою VDM:

1. Визначення термінів, якими буде специфікуватися програма.

2. Опис понять і об'єктів, для позначення яких використовується денотат, що ідентифікується за допомогою деякого імені (або фрази).

3. Опис інваріантних властивостей програми.

4. Визначення операцій над структурами програми (наприклад, ввести об'єкт, видалити і ін.), що змінюють її стан і збереження інваріантних властивостей.

5. Розроблення формальних умов виконання інваріанта програми та специфікація передумов, постумов, а також тверджень щодо виконання інваріанта програми.

6. Статичний огляд інваріанту програми щодо специфікованого формалізму доведення цього інваріанта.

7. Використання діючих CASE-засобів, що забезпечують виконання процесу верифікації програм.

Приклад програми. Алгоритм реєстрації компонента (ком) в репозитарії (repoz). Опис даних на мові «Паскаль» з використанням мовних конструкцій VDM при опису роботи зі списком компонентів (list_ptr).

```
post-Add_rd (r): Rn → bool
r ∉ elems l1.rds
li'.rds =li.rds <mk-R.card(r, <>)>& li'.ctlg = li.ctlgli'ril = li.ril= type
   list = record
          next: pointer
          value: pointer
    list_ptr = list;
  R_card = record
          r.name:  string;
          ком: list_ptr
       end;
   repoz = record
          rdrs: list_ptr;
          ctlg: list_ptr;
          ril: list_ptr
        end;
var
   L: repoz;
procedure Add_rd(r=string);
   rcrd:R_card;
   LL : list_ptr;
begin
    if find _rds, r) then
       begin
         writeln ('компонент зареєстрований)
       end;
   new (Rcrd);
   Rcrd.r_name := r;
   Rcd.ком:=nil;
   new (LL);
   LL.next:=L.rds;
   LL.value:=ltcrd;
   L.rds:=LI end.
```

6.1.2. Мова формальної специфікації – RAISE

Мова RAISE і RSL-специфікація (RAISE Specification Language) були розроблені в 80-роках XX ст. як результат попереднього дослідження формальних методів верифікації програм і поповнення їх новими можливостями щодо

доведення. Метод містить у собі нотації, техніку і інструменти для формального опису (RSL, C++ і Паскаль) програм і доведення їх правильності [9, 10].

До складу мови RSL входять абстрактні параметричні типи даних (специфікації, алгебри) і конкретні типи даних (модельно-орієнтовані), підтипи, операції для завдання послідовних і паралельних програм. Тобто ця мова надає аплікативний і імперативний стиль специфікації абстрактних програм, а також формальне конструювання окремих програм в інших мовах програмування і апарат доведення їх правильності. Синтаксис цієї мови близький до синтаксису мов C++ і Паскаль.

У мові RSL є абстрактні типи даних і конструктори складних типів даних, такі як добуток (*product*), множини (*sets*), списки (*list*), відображення (*map*), записи (*record*) і т.п.

Добуток типів – це впорядкована скінченна послідовність типів $T_1, T_2 ., T_n$ добутку (*product*) $T_1 \times T_2 ,..., \times T_n$.

Кількість компонентів добутку *d* – це *size (d) = id ∇ (null (couter inc(counter)))*. Конструктор добутку d_1 і d_2 будує добуток $d_1 \times d_2$ і значення типу *product* (*T1× T, × Tn*), тобто

make product (value 1 ., value n) = (value i ⇒ 1) ∇. ∇ (value n n),

де значення *value* і має тип T_i, а результуюче значення – тип добутку $T_1 \times T_2 \times, ..., \times T_n$ має значення *∇ (value ⇒ n).*

Списки типів – це послідовність значень одного типу *list T* – можуть бути скінченним списком типів *Tk* і нескінченним списком типів *Tn.* За структуру даних типу списків можна взято бінарне дерево, в якому є голова (*head*), син (tail), який слідує за ним у списку, і хвіст. До операцій списку належить операція *hd* – узяття першого елемента списку, тобто голови, і операція *tl* – узяття хвоста – решти елементів (як у мові VDM).

Функція *Caddr (I) = L ⇒ tail ⇒ tail ⇒ Head* вибирає із списку *I* елемент та індекс голови елемента, кількість елементів у списку і ін.

Відображення – це структура (*map*), яка ставить у відповідність значенням одного типу значення іншого типу. Разом з тим відображення – це бінарне відношення декартових добутків двох множин як сукупності двокомпонентних пар, в яких перший компонент – *arg,* що містить у собі елементи аргументів відображення, а другий компонент – *res* – відповідні елементи значень цього відображення.

Операції над відображеннями такі: накладення, об'єднання, композиція, зріз, композиція відображень (m_1, m_2).

Запис – це сукупність іменованих полів даних. Цей тип відповідає типу *record* у мові Паскаль і *struct* у мові C++. У мові RAISE для запису визначено два конструктори – *record, shurt record*.

Об'єднання – це конструктор *union,* що забезпечує об'єднання типів, наприклад,

type id = id_1, id_2,., idn і тип *id* , який одержує одне із значень у списку елементів.

Конструктор типу – це тип виду *type id = id_from_ id1 (id_to_ id1: id1*. Таким чином, мови VDM і RAISE слугують для математичного опису програм і конструювання структур даних як специфікацій, що використовуються при доведенні програм.

6.1.3. Концепторна мова специфікації

Для постановки складних математичних задач (підсумовування нескінченних рядів, теоретико-множинних операцій з нескінченними множинами тощо) і задач штучного інтелекту (ігри, розпізнавання образів тощо) з метою їх формального опису запропонована *загально математична процедурна мова,* а саме, *концепторна мова (КМ)* [16]. У цій мові процес опису складного завдання проводиться шляхом обґрунтування розв'язку задачі з математичної точки зору, потім формального опису постановок задач і, нарешті, переходу до алгоритмічного опису.

Специфікація складних завдань. *Концепторна мова* містить у собі декларативні й імперативні засоби теорії множин Цермело–Френкеля. Ядро цієї мови містить набір елементів (типи, вирази, оператори) і засоби визначення нових типів, виразів і операторів.

Декларативні засоби КМ – це типізована логіко-математична мова для опису *виразів* і структуризації множин значень (денотат). Вирази складаються з термів і формул, терми позначають об'єкти ПрО, а формули – твердження про об'єкти і відношення між ними.

Базові елементи мови – конструктори складених типів і формул, а саме функтори, предикати, конектори і субнектори.

Функтор – це конструктор, що перетворює терми на терми.

Предикати перетворюють терми на формули.

Конекторы вміщують логічні зв'язки і квантори для перетворення однієї формули в іншу.

Субнектор (дескриптор) – це конструктор побудови термів з виразів і формул.

Імперативні засоби КМ – це оператори і процедури для опису об'єктів ПрО за допомогою концепторів. Опис процедури має такий загальний вигляд:

концептор К (< список параметрів >)
< список імпортних параметрів>
< визначення констант, типів, предикатів>
< опис глобальних змінних>
< визначення процедури>
початок К
< тіло концептора>
кінець К.

Денотаційний підхід полягає у визначенні *семантики* і підстановці в кожний вираз опису елемента з множини денотатів функції φ символів з сигнатури мови. Кожній константі $c \in C$, функціональному символу $f \in F$ і предикативному символу $p \in P$ зіставляється об'єкт з множини денотат. Цей спосіб інтерпретації семантики виразів і операторів мови аналогічний денотаційній семантиці мов програмування.

Аксіоматичний опис КМ – це аксіоми і твердження щодо концепторного опису і проведення дедуктивного доведення і верифікації цього опису.

Логіко-алгебраїчні специфікації КМ призначені для специфікації ПрО, що задаються у вигляді алгебраїчної системи, за допомогою відповідних носіїв, сигнатури і трьох принципів. *Перший принцип* – логіко-алгебраїчна специфікація ПрО і уточнення понять ПрО, *другий принцип* – опис властивостей ПрО у вигляді

аксіом, які формулюються мовою предикатів першого порядку і хорновських атомарних формул, і, нарешті, *третій принцип* – це визначення термальних моделей з основних термів специфікації.

Засоби КМ використовуються при формальній специфікації *поведінки дискретних систем.* Для опису властивостей апаратно-програмних засобів динамічних систем застосовуються логіко–алгебраїчні специфікації. Техніка опису таких систем складається з двох процесів.

На першому процесі дискретна система *S* розглядається як «чорна скринька» з скінченним набором входів, виходів і станів. Області значень входів і виходів – довільні, а функціонування системи *S* – це набір часткових відображень і операцій алгебраїчної системи. Вони утворюють часткову алгебру, формальний опис якої виконується за допомогою алгебраїчних специфікацій, і це є програмою моделювання станів дискретної системи.

На другому процесі система *S* деталізується у вигляді сукупності взаємозалежних підсистем $S_1,..., Sn$, кожна з яких описується алгебраїчною специфікацією. Внаслідок цього одержують специфікацію системи *S* із функцій переходів і виходів, для яких необхідно доводити коректність. Процес деталізації виконується на рівні елементної бази або елементарних програм і супроводжується доведенням їх коректності. Отже, маємо, що система *S* еквівалентна початковій абстрактній специфікації.

Приклади доведення. Нехай треба побудувати специфікацію натуральних чисел з множини цих чисел і сигнатури операцій $\Sigma = (+, \times, \leq)$. При побудові використовується число 0 і функція проходження s: $N \rightarrow N'$. Специфікація складається з таких аксіом:

$x+0 = x$

$x+s(y) = s\ (x+y)$

$x \times 0 = 0$

$x+s(y) = s\ (x \times y) +x$

$0 \leq x$

$x \leq в \supset s\ (x) \leq s\ (y)$

$s\ (x) \leq s\ (y) \supset x \leq в$

Алгебраїчні

специфікації називають мовами логіко-алгебраїчних специфікацій, їх операційна семантика заснована на переписуванні термів, а утворена алгебраїчна специфікація одержує логічну семантику, використовувану при доведенні теорем.

6.1.4. Звичайна мова специфікації Spec#

Сучасна мова специфікація Spec# є розширенням об’єктно-орієнтованої мови C# засобами, що забезпечують верифікацію програм для платформи .Net [27]. Ці засоби подаються до програми в C# невеликими додатковими описами, а саме:

– ненульових посилань до параметрів викликів CALL;

– контрактів між викликами і реалізаціями;

– обробки виникаючих виключних ситуацій програми для інформування розробника;

– змінювання полів даних об’єктів тощо.

Ненульові типи даних помічаються типом T! і відповідають деяким змінним програми, які можуть використовуватися при специфікації полів даних,

формальних параметрів і з обернених цьому типу значень, локальних змінних програми. Цей тип не належить до елементів масиву. Головне призначення ненульових типів – забезпечити посилання до інших елементів, опис патернів, перевірку умов виходу з виразів і циклів, обумовлених контрактом.

Контракт ставиться між тим, хто робить виклик, і хто – реалізацію. У передумові додається опис стану параметрів при виклику, в постумові визначається умова отримання результату об'єкта, що викликався, і передача цього результату протилежна. Spec# надає підтримку більш дисциплінованому використанню виключення, щоб забезпечити ясність і підтримку життєздатності програми. У програмі можуть бути відмови і помилки. У даному випадку у методі використовується аналіз забороненої умови, коли передумова була не задоволена. Більшість відмов у програмі – це, коли порушена умова контракту. Наприклад, отримання виходу з циклу при перевищенні значення параметра циклу, що не було визначено у передумові.

Обробка виключних ситуацій виконується при роботі з масивами, коли елемент не відповідає типу. Якщо в передумові специфікується індекс, що знаходиться всередині меж масиву, а при виконанні цього не відбувається, то компілятор відповідає клієнту про невиконання передумови в реальному часі.

Змінювання полів даних задається фреймовими умовами, що вміщується у контракт, і починаються modifies, за яким слідкує оператор частини програми методу реалізації, що дозволяє зміну.

Приклад.

```
Class C{
Int x, y;
Void M () modifies x: {…}.
}.
```

Тут метод M дозволяє зміну *x* при умові, що на виході з цього методу *y* має те саме значення, що на вході. У випадку масиву такий оператор змінює не елементи масиву, а посилання – на цей масив.

Фреймові умови для сервера використовуються під час виконання програми. При цьому modifies перевіряє на вході усі передумови і постумови виконання операторів програми в C#, на яких базуються описи специфікацій в Spec# і рахуються коректними.

Успадкування специфікацій. Контрактний метод успадковується звичайним методом, який під час виконання звертається до нього. Специфікація в Spec# подає код виконання в більш наглядній формі і більш зручній для перевірки заданих постумов. Метод може додавати до опису специфікації додаткові постумови і різні реакції на виключні ситуації. Метод має об'яву в інтерфейсі, подібно до об'яві в класі. Коли клас виконує метод інтерфейса, то його опис містить у собі фреймові умови, котрі є суперкласом виконануваного методу. Для виконання фреймових умов використовується оператор **expose.** Він, як правило, визначає об'єкт, інваріантний модифікації. Специфікація в Spec# являє собою блок операторів, який явно вказує на те, коли об'єктний інваріант можуть явно використовувати оператори, що вказані після **expose**. При цьому можуть модифікуватися усі поля в структурі об'єкта.

Підхід до реалізації специфікації. Опис специфікації в Spec# є самостійною програмою, що містить у собі перед- і постумови, а також різні дії над фреймовими структурами щодо програми, яка перевіряється мовою C#. Ця специфікація

подається в мово-незалежному форматі і перебудовується в мову C# за допомогою спеціального транслятора, що працює на платформі .Net. Цей транслятор має аналізатор і версифікатор для перевірки правильності опису специфікації. Транслятор виконує переклад у вигляді частини програми, для якої створювалася специфікація. Верифікація специфікації є статичною, вона орієнтована на перевірку правильності опису, а саме, меж масивів, явних значень змінних тощо. Прувер транслятора виконує перевірку деяких умов і операторів, а також значень змінних. Специфікації в Spec# накопичуються у репозитарії, і при застосуванні звертаються до Base Class Library, де накопичуються об'єкти, їхні інваріанти та контракти.

Контракти і аналогічні механізми верифікації програм з мов програмування реалізовані також в системах Java, Eiffel і Spark. У мові Java контракти вміщуються в опис програми як стилізовані коментарі. Середовище Java має такі засоби: перевірку контрактів у динаміці, виконання, виклики та об'єктні інваріанти. В об'єктно-орієнтованій системі Eiffel є бібліотека констрейнів, котрі вставляються у опис об'єкта і виконують верифікацію в динаміці виконання. Однак механізми модифікації не дозволяють для callbacks об'єктних інваріантів і тому вони не вміщуються в модульні об'єкти. Система Spark підтримує *підмножину* мови Ада при вставці теорем для прувера, помічених як коментарі, але компілятор цієї мови їх не використовує. Засоби верифікації в Spark орієнтовані на окремий опис умов виконання Ада-програм для верифікації, аналогічно до методології Spec#. Вони окремо транслюються у вихідний код Ада-програми і виконуються разом з нею в режимі верифікації.

Лекція 7. Методи доведення правильності програм. Верифікація і валідація програм

Формальні методи тісно пов'язані з математичною специфікацією, верифікацією і доведенням правильності програм. Ці методи містять у собі математичну символіку, формальну нотацію і апарат виведення. Правила доведення є громіздкими і тому на практиці рідко використовуються рядовими програмістами. Проте з теоретичної точки зору вони слугують розвитку логіки застосування математичного методу індукції в процесі перевірки правильності програм [4, 5, 17, 18].

7.1. Базові методи доведення

Відомо багато методів доведення специфікацій програм, деяким з них дамо коротке визначення.

Метод Флойда заснований на знаходженні умов для вхідних і вихідних даних і полягає у виборі контрольних точок у програмі, яка доводиться, таким чином, щоб шлях проходження перетинав хоча б одну контрольну точку. Для цих точок формулюються твердження про стан і значення змінних у них (для циклів ці твердження повинні бути істинними при кожному проходженні циклу – інваріанта).

Кожна точка розглядається для індуктивного твердження того, що формула залишається істинною при поверненні програми в цю точку, і залежить не тільки від вхідних і вихідних даних, а й від значень проміжних змінних. На основі індуктивних тверджень і умов на аргументи програми створюються твердження з умовами перевірки правильності цієї програми в окремих її точках. Для кожного шляху програми між двома точками встановлюється перевірка на відповідність

умов правильності і визначається істинність цих умов при успішному завершенні програми на даних, що задовольняють вхідні умови.

Метод Хоара – це вдосконалений метод Флойда, заснований на аксіоматичному описі семантики мови програмування початкових програм. Кожна аксіома описує зміну значень змінних за допомогою операторів цієї мови. Формалізація операторів переходу і викликів процедур забезпечується за допомогою правил виводу, що містять у собі індуктивні вислови для кожної точки і функції початкової програми.

Метод Маккарті полягає у структурній перевірці функцій, що працюють над структурними типами даних, структур даних і шляхів переходу під час символьного виконання програм. Ця техніка складається з моделювання виконання коду з використанням символів для змінних даних. Тестова програма має вхідний стан, дані і умови її виконання.

Виконувана програма розглядається як серія змін станів. Саме останній стан програми вважається вихідним станом і, якщо його одержали, то програма вважається правильною. Даний метод забезпечує високу якість початкового коду.

Метод Дейкстри пропонує два підходи до доведення правильності програм. Перший підхід заснований на моделі обчислень, що оперує історіями результатів обчислень програми, аналізом шляхів проходження і правил оброблення великого об'єму інформації. Другий підхід базується на формальному дослідженні тексту програми за допомогою предикатів першого порядку. У процесі виконання програма одержує деякий стан, який запам'ятовується для подальших порівнянь.

Основу методу становить математична індукція, абстрактний опис програми і її обчислення. За допомогою цього методу можна довести істинність деякого припущення $P(n)$ залежно від параметра n для всіх $n \geq n_0$, і тим самим довести випадок $P(n_0)$. Виходячи з істинності $P(n)$ для будь-якого значення n, доводимо $P(n+1)$, що достатньо для доведення істинності $P(n)$ для всіх $n \geq n_0$.

7.2. Модель доведення програми за твердженнями

Розглядається формальне доведення програми, заданої структурною логічною схемою і сукупністю тверджень, що описуються логічними операторами, комбінаціями змінних (true/false), операціями (кон'юнкція, диз'юнкція й ін.) і кванторами загальності й існування (табл.7.1).

Таблиця 7.1. Список логічних операцій

Логічна операція		
Назва	**Приклад**	**Значення**
Кон'юнкція	$x \,\&\, y$	x і y
Диз'юнкція	$x * y$	x або y
Суперечність	$\neg x$	ні x
Імплікація	$x \rightarrow y$	якщо x то у
Еквівалентність	$x = y$	X рівнозначно y
Квантор загальності	$\forall x\, P(x)$	для всіх x, умова істинно
Квантор існування	$\exists x\, P(x)$	Існує x, для якого $P(x)$ істина

Мета алгоритму програми – побудова для масиву цілих чисел *T* довжини *N* (array *T*[1:*N*]) еквівалентного масиву *T'* тієї ж довжини *N,* що і масив *T*. Елементи в масиві *T'* повинні розташовуватися в порядку зростання їхніх значень.

Даний алгоритм реалізується сортуванням елементів вихідного масиву *T* за їхнім зростанням.

Доведення правильності алгоритму сортування елементів масиву *T* (рис. 7.1) проводиться з використанням ряду тверджень про елементи цього алгоритму, якій описуються пунктами П1– П6.

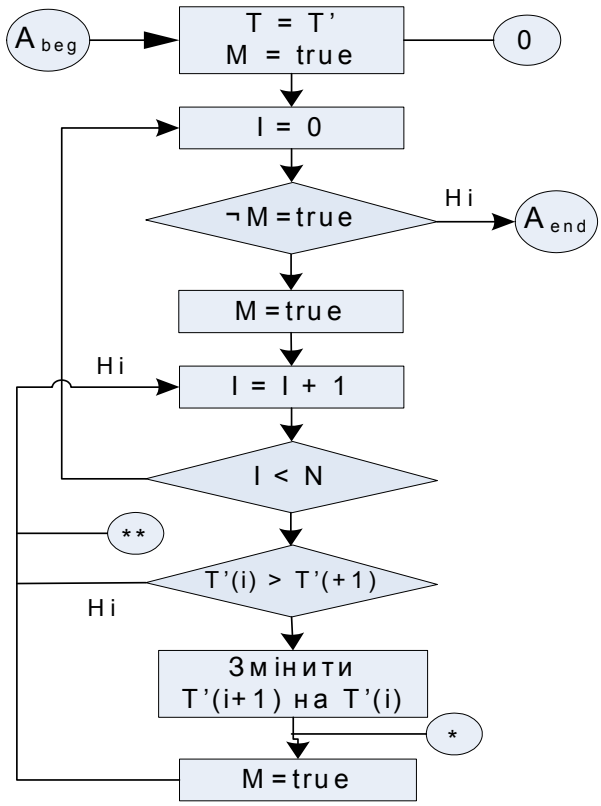

Рис.7.1. Схема сортування елементів масиву *T'*

П1. Вхідна умова алгоритму задається у вигляді початкового твердження:
A_{beg}: (*T* [1:*N*] – масив цілих) & (*T'* [1:*N*] – масив цілих).
Вихідне твердження A_{end} – це кон'юнкція таких умов:
(а) (*T*– масив цілих) & (*T'* – масив цілих)
(б) ($\forall i$, якщо $i \le N$, то $\exists j$ ($T'(i) \le T'(j)$)),
(в) ($\forall i$, якщо $i < N$, то ($T'(i) \le T'(i+1)$),
Тобто A_{end} – це:
(*T* – масив цілих) & (*T'* – масив цілих)
& $\forall i$, якщо $i \le N$, то $\exists j$ ($T'(i) \le T'(j)$),
& $\forall i$, якщо $i < N$, то ($T'(i) \le T'(i+1)$).

Розташування елементів масиву T в порядку зростання їхніх величин у масиві T' здійснюється алгоритмом бульбашкового сортування, суть якого полягає в попередньому копіюванні масиву T з масиву T', а потім проводиться сортування елементів згідно з умовою їхнього зростання.

У кружках також дано: початковий стан – 0, стан після перестановки місцями двох сусідніх елементів у масиві T' – одна зірочка, стан після зміни місцями всіх пар за один прохід усього масиву T' – дві зірочки.

Крім уже відомих змінних T, T' і N, в алгоритмі використані ще дві змінні: i – типу ціла і M – булева змінна, значенням якої є логічні константи true і false.

П2. Для доведення того, що алгоритм дійсно забезпечує виконання вихідних умов, розглянемо динаміку їхнього виконання послідовно у визначених точках алгоритму.

Зазначимо, що точки поділяють алгоритм на відповідні частини, правильність кожної з них визначається окремо.

Так, оператор присвоєння означає, що для всіх i ($i \le N$ & $i>0$) виконується ($T'[i] := T[i]$).

Результат виконання алгоритму в точці з нулем може бути виражений твердженням

($T[1: N]$ – масив цілих) & ($T'[1: N]$ – масив цілих)

& ($\forall i$ якщо $i \le N$, то ($T[i] = T[i]$)).

Доведення очевидно, оскільки за семантикою оператора присвоювання (по елементне пересилання чисел з T в T') самі елементи при цьому не змінюються, до того ж у даній точці їхній порядок у T и T' однаковий. Отже, одержали, що виконується умова «б» вихідного твердження.

Зазначимо, що перший рядок доведеного твердження збігається з умовою «а» вихідного твердження A_{end} і залишається справедливим до кінця роботи алгоритму, тому в наступних твердженнях наводитися не буде.

У точці з одною зірочкою виконаний оператор

$(i < N)\ (T'(i)) > T'(i + 1) \rightarrow (T'(i)$ і $T'(i + 1)$ міняє місцями елементи.

Як результат роботи оператора буде справедливе таке твердження:

$\exists\ i$, якщо $i < N$, то $(T'(i) < T'(i + 1)$,

яке є частиною умови «в» твердження A_{end} (для однієї конкретної пари суміжних елементів масиву T'). Очевидно також, що семантика оператора зміни місцями не порушує умову «б» вихідного твердження A_{end}.

У точці з двома зірочками виконані всі можливі перестановки місцями пари суміжних елементів масиву T' за один прохід через T', тобто оператор зміни працював раз або більше. Однак бульбашкове сортування не дає гарантії, що досягнуто упорядкування за один прохід по масиву T', оскільки після чергової зміни індекс i збільшується на одиницю незалежно від того, як співвідноситься новий елемент $T'(i)$ з елементом $T'(i – 1)$.

У цій точці також справедливе твердження

$\exists i$, якщо $i < N$, то $T'(i) < T'(i + 1)$.

Частина алгоритму, позначена точкою з двома зірочками, виконується доти, поки не буде упорядкований весь масив, тобто не буде виконуватися умова «а» твердження A_{end} для всіх елементів масиву T':

$\forall i$, якщо $i < N$, то $T'(i) \le T'(i+1)$.

Отже, виконання вихідних умов забезпечене порядком і відповідною семантикою операторів перестановки масиву.

Доведено, що виконання алгоритму програми завершено успішно, це означає її правильність.

П3. Цей алгоритм можна подати у вигляді серії теорем, що доводяться. Починаючи з першого твердження і переходячи від одного перетворення до іншого, визначаємо індуктивний шлях висновку. Якщо одне твердження є правильним, то істинним є й інше. Іншими словами, якщо дано перше твердження A_1 і перша точка перетворення A_2, то перша теорема – $A_1 \rightarrow A_2$. Якщо A_3 – наступна точка перетворення, то другою теоремою буде $A_2 \rightarrow A_3$.

Інакше кажучи, формулюється загальна теорема $A_i \rightarrow A_j$, де A_i й A_j – суміжні точки перетворення. Ця теорема формулюється так: якщо умова істинна в останній точці, то і вихідне твердження $A_k \rightarrow A_{end}$ є істинним.

Тобто, можна повернутися до точки перетворення A_{end} і до попередньої точки перетворення. Якщо доведемо, що $A_k \rightarrow A_{end}$ справджується, то виходить, що справджується й $A_j \rightarrow A_{j+1}$, і так далі, поки не одержимо, що $A_1 \rightarrow A_0$.

П4. Далі специфікується твердження типу *if – then*.

П5. Щоб довести, що програма коректна, необхідно послідовно розташувати усі твердження, починаючи з A_1 і закінчуючи A_{end}, цим буде підтверджено істинність вхідної і вихідної умов.

П6. Доведення алгоритму програми завершено.

7.3. Верифікація і валідація програм

Верифікація і валідація – це методи аналізу, перевірки специфікацій і правильності виконання програм відповідно до заданих вимог і формального опису програми [18–21].

Метод верифікації допомагає зробити висновок про коректність створеної програмної системи при її проектуванні і після завершення її розроблення. Валідація дозволяє встановити здійсниміcть заданих вимог шляхом їх перегляду, інспекції і оцінки результатів проектування на процесах ЖЦ для підтвердження того, що здійснюється коректна реалізація вимог, дотримання заданих умов і обмежень до системи. Верифікація і валідація забезпечують перевірку повноти, несуперечності і однозначності специфікації і правильності виконання функцій системи.

Верифікації і валідації піддаються:

– компоненти системи, їх інтерфейси (програмні, технічні і інформаційні) і взаємодія об'єктів (протоколи, повідомлення) у розподілених середовищах;

– описи доступу до баз даних, засоби захисту від несанкціонованого доступу до даних різних користувачів;

– документація до системи;

– тести, тестові процедури і вхідні набори даних.

Верифікація і валідація як методи перевірки правильності виконання заданих функцій і відповідності їх вимогам замовника подані в стандарті [7-9] у вигляді самостійних процесів ЖЦ і використовуються, починаючи від етапу аналізу вимог і закінчуючи перевіркою правильності функціонування програмного коду на заключному процесі, а саме, під час тестування.

Для цих процесів визначені цілі, задачі і дії з перевірки правильності створюваного проміжного продукту на процесах ЖЦ. Розглянемо їхнє стандартне подання.

Процес верифікації. Мета процесу – переконатися, що кожен програмний продукт (і/або сервіс) проекту відбиває погоджені вимоги до їхньої реалізації. Цей процес ґрунтується:

– на стратегії і критеріях верифікації всіх робочих програмних продуктів на ЖЦ;

– на виконанні дій з верифікації відповідно до стандарту;

– на усуненні недоліків, виявлених у програмних (робочих, проміжних і кінцевих) продуктах;

– на узгодженні результатів верифікації з замовником.

Процес верифікації може проводитися виконавцем програми або іншим співробітником тієї ж організації, або співробітником іншої організації, наприклад представником замовника. Цей процес містить у собі дії з його впровадження і виконання.

Впровадження процесу полягає у визначенні критичних елементів (процесів і програмних продуктів), що повинні піддаватися верифікації, у виборі виконавця верифікації, інструментальних засобів підтримки процесу верифікації, у складанні плану верифікації і його затвердження. У процесі верифікації виконуються задачі перевірки умов: контракту, процесу, вимог, інтеграції, коду і документації.

Відповідно до плану і вимог замовника перевіряється правильність виконання функцій системи, інтерфейсів і взаємозв'язків компонентів, а також доступ до даних і до засобів захисту.

Процес валідації. Мета процесу – переконатися, що специфічні вимоги для програмного продукту виконано, і здійснюється це за допомогою:

– розробленої стратегії і критеріїв перевірки всіх робочих продуктів;

– обговорених дій з проведення валідації;

– демонстрації відповідності розроблених програмних продуктів вимогам замовника і правилам їхнього використання;

– узгодження із замовником отриманих результатів валідації продукту.

Процес валідації може проводитися самим виконавцем або іншою особою, наприклад, замовником, що здійснює дії з впровадженню і проведенню цього процесу за планом, у якому відбиті елементи і задачі перевірки. При цьому використовуються методи, інструментальні засоби і процедури виконання задач процесу для встановлення відповідності тестових вимог і особливостей використання програмних продуктів проекту на правильність реалізації вимог.

На інших процесах ЖЦ виконуються додаткові дії:

– перевірка і контроль проектних рішень за допомогою методик і процедур перегляду ходу розроблення;

– звернення до CASE-систем [10], що містять у собі процедури перевірки вимог до продукту;

– перегляди й інспекції проміжних результатів на відповідність вимогам для підтвердження того, що ПС має коректну реалізацію вимог і задовольняє умови виконання системи.

Таким чином, основні задачі процесів верифікації і валідації полягають у тому, щоб *перевірити і підтвердити*, що кінцевий програмний продукт відповідає призначенню і задовольняє вимогам замовника. Ці процеси взаємозалежні і визначаються, як правило, одним загальним терміном «верифікація і валідація» або «Verification and Validation» (V&V) [19].

V&V засновані на плануванні їх як процесів, так і перевірки для найбільш критичних елементів проекту: компонентів, інтерфейсів (програмних, технічних і інформаційних), взаємодій об'єктів (протоколів і повідомлень), передачі даних між компонентами і їхнього захисту, а також створення тестів і тестових процедур.

Після перевірки окремих компонентів системи проводяться їхня інтеграція, повторна верифікація і валідація інтегрованої системи, створюється комплект документації, що відображає правильність виконання вимог за результатами інспекцій і тестування.

7.3.1. Підхід до валідації сценарію вимог

До процесу створення програм належить опис вимог мовою UML за допомогою сценаріїв і діючих виконавців – акторів як зовнішніх сутностей щодо системи [22]. Вимоги потрібно перевіряти до їхньої перебудови у програмні елементи. Сценарій після трансформації – це послідовність взаємодій між одним або декількома акторами і системою, у якій актор виконує мету сценарію при взаємодії з нею. У моделі вимог сценарій задає кілька альтернативних подій, заданих мовою діаграм UML. Вони розділяються на функціональні (системні) і внутрішні, як визначальне поводження системи. На основі опису сценарію вимоги перевіряються шляхом валідації для виявлення помилок у поданні сценарних вимог. Ця перевірка відбувається ітераційною і складається з наступних кроків:

1. Формалізований опис вимог у вигляді сценаріїв;
2. Створення моделі вимог;
3. Створення спеціальних сценаріїв для валідації вимог;
4. Застосування валідаційних сценаріїв у моделі вимог;
5. Оцінювання результатів поводження моделі вимог;
6. Перевірка умов завершення процесу валідації і при виявленні яких-небудь неточностей повторення кроків, починаючи з п. 2.

При виконанні сценаріїв можуть виникнути помилкові ситуації, за яких поведінки системи стає не детермінованим. За цих цілей проводиться контроль покриття сценаріїв у моделі вимог валідаційними сценаріями з метою виявлення помилок або ризиків (рис. 7.2).

Створюється модель помилок, що покриває модель вимог системи з типовими помилками, що використовуються при доведенні сценарієв.

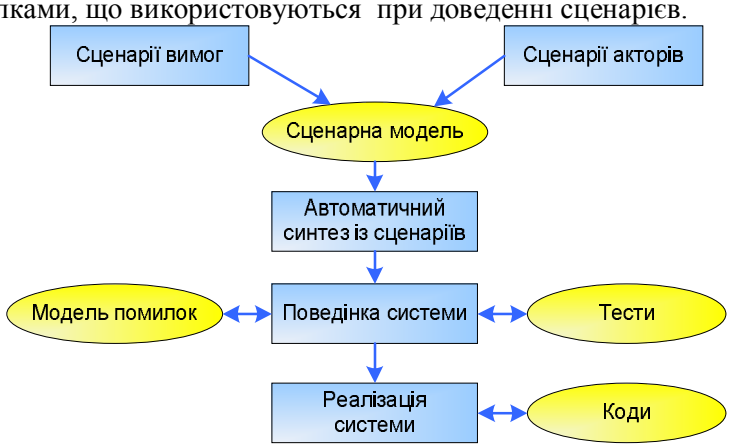

Рис. 7.2. Валідація сценаріїв вимог до системи

Складова частина валідації вимог за сценаріями – визначення класів еквівалентності вхідних і вихідних даних для валідації і синтезу сценаріїв. Вхідна інформація для синтезу сценаріїв – сценарна модель, що задається мовою взаємодії.

Інформація використовується при генерації додаткових сценаріїв з метою поліпшення процесу валідації, автоматичного синтезу сценаріїв моделі й отримання моделі поведінки системи під керуванням актора.

Модель перевіряється за допомогою тестів і моделі помилок, що в цілому дозволяє знайти неповноту вихідних вимог або суперечності у вимогах.

Автоматичний синтез програми заснований на наступних процедурах:

– валідація вимог шляхом виконання валідаційних сценаріїв;

– додавання перевірених сценаріїв до набору валідаційних сценаріїв і їхнє використання як вхідних даних для синтезу;

– пошук помилок у сценаріях і перевірка різних композицій сценаріїв.

Синтез специфікацій сценаріїв вимог, трансформованих до діаграмам взаємодії, може проводитися в середовищі системи Rational Rose.

7.3.2. Верифікація об’єктних моделей

Верифікація об’єктної моделі (ОМ) ґрунтується на специфікації:

– базових (простих) об'єктів ОМ, атрибутами яких є дані та операції об'єкта – функції над цими даними;

– об'єктів, які вважаються перевіреними, якщо їх операції використовуються як теореми, що застосовуються над підоб'єктами і не виводять їх з множини станів цих об'єктів.

Доведення правильності побудови ОМ передбачає:

– введення додаткових і (або) видалення зайвих атрибутів об'єкта і його інтерфейсів в ОМ, доведення правильності об'єкта ОМ на основі специфікації інтерфейсів і взаємодій з іншими об'єктами;

– доведення правильності завдання типів для атрибутів об'єкта, тобто правильності того, що вибраний тип реалізує операцію, а множина його значень визначається множиною станів об'єкта.

Це доведення є завершальним при перевірці правильності ОМ.

Верифікація інтерфейсів об'єктів ОМ зводиться до доведення правильності передачі типів і кількості даних в параметрах повідомлень про їхні специфікації в мові IDL. Інтерфейс складається з операцій звернення до об'єкта, який посилає дані іншому об'єкту через повідомлення. Для доведення правильності специфікації повідомлення створюється набір тверджень, який доводить, що для будь-якої пари елементів повідомлення, наприклад, A і B, перехід від A до B відбувається за один крок. Дія, що виконується в проміжку між A і B, приводить до B. При цьому частина тверджень перевіряє вхідний параметр і його надходження на вхід іншого об'єкта з метою підтвердження його на виході. Якщо доведено, що об'єкт, ініційований повідомленням, формує правильний вихідний результат у вихідному параметрі, то повідомлення вважається правильним.

Верифікація моделі розподіленого застосування виконується на основі специфікації SDL (Specification Description Language), моделі перевірки (Model Checking), індуктивних тверджень, запропонованих Новосибірською школою програмування [13].

Метод перевірки полягає в редукції системи з нескінченним числом станів до системи із скінченного числа станів, а також у доведенні коректності розподіленого

застосування за допомогою індуктивних міркувань і системи переходів скінченного автомата.

Основні підходи до верифікації – аксіоматичний і семантичний шлях Model Checking.

Аксіоматичний (за методом Hoare) підхід міститься в описі програми набором аксіом для завдання станів з використанням теорія логіки.

Семантичний підхід ґрунтується на теорії темпоральної логіки Манна для завдання специфікації програм. Аксіоми використовуються для керування семантикою мови специфікації.

Основними типами даних специфікації в SDL є наперед визначені і сконструйовані типи даних (масив, послідовність і т.д.). У мові описуються формули за допомогою предикатів, булевих операцій, кванторів, змінних і модальностей. Семантика їх визначення залежить від можливих послідовностей дій (поведінки), що виконуються специфікацією процесу, а також моменту часу його виконання.

Схема специфікації процесу – це опис умов виконання і діаграм процесів. Вона ініціюється посиланням повідомлення із зовнішнього середовища для виконання. Діаграма процесу складається з описів переходів, станів, набору операцій процесу і переходу до наступного стану.

Кожна операція визначає поведінку процесу і спричиняє деяку подію. Логічна формула задає модальність поведінки специфікації і моменти часу. Процес, наданий формальною специфікацією, виконується не детерміновано. Обмін із зовнішнім середовищем відбувається через вхідні і вихідні параметри повідомлень.

Подія. У кожний момент часу виконання процес має деякий стан, який може бути поданий у вигляді знімка, що характеризує деяку подію, яка містить у собі значення змінних, яким відповідають параметри і характеристики станів процесу.

Таким чином, модель перевірки, набір аксіом мовою логіки і твердження про виконання розподілених програм забезпечують процес їхньої верифікації.

7.3.3. Підхід до верифікації композиції компонентів

Метод верифікації композиції компонентів базується на специфікації функцій і часових (temporal) властивостей готових перевірених компонентів (типу reuse) і виконується за допомогою абстракцій моделі перевірки Model Checking [20].

Загальна компонентна модель (ЗКМ) – це сукупності перевірених специфікацій компонентів, часових властивостей і умов функціонування для асинхронної передачі повідомлень (АПП).

Модель перевірки забезпечує верифікацію програм на надійність шляхом розв'язання наступних задач:

– специфікація компонентів мовою xUML [26] діалекту UML з описом часових властивостей;

– опис функцій reuse компоненти, специфікації інтерфейсу і часових властивостей;

– перевірка властивостей складних компонентів композиційним апаратом.

Компоненти моделі можуть бути примітивними і складними.

Властивості примітиву перевіряються за допомогою моделі перевірки, а властивість складного компонента – на абстракції компонентів, зібраних з примітивів і перевірених їхніх властивостей в інтегрованому середовищі.

Даний підхід може використовуватися у розподілених програмних системах, що функціонують на платформах типу CORBA, DCOM і EJB.

Модель компонента в ЗКМ моделі має вигляд*:*

C = (E, I, V, P),

де *E* – початковий опис компонента; *I* – інтерфейс цього компонента; *V* – множина змінних, визначених в множині *E* і пов'язаних з властивостями з множини *P*; *P* – часові властивості середовища компонента.

Властивість компонента *C* включається в *P* тоді, коли вона перевірена у середовищі і представлена парою *(p, A(p)) на* множині *E,* де *p* – властивість компонента *C* в *E*, *A(p)* – множина часових формул з властивостей, визначених на множинах *I* і *V*.

Композиція компонентів – це сукупність простіших компонентів: (*E_0, I_0, V_0, P_0), ..., (En–i, In–i, Vn–i, Pn–i)*, визначених на моделі компонента *C* .

Модель обчислень АПП – це обчислювальна модель системи, задана на скінченній множині взаємодіючих процесів, представлених кортежами:

P = (X, Q ∇), де *X* – множина змінних, кожна з яких має тип; *∑* – розширена модель стану; *Q* – черга повідомлень у порядку надходження; *∇* – множина початкових значень для кожної змінної з *X, E* і порожнє для *Q*.

Модель стану ПС (Ф, M, T), де *Ф* – множина станів; *M* – множина типів повідомлень; *T* – набір переходів, визначених на множині *Ф* і *M*.

Асинхронна передача повідомлень АПП викликає чергування переходів станів і дій процесів. Для двох процесів *P1* і *P2* передача повідомлення від *P1* до *P2* містить у собі: тип повідомлення *m* з множини *M* для *P2* і відповідні параметри. Коли оператор дії виконується, повідомлення *m* з параметрами ставиться у чергу до процесу *P2*.

7.3.4. Загальні перспективи верифікації програм

Методи формальної верифікації використовувалися для перевірки правильності моделей ПрО, функцій в мові API, безпеки і цілісності БД – у проекті SDV фірми Microsoft і у *міжнародному проекті з формальної верифікації ПС*.

Ідея створення цього проекту належить Т.Хоару і обговорювалася на симпозіумі з верифікованого ПС у лютому 2005г. у Каліфорнії. Потім у жовтні того ж року на конференції IFIP в Цюріху був прийнятий міжнародний проект строком на 15 років з розроблення цілісного автоматизованого набору інструментів для перевірки коректності ПС [14, 15]. У проекті сформульовані такі основні задачі:

– розроблення єдиної теорії створення і аналізу програм;

– побудова всеосяжного інтегрованого набору інструментів верифікації для всіх процесів, включаючи розроблення специфікацій і їх перевірку, генерацію тестових прикладів, уточнення, аналіз і верифікацію програм;

– створення репозитарію формальних специфікацій і верифікованих програмних об'єктів різних видів і типів.

Репозітарій – це сховище правильних програм, специфікацій і інструментів.

Функції репозитарію:

– накопичення верифікованих специфікацій, методів доведення, програмних об'єктів і реалізацій кодів для різних програмних застосувань;

– накопичення всіляких методів верифікації, їх оформлення у вигляді, придатному для пошуку і відбору реалізованої теоретичної концепції для подальшого застосування;

– розроблення стандартних форм для завдання і обміну формальними специфікаціями різних об'єктів, інструментів і готових систем;

– розроблення механізмів взаємодії для перенесення готових верифікованих продуктів з репозитарію в нові розподілені і мережні середовища для їхнього використання в нових ПС.

Даний проект передбачається розвивати протягом 50 років. Відомо, що більш ранні проекти ставили подібні цілі: поліпшення якості ПС, формалізації сервісних моделей, зниження складності за рахунок використання КПВ, створення налагоджувального інструментарію для візуальної діагностики помилок і їх усунення тощо. Проте корінної зміни в програмуванні поки не відбулося. Залучення техніки формальної специфікації програм ще не означає, що в програмі будуть відсутні помилки, оскільки помилки в програмних проектах, в інтерпретації специфікацій МП, у документації поки не можна розпізнати. Реалізація міжнародного проекту з верифікації ПС допоможе вирішити багато з цих питань.

Лекція 8. Тестування програмних систем

Тестування – це метод виявлення помилок у ПС шляхом виконання вихідного коду на тестових даних, збирання робочих характеристик у динаміці виконання в конкретному операційному середовищі, виявлення різних помилок, дефектів, відмовлень і збоїв, викликаних нерегулярними, аномальними ситуаціями або аварійним припиненням роботи системи. Його можна розглядати, як процес семантичного налагодження (перевірки) програми, що полягає у виконанні послідовності різних наборів контрольних тестів, для яких заздалегідь відомий результат. Тобто тестування припускає виконання програми й одержання конкретних результатів виконання тестів [24–26].

Тести підбираються так, щоб вони охоплювали як найбільше типів ситуацій алгоритму програми. Менш тверда вимога – виконання хоча б один раз кожної гілки програми.

Історично першим видом тестування було налагодження.

Налагодження – це перевірка опису програмного об'єкта на МП з метою виявлення в ньому помилок і подальшого їхнього усунення. Помилки виявляються компіляторами при їхньому синтаксичному контролі. Після цього проводиться верифікація з перевірки правильності функцій коду і валідація з перевірки відповідності продукту заданим вимогам.

Мета тестування – перевірка виконання реалізованих функцій відповідно до їхньої специфікації. На основі зовнішніх специфікацій функцій і проектної інформації на процесах ЖЦ створюються функціональні тести, за допомогою яких проводиться тестування з урахуванням вимог, сформульованих на процесі аналізу предметної області. Методи функціонального тестування підрозділяються на статичні і динамічні.

8.1. Статичні методи тестування

Статичні методи використовуються при проведенні інспекцій і розгляді специфікацій компонентів без їхнього виконання.

Техніка статичного аналізу полягає в методичному перегляді (або огляді) і аналізі структури програм, а також у доведенні їхньої правильності вручну за столом. Статичний аналіз направлений на аналіз документів, розроблених на всіх процесах ЖЦ і полягає в інспекції вхідного коду і наскрізного контролю програми.

Інспекція ПС – це статична перевірка відповідності програми заданим специфікаціями, проводиться шляхом аналізу різних представлень результатів проектування (документації, вимог, специфікацій, схем або коду програм) на процесах ЖЦ. Перегляди й інспекції результатів проектування і відповідності їх вимогам замовника забезпечують більш високу якість створюваних ПС.

При інспекції програм розглядаються документи робочого проектування на процесах ЖЦ разом з незалежними експертами й учасниками розробки ПС. На початковому процесі проектування інспекція припускає перевірку повноти, цілісності, однозначності, несуперечності і сумісності документів з вимогами до програмної системи. На процесі реалізації системи під *інспекцією* розуміють аналіз текстів програм на дотримання вимог стандартів і прийнятих керівних документів технології програмування.

Ефективність такої перевірки полягає в тому, що залучені експерти намагаються подивитися на проблему «з боку» і піддають її всебічному критичному аналізу.

Ці прийоми дозволяють на більш ранніх процесах проектування знайти помилки або недоробки шляхом багаторазового перегляду вхідного опису програми. Символьне тестування застосовується для перевірки окремих ділянок програми на вхідних символьних значеннях.

Крім того, розробляється безліч нових засобів автоматизації символьного виконання програм. Наприклад, автоматизований засіб статичного контролю для мовно-орієнтованої розробки, інструменти автоматизації доведення коректності й автоматизований апарат мереж Петрі.

8.2. Динамічні методи тестування

Динамічні методи тестування використовуються в процесі виконання програм. Вони базуються на графовій структурі, що пов'язує причини помилок з очікуваними реакціями на них. У процесі тестування накопичується інформація про помилки, що використовується при оцінці показників надійності і якості ПС.

Динамічне тестування орієнтоване на перевірку коректності ПС на множині тестів, що проганяються по ПС, з урахуванням зібраних даних на процесах ЖЦ, проведення виміру окремих показників (число відмов, збоїв) тестування для оцінки характеристик якості, зазначених у вимогах, шляхом виконання системи на ЕОМ. Тестування ґрунтується на систематичних, статистичних, (імовірнісних) і імітаційних методах. Охарактеризуємо їх

Систематичні методи тестування поділяються на методи, у яких програма розглядається як «чорна скринька» (використовується інформація про розв'язувану задачу), і методи, у яких програма розглядається як «біла скринька» з використанням структури програми. Цей вид називають тестуванням з керуванням за даними або керуванням на вході-виході. Ціль – з'ясування обставин, при яких поводження програми не відповідає її специфікації. При цьому кількість виявлених помилок у

Ціль динамічного тестування програм за принципом «чорної скриньки» – виявлення одним тестом максимального числа помилок з використанням невеликої підмножини можливих вхідних даних.

Методи «чорної скриньки» забезпечують:

– еквівалентне розбиття;

– аналіз граничних значень;

– застосування функціональних діаграм, що в поєднанні з реверсивним аналізом дають досить повну інформацію про функціонування тестованої програми.

Еквівалентна розбивка складається з розділу вхідної області даних програми на скінченне число класів еквівалентності так, щоб кожен тест, що є представником деякого класу, був еквівалентний будь-якому іншому тесту цього класу.

Класи еквівалентності виділяються шляхом перебору вхідних умов і розбивки їх на дві групи або більше. При цьому розрізняють два типи класів еквівалентності: правильні, що задають вхідні дані для програми, і неправильні, засновані на завданні помилкових вхідних значень.

Розроблення тестів методом еквівалентного розбиття здійснюється в два етапи: виділення класів еквівалентності і побудова тестів. При побудові тестів, заснованих на виборі вхідних даних, проводиться символьне виконання програми.

Отже, методи тестування за принципом «чорної скриньки» використовуються для тестування функцій, реалізованих у програмі, шляхом перевірки невідповідності між реальною поведінкою функцій і очікуваною поведінкою з урахуванням специфікацій вимог. Під час підготовки до цього тестування будуються таблиці умов, причинно-наслідкового графа й області розбиття. Крім того, готуються тестові набори, що враховують параметри й умови середовища, які впливають на поведінку функцій. Для кожної умови визначається множина значень і обмежень предикатів, за якими тестується програма.

Метод «білої скриньки» дозволяє досліджувати внутрішню структуру програми, при чому виявлення всіх помилок у програмі є критерієм вичерпного тестування маршрутів потоків (графа) передач керування, серед яких розглядають:

а) критерій покриття операторів – набір тестів у сукупності повинен забезпечити проходження кожного оператора не менше ніж один раз;

б) критерій тестування областей (відомий як покриття рішень або переходів) – набір тестів у сукупності повинен забезпечити проходження кожної гілки і виходу, принаймні, один раз.

Критерій «б» відповідає простому структурному тесту і найбільш розповсюджений на практиці. Для задоволення цього критерію необхідно побудувати систему шляхів, що містить у собі усі області програми. Перебування такого оптимального покриття в деяких випадках здійснюється просто, а в інших є більш складною задачею.

Тестування за принципом «білої скриньки» орієнтовано на перевірку проходження всіх віток програм за допомогою застосування шляхового й імітаційного тестування.

Шляхове тестування застосовується на рівні модулів і графової моделі програми з вибором тестових ситуацій, підготовки даних і містить у собі тестування наступних елементів:

– операторів, що повинні бути виконані хоча б один раз, без обліку помилок, що можуть залишитися в програмі через велику кількість логічних шляхів і необхідності проходження підмножин цих шляхів;

– шляхів по заданому графу потоків керування для виявлення різних маршрутів передачі керування за допомогою шляхових предикатів, для обчислення якого створюється набір тестових даних, що гарантують проходження всіх шляхів. Проте усі шляхи протестувати неможливо, тому залишаються не виявлені помилки, що можуть виявитися в процесі експлуатації;

– блоків, що розділяють програми на окремі дрібні блоки, які виконуються хоча б один раз або багаторазово при проходженні через шляхи програми, що вміщують сукупність операторів реалізації однієї функції, або на вхідній множині даних, що буде використовуватися при виконанні зазначеного шляху.

«Біла скринька» базується на структурі програми, у випадку ж «чорної скриньки» про структуру програми нічого невідомо. Для виконання тестування за допомогою цих «скриньок» відомими вважаються виконувані функції, входи (вхідні дані) і виходи (вихідні дані), а також логіка обробки програми і опису документації.

8.3. Функціональне тестування

Мета функціонального тестування – виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації і вимог. Функціональні тести повинні охоплювати всі реалізовані функції з урахуванням найбільш ймовірних типів помилок. Тестові сценарії, що поєднують окремі тести, орієнтовані на перевірку якості розв'язку функціональних задач.

Функціональні тести створюються за зовнішніми специфікаціями функцій, проектної інформації і за текстом на МП, що стосуються його функціональних характеристик і застосовуються на процесі комплексного тестування й іспитів для визначення повноти реалізації функціональних задач і їхньої відповідності вхідним вимогам.

До задач функціонального тестування належать:

– ідентифікація множини функціональних вимог;

– ідентифікація зовнішніх функцій і побудова послідовностей функцій відповідно до їхнього використання в ПС;

– ідентифікація множини вхідних даних кожної функції і визначення областей їхньої зміни;

– побудова тестових наборів і сценаріїв тестування функцій;

– виявлення і подання усіх функціональних вимог за допомогою тестових наборів і проведення тестування помилок у програмі і при взаємодії із середовищем.

Тести, створювані за проектною інформацією, пов'язані зі структурами даних, алгоритмами, інтерфейсами між окремими компонентами і застосовуються для тестування компонентів і їхніх інтерфейсів. Основна мета – забезпечення повноти і погодженості реалізованих функцій і інтерфейсів між ними.

В основу комбінованого методу «чорної скриньки» і «білої скриньки» покладено розбивку вхідної області функції на підобласті виявлення помилок. Підобласть містить у собі однорідні елементи, які обробляються коректно або

некоректно. Для тестування підобласті застосовується виконання програми на одному з елементів цієї області.

Передумови функціонального тестування:

– коректне оформлення вимог і обмежень до якості ПС;

– коректний опис моделі функціонування ПС у середовищі експлуатації замовника;

– адекватність моделі ПС заданому класу.

8.4. Інфраструктура перевірки правильності програмних систем

Під *інфраструктурою* перевірки правильності (доведення, верифікації і тестування) програмних систем розуміють інтегрований набір загальнодоступних технічних, технологічних і методологічних ресурсів, що знаходяться у розпорядженні команди розробників, верифікаторів і тестувальників, які виконують роботи з розроблення правильної системи за договорами із організаціями-замовниками.

Команда виконує дії з підготовки і проведення таких задач:

– виділення об'єктів перевірки на процесах ЖЦ та на завершальному процесі тестування;

– аналіз і класифікація помилок для розглянутого класу програм, що перевіряються;

– підготовка даних для верифікації правильності виконання функцій;

– підготовка даних і тестів для їхнього виконання і пошуку різного роду помилок і відмовлень у компонентах і в системі в цілому;

– розроблення завдань підгрупами команди з проведення і керування процесом досягнення правильної програми;

– підготовка до перевірки виконання вимог до ПС і до системи;

– аналіз результатів верифікації і тестування системи, отриманих підгрупами. Об'єкти процесу – компоненти, групи компонентів, підсистеми і система. Для кожного з них формується стратегія проведення верифікації і тестування. Якщо об'єкт готовий і належить до «білої скриньки» або до «чорної скриньки», склад компонентів якого невідомий, то верифікація функцій проводиться зі спеціально підготовленими даними перевірки функцій і вхідних тестових даних для тестування й отримання вихідних даних. Головна мета верифікації перевірити правильність виконання функцій.

Стратегічна мета тестування – переконатися, що кожен розглянутий вхідний набір даних відповідає очікуваним вихідних даним. Проектувальник тестів повинен заглянути усередину «чорної скриньки» і досліджити деталі процесів обробки даних, питання забезпечення захисту і відновлення даних, а також інтерфейси з іншими програмами і системами. Це сприяє підготовці тестових даних для проведення тестування. Для деяких типів об'єктів група інженерії тестування не може згенерувати представницьку безліч тестових наборів, що демонстрували б функціональну правильність роботи компонентів при всіх можливих наборах тестів.

Тому кращим є метод «білої скриньки», при якому можна використовувати логічну структуру об'єкта для організації перевірки програми в різних її областях. Наприклад, можна виконати верифікацію функцій і підготувати тестові набори, що

проходять через всі оператори або всі контрольні точки компонента для того, щоб переконатися в отриманні правильних результатів.

8.4.1. Класифікація помилок і методи їхнього пошуку

Міжнародний стандарт ANSI/IEEE–729–83 розділяє всі помилки в розробці програм на такі типи.

Помилка (error) – стан програми, при якому видаються неправильні результати, причиною яких є недоліки (flaw) в операторах програми або в технологічному процесі її розроблення, що приводить до неправильної інтерпретації вихідної інформації, отже, і до невірного розв'язку.

Дефект (fault) у програмі – наслідок помилок розробника на кожному з процесів проектування, що може утримуватися у вхідних або проектних специфікаціях, текстах кодів програм, експлуатаційній документації тощо. У процесі виконання програми можуть бути виявлені дефект або збій.

Відмова (failure) – це відхилення програми від функціонування або неможливість програми виконувати функції, визначені вимогами й обмеженнями, що розглядається як подія, яка сприяє переходу програми в непрацездатний стан через помилки, приховані у ній дефекти або збої у середовищі функціонування [6, 11]. Відмова може бути за таких причин:

– помилкова специфікація або пропущена вимога, яка означає, що специфікація точно не відбиває того, що припускав користувач;

– специфікація може містити у собі вимогу, яку неможливо виконати на даній апаратурі і програмному забезпеченні;

– проект програми може містити у собі помилки (наприклад, база даних спроектована без засобів захисту від несанкціонованого доступу користувача, а потрібен захист);

– програма може бути неправильною, тобто вона виконує невластивий алгоритм або він реалізований не цілком.

Таким чином, відмова, як правило, є результатами однієї помилки або більше у програмі, а також наявності різного роду дефектів.

Загальні класи помилок. Усі помилки, що виникають у програмах, розподіляють на такі класи [14, 26]:

– логічні і функціональні помилки;

– помилки обчислень і часу виконання;

– помилки вводу-виводу і маніпулювання даними;

– помилки інтерфейсів;

– помилки обсягу даних і ін.

Логічні помилки – наслідок порушення логіки алгоритму, внутрішньої непогодженості змінних і операторів, а також мовних правил програмування. Функціональні помилки – наслідок неправильно визначених функцій, порушення порядку їхнього застосування або відсутності повноти їхньої реалізації і т.д.

Помилки обчислень виникають через неточність вхідних даних і реалізованих формул, похибок методів, неправильного застосування операцій обчислень. Помилки часу виконання зв'язані з відсутністю необхідної швидкості обробки запитів, або часу виконання або відновлення програми.

Помилки вводу-виводу і маніпулювання даними є наслідком неякісної підготовки даних для виконання програми, збоїв при занесенні їх у базу даних або при вибірці з неї.

Помилки інтерфейсу належать до помилок взаємозв'язку окремих елементів одного з одним, що виявляється при передачі даних між ними, а також при взаємодії із середовищем функціонування.

Помилки обсягу належать до даних і є наслідком того, що реалізовані методи доступу і розміри баз даних не задовольняють реальні обсяги інформації системи або інтенсивності їхньої обробки.

Наведені основні класи помилок властиві різним типам компонентів ПС і виявляються вони в програмах по-різному. Так, при роботі з БД виникають помилки подання і маніпулювання даними, логічні помилки в завданні прикладних процедур обробки даних та ін. У програмах обчислювального характеру переважають помилки обчислень, а в програмах керування й обробки – логічні і функціональні помилки. У ПС, що складається з багатьох різномовних програм, які реалізують різні функції, можуть міститися помилки різних типів. Помилки інтерфейсів і порушення обсягу характерні для будь-якого типу ПС.

Аналіз типів помилок у програмах є необхідною умовою створення планів і методів тестування для забезпечення правильності ПС.

На сучасному процесі розвитку засобів підтримки розробки ПС (CASE–технології, інструменти) при проектуванні ПС захищається від найбільш типових помилок і запобігається поява дефектів.

Фірма IBM розробила підхід до класифікації помилок, названий ортогональною класифікацією дефектів [21]. При такому підході розподіл помилок за категоріями робить відповідальний розробник. Класифікація не залежить від продукту, організації розробки, вона може застосуються до всіх процесів розроблення ПС різного призначення. Відповідно до даної класифікації в табл. 8.1 наведено список помилок.

Таблиця 8.1. Ортогональна класифікація дефектів IBM

Контекст помилки	**Класифікація дефектів**
Функція	Помилки інтерфейсів кінцевих користувачів ПС, викликані апаратурою або зв'язані з зовнішніми структурами даних
Інтерфейс	Помилки у взаємодії з іншими компонентами, у викликах, макросах, що керують блоками або в списку параметрів
Логіка	Помилки в програмній логіці, неохопленій валідацією, а також у використанні значень змінних
Присвоювання	Помилки в структурі даних або в ініціалізації змінних окремих частин програми
Зациклення	Помилки, викликані ресурсом часу, реальним часом або розподілом часу
Середовище	Помилки в репозитарії, у керуванні змінами або в контрольованих версіях проекту
Алгоритм	Помилки, пов'язані з забезпеченням ефективності, коректності алгоритмів або структур дані системи
Документація	Помилки в записах документів супроводу або в публікаціях

Передбачено ситуації, коли знайдена неініційована змінна або ініційованій змінній привласнене неправильне значення.

Ортогональність схеми класифікації полягає в тому, що будь-який її термін належить тільки до однієї категорії. Іншими словами, помилка, що простежується в системі, повинна знаходитися в одному з класів, що дає можливість різним розробникам класифікувати помилки однаковим способом.

Фірма Hewlett–Packard використовувала класифікацію Буча, встановивши відсоткове співвідношення помилок, що виявляються в ПС на різних стадіях розробки (рис. 8.1.).

Це співвідношення, типове для багатьох фірм, що роблять ПС, має деякі відхилення від інших.

Згідно з даними [32] вартість аналізу і формування вимог, внесення до них змін становить приблизно 10%, аналогічно оцінюється вартість специфікації продукту. Вартість кодування оцінюється більше ніж 20%, а вартість тестування продукту дорівнює більш ніж 45% його загальної вартості. Значну частину вартості становить супровід готового продукту і виправлення виявлених у ньому помилок.

Дослідження фірм IBM показали, чим пізніше виявляється помилка в програмі, тим дорожче коштує її виправлення, ця залежність близька до експонентної. Так, військово-повітряні сили США оцінили вартість розробки однієї інструкції в 75 доларів, а вартість супроводу дорівнює близько 4000 доларів.

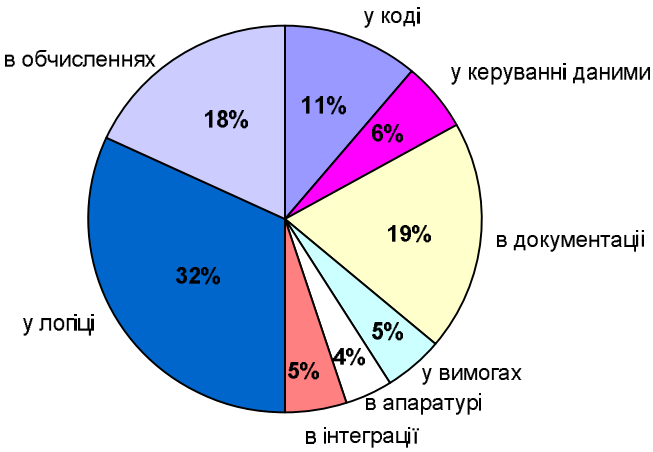

Рис. 8.1. Відсоткове співвідношення помилок при розробці ПС

Зв'язок помилки з відмовою. Наявність помилки в програмі, як правило, приводить до відмови ПС при його функціонуванні. Для аналізу причинно-наслідкових зв'язків «помилка-відмова» виконуються такі дії:

– ідентифікація помилок у технологіях проектування і програмування;

– взаємозв'язок помилок процесу проектування і помилок, що допускаються людиною;

– класифікація відмов, помилок і можливих помилок, а також дефектів на кожному процесі розробки;

– зіставлення помилок людини, що допускаються на визначеному процесі розробки, і дефектів в об'єкті, як наслідків помилок специфікації проекту, моделей програм;

– перевірка і захист від помилок на всіх процесах ЖЦ, а також виявлення дефектів на кожному процесі розробки;

– зіставлення дефектів і відмовлень у ПС для розробки системи взаємозв'язків і методики локалізації, збирання й аналізу інформації про відмови і дефекти;

– розробка підходів до процесів документування й супроводження ПС.

Кінцева мета причинно-наслідкових зв'язків «помилка-відмова» полягає у визначенні методів і засобів тестування і виявлення помилок визначених класів, а також критеріїв завершення тестування на множині наборів даних; у визначенні шляхів удосконалювання організації процесу розроблення, тестування і супроводу ПС.

Наведемо таку класифікацію типів відмов:

– апаратна, при якому загальносистемне ПС не працездатне;

– інформаційна, викликана помилками у вхідних даних і передачі даних по каналах зв'язку, а також при збоях пристроїв вводу, як наслідок апаратних відмов;

– програмна, при наявності помилок у компонентах системи;

– ергономічна, викликана помилками оператора при його взаємодії з комп'ютером (це відмовлення – вторинна відмова, може привести до інформаційної або функціональної відмови).

Деякі помилки можуть бути, з одного боку, наслідком недоробок при визначенні вимог, проекту, генерації вихідного коду або документації, з іншого боку, вони виникають в процесі розробки програми або інтерфейсів окремих елементів програми (порушення порядку параметрів і т.п.).

8.4.2. Процес тестування за життєвим циклом

Наведені типи помилок розподіляються за процесами ЖЦ і їм відповідають такі джерела їхнього виникнення [23, 24]:

– ненавмисне відхилення розробників від робочих стандартів або планів реалізації;

– специфікації функціональних і інтерфейсних вимог виконані без дотримання стандартів розробки, що призводить до порушення функціонування програм;

– організації процесу розробки – недосконале або недостатнє управління керівником проекту ресурсами (людськими, технічними, програмними і т.д.) і питаннями тестування й інтеграції елементів проекту.

Розглянемо процес тестування, виходячи з рекомендацій стандарту ISO/IEC–12207, і наведемо типи помилок, що виявляються під час кожного процесу ЖЦ.

Процес розробки вимог. При визначенні вихідної концепції системи і вихідних вимог до системи виникають помилки аналітиків при специфікації вищого рівня системи і побудові концептуальної моделі предметної області.

Характерними помилками цього процесу є:

– неадекватність специфікації вимогам кінцевих користувачів;

– некоректність специфікації взаємодії ПС із середовищем функціонування або з користувачами;

– невідповідність вимог замовника окремим і загальним властивостям ПС; – некоректність опису функціональних характеристик;

– незабезпеченість інструментальними засобами всіх аспектів реалізації вимог замовника й ін.

Процес проектування. Помилки при проектуванні компонентів можуть бути наслідком недоліків в описі алгоритмів, логіки керування, структур даних, інтерфейсів, логіки моделювання потоків даних, форматів вводу-виводу та ін. В основі цих помилок лежать дефекти специфікацій заданих аналітиками і недоробки проектувальників. До них належать помилки, пов'язані з:

– погодженістю інтерфейсу користувача із середовищем;

– описом функцій (неадекватність цілей і задач компонентів, що виявляються при перевірці комплексу компонентів);

– визначенням процесу обробки інформації і взаємодії між процесами (результат некоректного визначення взаємозв'язків компонентів і процесів);

– некоректним завданням даних і їхніх структур при описі окремих компонентів і ПС у цілому;

– некоректним описом алгоритмів модулів;

– визначенням умов виникнення можливих помилок у програмі;

– порушенням прийнятих для проекту стандартів і технологій.

Процес кодування. На даному процесі виникають помилки, що є результатом дефектів проектування, помилок програмістів і менеджерів у процесі розроблення і налагодження системи. Причиною помилок є:

– безконтрольність значень вхідних параметрів, індексів масивів, параметрів циклів, вихідних результатів та ін.;

– неправильна обробка нерегулярних ситуацій при аналізі кодів повернення від викликуваних підпрограм, функцій і ін.;

– порушення стандартів кодування (погані коментарі, нераціональне виділення модулів і компонентів та ін.);

– використання одного імені для позначення різних об'єктів або різних імен одного об'єкта, погана мнемоніка імен;

– непогоджене внесення змін у програму різними розробниками та ін.

Процес тестування. На цьому процесі помилки допускаються програмістами і тестувальниками при виконанні технології збирання і тестування, вибору тестових наборів і сценаріїв тестування та ін. Відмови в програмному забезпеченні, викликані такого роду помилками, повинні виявлятися, усуватися і не впливають на статистику помилок компонентів і на програмне забезпечення в цілому.

Процес супроводу. На процесі супроводу виявляються помилки, причиною яких є недоробки і дефекти експлуатаційної документації, недостатні показники кодифікованості й легкості читання, а також некомпетентність осіб, відповідальних за супровід і/або удосконалення ПС. Залежно від сутності внесених змін на цьому процесі можуть виникати практично будь-які помилки, аналогічні раніше перерахованим помилкам на попередніх процесах.

Джерела помилок. Помилки можуть бути виникнути в процесі розроблення проекту, компонентів, коду і документації. Як правило, вони виявляються при виконанні або супроводі програмного забезпечення в найбільш несподіваних і різних її точках.

Причиною появи помилок є – нерозуміння вимог замовника; неточна специфікація вимог у документах проекту та ін. Це приводить до того, що реалізуються деякі функції системи, що будуть працювати не так, як пропонує замовник. У зв'язку з цим проводиться спільне обговорення замовником і розробником деяких деталей вимог для їхнього уточнення.

Команда розробників системи може також змінити мову опису системи. Деякі помилки можуть бути не виявлені (наприклад, неправильно задані індекси або значення змінних цих операторів).

Визначення тесту. Для перевірки правильності програм спеціально розробляються тести і тестові дані. Під *тестом* розуміється деяка програма, призначена для перевірки працездатності іншої програми і виявлення в ній помилкових ситуацій. Тестову перевірку можна провести також шляхом введення в програму, які перевіряється, операторів, які будуть сигналізувати про хід її виконання й отримання результатів.

Тестові дані слугують для перевірки роботи системи і складаються різними способами: генератором тестових даних, проектною групою на основі документів або наявних файлів, користувачем з специфікаціях вимог та ін. Дуже часто розробляються спеціальні форми вхідних документів, у яких відображається процес виконання програми за допомогою тестових даних [23–25, 31].

Створюються тести, що перевіряють:

– повноту функцій;

– погодженість інтерфейсів;

– коректність виконання функцій і правильність функціонування системи в заданих умовах;

– надійність виконання системи;

– захист від збоїв апаратури і не виявлених помилок та ін.

Тестові дані готуються як для перевірки окремих програмних елементів, так і для груп програм або комплексів на різних стадіях процесу розроблення.

Багато типів тестів готуються замовником для перевірки роботи програмної системи. Структура і зміст тестів залежать від виду елемента тестування, яким може бути модуль, компонент, група компонентів, підсистема або система. Деякі тести залежать від мети і необхідності знати: чи працює система відповідно до її проекту, чи задоволені вимоги і чи бере участь замовник у перевірці роботи тестів тощо.

Залежно від задач, що ставляться перед тестуванням програм, складаються тести перевірки проміжних результатів проектування елементів на процесах ЖЦ, а також створюються тести іспитів остаточного коду системи.

Тестування інтегрованої системи. Тести для перевірки окремих елементів системи і тести інтегрованої системи мають загальні і відмінні риси. Як приклад розглянемо схему інтеграції компонентів.

На першому рівні схеми знаходяться, наприклад, компоненти A, B, D, на другому рівні – E, C, G. Вони пов'язані між собою інтерфейсом (рис.8.2).

Кожен компонент схеми тестується окремо від інших компонентів тестами, що містять у собі набори даних і сценаріїв, складені відповідно до їхніх типів і функцій, специфікованих у вимогах до системи. Тестування проводиться в контрольному операційному середовищі на заданій безлічі тестових даних і операцій, розроблених з ними.

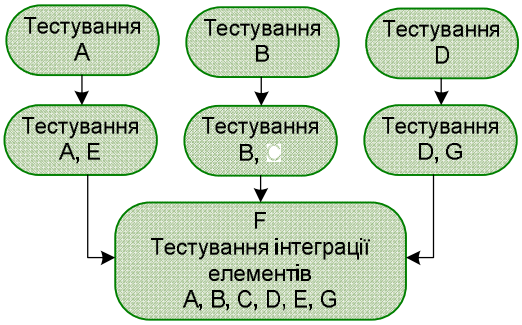

Рис. 8.2. Схема тестування інтегрованих компонентів

Тести перевіряють внутрішню структуру, логіку і граничні умови виконання кожного компонента. Спочатку тестуються компоненти *A, B*, *D* незалежно один від одного і кожний з окремим тестом. Після їхньої перевірки виконується перевірка інтерфейсів для зв'язку компонентів другого рівня: *A* → *E*, *B* → *C*, *D* → *G,* а потім вже компоненти *E*, C, *G*.

Компоненти й інтерфейси інтегруються і утворюють компонент *F,* він перевіряється на правильність інтеграції і функцій.

При тестуванні можуть виникати помилки. Вони, зазвичай, – результат неправильного завдання параметрів в операторах виклику або помилок в алгоритмі обчислення процедур або функцій. Помилки, що виникають у в зв'язках, усуваються, а потім повторно перевіряється зв'язок з компонентом *F* у вигляді трійки: компонент–інтерфейс–компонент.

Наступний крок тестування комплексної системи – перевірка функціонування системи за допомогою тестів перевірки функцій і вимог до них. Після цього перевіряється комплекс на виконавчих і іспитових тестах відповідно до вимог до ПС, апаратури і виконуваних функцій. Іспит системи проводиться в реальному середовищі, у якому система буде функціонувати надалі.

Приклад. Оцінювання часу тестування та ризиків відмов модулів.

Нехай ми маємо приклад деякої системи інформаційно-аналітичної підтримки прийняття управлінських рішень із модулів [31, 33] (табл. 8.2.).

Вони функціонують у розподіленому середовищі Oracle RunTime та MS Office (Word, Excel).

Таблиця 8.2. Склад програмних комплексів ПС **Шифр ПК**

	Призначення
ПК1	Підготовка вхідних даних
ПК2	Ведення діловодства і контролю виконавської діяльності
ПК3	Контроль і введення регламентованої інформації до БД
ПК4	Надання довільних та регламентованих довідок
ПК5	Формування звітної доповіді
ПК6	Діагностична експертиза
ПК7	Моніторинг стану діяльності ЗСУ

Найбільший внесок у ризик відмов робить ПК3 контролю і введення даних до БД Oracle, який функціонує на 10 робочих місцях. Його головне призначення – контроль даних у формах, підготовлених за допомогою ПК1 та їх експорт до БД.

ПК3 виконує запис у більш як 90 таблиць БД, використовує понад 50 нормативно-довідкових таблиць та класифікаторів.

На процесі проектування для цього ПК був виконаний аналіз можливих сценаріїв функціонування (рис. 8.3) за декількома способами використання.

Таблиця 8.3. Функції модулів, що надають найбільший внесок у відмови ПС

Модуль	**Функції**
М1. Реєстрація та імпорт	Реєстрація документів, їх форм і даних про стан комплексу. Перехід у середовище Excel для опрацювання форм
М2. Контроль стану	Контроль опрацювання форм. Визначення ступеню повноти завантаження даних з форм за кожним надісланим документом (повністю, не повністю, не виконане)
М3. Експорт	Завантаження даних з форми до БД (підключення до сервера БД, параметризація збережених процедур, завантаження)
М4. Контроль	Контроль правильності наданих форм у Excel. Синтаксичний контроль форми (типи та формати даних, коди класифікаторів тощо) та семантичний контроль (непротирічність даних, відповідність обмеженням)
М5. Запити до БД	Запити до БД з метою виключення можливого дублювання інформації, яка надходить у формах з різних джерел

Функції кожного з модулів М1-М5 дійсно є критичними для ПС, оскільки від їх безвідмовної роботи залежить цілісність системи.

Для реєстрації часу виконання t та моментів відмов у модулі М1-М5 на час тестування були вбудовані відповідні фрагменти коду, що мали моменти початку та завершення (нормального або аварійного) роботи модуля та їх реєстрація у журналі подій і відмов.

Очікуваний час t_0 використання кожного модуля при експлуатації ПС та їхні внески $C_м$ визначалися тижневе, місячно, а також – частоти звернення до модулів у кожному сеансі роботи ПК3. Отримані дані про відмови та оцінені параметри моделі надійності наведені в табл. 8.4.

Вартість тестування і усунення відмов розраховувалася за такими чинниками:

– вартості часу роботи фахівців (тестувальників та розробників); –

визначеного реального часу виконання кожного модуля під час тестування та часу, витраченого на усунення дефектів.

Таблиця 8.4. Оцінки параметрів моделі надійності для п’яти модулів ПК3

Модуль	**Кількість дефектів**	**Коефіцієнт пропорційності**
М1. Реєстрація та імпорт	42.8	0.000082
М2. Контроль стану	43.1	0.000322
М3. Експорт	56.7	0.000076
М4. Контроль	46.6	0.00083
М5. Запити до БД	39.7	0.00017

Для модулів були встановлені значення різних даних з процесу тестування (табл. 8.5.), а саме,

– вартість одиниці часу тестування $c_1 = 0.8$ грн.

– вартість усунення дефекту $c_2 = 60$ грн.

У цієї таблиці наведені отримані оцінки з часу функціонування модулів та даних про ризик та оптимальний час тестування.

Таблиця 8.5. Оцінки часу використання, внесків, ризику та оптимального часу тестування модулів ПК3

Модуль	**Час використання** t_0	**Внесок (грн)** $См(t_0)$	**Очікувана кількість відмов** $\mu(t_e)$	**Ризик модуля** $R\ (t_0)$	**Оцінка** t^*
М1. Реєстрація та імпорт	160	25000	0.58625	14656.14	1699.54
М2. Контроль стану	100	5000	1.367	6835.33	1668.99
М3. Експорт	160	29000	0.685296	19872.61	4341.8
М4. Контроль	160	10000	1.14874	11487.4	3381.91
М5. Запити до БД	100	16000	0.74306	7440.63	488.246

Наведені таблиці з тестування групи програмних компонентів (ПК1–ПК7) демонструють послідовний процес аналізу і оброблення інформації при їхньому тестуванні і оцінюванні результатів.

8.4.3. Інженерія керування тестуванням

За функціональні і системні тести несуть відповідальність розробник і замовник, останній більше впливає на складання тестів для випробувань системи [26, 28, 32].

Цей процес реалізує група тестувальників, що не залежать від групи розробників ПС. Її очолює керівник групи, який повинен мати:

– досвід в області тестування;

– здатність бути лідером і керувати групою тестувальників;

– знання з задач предметної області (і програмного продукту);

– знання з інфраструктури (апаратного і системного програмного забезпечення).

Рядовий тестувальник повинен знати:

– галузь виробництва продуктів/технологій створення ПС;

– елементи інфраструктури розроблення ПС;

– вимоги до системи і стандарти тестування;

– підходи до використання робочих продуктів процесу тестування; –

інструменти і стратегії тестування;

– вміти аналізувати результати і підбирати нові тестові дані або додавати дані для оцінювання процесу тестування.

Деякі члени цієї групи – досвідчені фахівці або навіть професіонали в цій галузі. До них також належать аналітики, програмісти, що працюють в галузі розроблення систем від її початку. Вони мають справу не тільки зі специфікаціями, а й з методами і засобами проектування, тестування, організують створення і

виконання тестів. Із самого початку тестувальники складають плани тестування, тестові дані, сценарії, а також графіки виконання тестів.

Професійні тестувальники працюють разом із групою керування конфігурацією, щоб забезпечити їх документацією й іншими механізмами для зв'язку між собою тестів і вимог проекту, конфігурації і коду. Вони розробляють методи і процедури тестування. У цю команду включаються додаткові фахівці, що ознайомлені з вимогами системи або з підходами до її розробки. Аналітики входять до складу команди, тому що вони розуміють проблеми визначення специфікацій замовників.

Багато фахівців порівнюють тестування системи зі створенням нової системи, у якій аналітики визначають потреби і цілі замовника, працюючи разом із проектувальниками і намагаючись реалізувати ідеї і принципи роботи системи.

Проектувальники системи повідомляють групі тестувальників проектні цілі, щоб вони знали декомпозицію системи на підсистеми і її функції, а також принципи роботи. Після проектування тестів група тестувальників проводить аналіз можливостей системи.

Оскільки тести і тестові сценарії є прямим відображенням вимог до проекту в цілому, перспективи керування конфігурацією системи визначаються саме цією групою. Зміни, що виявляються в програмі, помилки в системі відбивають у документації, вимогах, проекті, а також в описах вхідних і вихідних даних або в інших розроблюваних артефактах. Внесені зміни в процесі розроблення призводять до модифікації тестових сценаріїв або більшою мірою до зміни планів тестування. Фахівці з керування конфігурацією враховують ці зміни і координують складання тестів з її урахуванням.

До групи тестувальників входять також користувачі. Вони оцінюють отримувані результати, зручність використання, а також висловлюють свою думку про принципи роботи системи.

Уповноважені замовника планують роботи доти, поки використовується і супроводжується система. При цьому вони можуть привнести деякі зміни в проект через неповноту заданих вимог і сформулювати системні вимоги для проведення верифікації системи і прийняття рішень про її готовність і корисність.

Планування тестування. Для проведення тестування розробляється план (Test Plan), у якому описуються стратегії, ресурси і графік тестування окремих компонентів і системи в цілому. У плані визначаються роботи для різних членів команди, що виконують свої ролі в цьому процесі. План містить у собі також визначення ролі тестів у кожнім процесі, ступінь покриття програми тестами і відсоток тестів, що виконуються зі спеціальними даними.

Тестові інженери створюють тестові сценарії (Test Cases), кожний з яких перевіряє результат взаємодії між актором і системою на основі перед- і постумов використання таких сценаріїв. Сценарії в основному належать до тестування за типом «білої скриньки» і орієнтовані на перевірку структури й операцій інтеграції компонентів системи.

Для проведення тестування тестові інженери пропонують процедури тестування (Test Procedures), що вміщують валідацію об'єктів і верифікацію тестових сценаріїв відповідно до плану графіку. Оцінка тестів (Test Evaluation) полягає в оцінці результатів тестування, ступеня покриття програм сценаріями і

статусу отриманих помилок. На рис. 8.3. наведено коло обов'язків інженера тестувальника.

Тестувальник інтегрованої системи перевіряє інтерфейси і дає оцінку виконання відповідних системних тестів, а потім аналізує результати тестування.

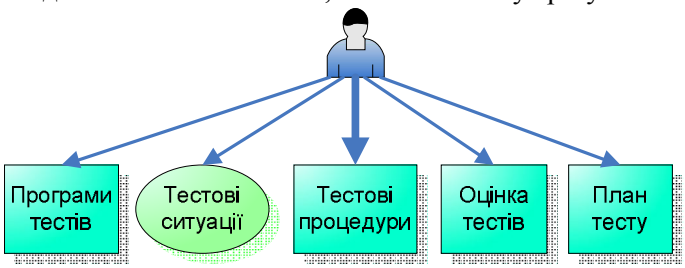

Рис. 8.3. Схема відповідальності інженера-тестувальника

При виконанні цих тестів, як правило, знаходяться дефекти з причини глибоко захованих недоліків у програмах, що виявляються при тривалому тестуванні системи на тестових даних.

Керування тестуванням. Усі засоби тестування ПС об’єднуються базою даних, де містяться результати тестування системи, а також компоненти, тестові контрольні дані й інформацію про документування процесу тестування.

База даних проекту підтримується спеціальними інструментальними CASE-засобами, що забезпечують ведення аналізу ПрО, збирання даних про їхні об'єкти, потоки даних тощо. База даних проекту зберігає також початкові й еталонні дані, що використовуються для зіставлення даних, накопичених у базі, з даними, що отримані в процесі тестування системи.

При тестуванні виконуються різні види обчислень характеристик цього процесу за методами планування і керування.

1. Розрахунок тривалості виконання функцій шляхом збирання середніх показників швидкості виконання операторів без виконання програми на машині. Виявляються компоненти, що вимагають тривалого часу виконання в реальному середовищі.

2. Керування виконанням тестування шляхом підбору тестів перевірки, їхнього виконання, селекції результатів тестування і зіставлення їх з еталонними значеннями. Результати даного процесу відображаються на дисплеї, наприклад, гілки виконання у графічній формі, дані про відмови і помилки або конкретні значення вихідних параметрів програми. Ці дані аналізуються розробниками для формулювання висновків про напрями подальшої перевірки правильності програми або їхньому завершенні.

3. Планування тестування призначене для розподілу термінів робіт з тестування, розподілу тестувальник за окремими видами робіт і складання ними тестів перевірки системи. Визначається стратегія і шляхи тестування. У діалозі запитуються дані про реальні значення процесу виконання системи, структури розгалуження вершин графа і параметрах циклів. Перевірені цикли, як правило, вилучаються зі шляхів виконання програми. При плануванні шляхів виконання створюються відповідні тести, критерії і вхідні значення.

4. Результати тестування документуються відповідно до діючого стандарту ANSI/IEEE 829 і містять у собі:

– опис задач, призначення і зміст ПС, а також перелік функцій відповідно до вимог замовника;

– технологію розробки системи;

– плани тестування різних об'єктів, що відповідають технологічним прийомам проведення тестування;

– тести, контрольні приклади, критерії і обмеження, методику оцінки результатів виконання програмного продукту на процесі тестування;

– облік процесу тестування, складання звітів про аномальні події, відмови і дефекти в підсумковому документі системи.

Висновки. Були розглянуті формальні специфікації програм, методи доведення програм за ними, а також сучасні методи і процеси верифікації та тестування ПС, засновані на понятті програм – «біла скринька» і «чорна скринька». Визначено критерії тестування, типи помилок, що виявляються в програмах, а також відмови і помилки на процесах ЖЦ. Сформульовано методи забезпечення процесів отримання правильних програм за допомогою спеціальних груп фахівців, зокрема тестувальників.

Список літератури до Лекцій 6-8

1. *Марков А.А.* Теория алгоритмов // Москва, АН СССР. – 1954. – 231 с.
2. *Ляпунов А.А*. О логических схемах программ// Проблемы кибернетики.– вып.1. – М.: 1958.
3. *Янов Ю.И.* О логических схемах алгоритмов// Проблемы кибернетики.– вып.1. – М.: 1958.
4. *Hoare C.A.R.* Prof of correctness of data representation // Acta Informatica, 1(4).– 271– 287. – 1972. – P. 214–224.
5. *Андерсон Р.* Доказательство правильности программ. – М.: Мир, 1982. – 165 с.
6. *Abrial I.R., Meyer B.* Spesification Language Z. – Boston: Massachusetts Computer Associates Inc., 1979. – 378 p.
7. *Biorner D., Jones C.B*. The Vienna Development Methods (VDM): The Meta – Language. – Vol. 61 of Lecture Notes in Computer Science. – Springer Verlag, Heiderberg, Germany, 1978. – 215 p.
8. *Петренко А.К.* Венский метод разработки программ // Программирование.– 2001. – № 1. – С. 3–23.
9. *The RAISE* Language Group. The RAISE Spesification Language. BCS Practitioner Series. – Prentice Hall, 1982. – 397 p.
10. *The RAISE* Methods Group. The RAISE Development Methods. BCS Practitioner Series. – Prentice Hall, 1985. – 493p.
11. *Агафонов В.Н.* Спецификации программ: понятийные средства и их организация.– Новосибирск: Наука, 1987. – 240 с.
12. *Непомнящий В.А., Сулимов А.А.* Об одном подходе к спецификации и верификации трансляторов. М.: Программирование.– 1983.– № 4. – С. 51–58.
13. *Непомнящий В.А., Шилов Н.В., Бодин Е.В.* Спецификация и верификация распределенных систем средствами языка Elementary–real// М.: Программирование.– 1999. – № 4. – С. 54–67.
14. *Вудкок Д.* Первые шаги к решению проблемы верификации программ// Открытые системы.– 2006.–№8.– С. 36-43.

15. *Hoare T.,* Misra J. Verified software: Theories, Tools, Experiments. Vision of Grant Challenge project.–Microsoft Research Ltd and the University of Texas at Austin, 2005.– 1–43с.
16. *Коваль В.Н.* Концепторные языки. Доказательное проектирование.– Киев.– Наук. думка, 2001.– 182с.
17. *Хоар Ч.*, Лауер П.Е. Непротиворечивые взаимодополняющие теории семантики языков программирования// М.: Мир, 1980.–С.186–221.
18. *Dolores R. Wallase* M. Ippolito, Cuthill B. Reference Information for the Software Verification and Validation Process // NIST Special Publication. – 1996 . – 80p.
19. *Herhart S.L.* Program Verification in the 90's.// Proc. Conf. on Computing in the 1980's, 1978.– P.80–89.
20. *Fei Xie* and James C. Browne. Verified Systems by Composition from Verified Components {feixie, browne}@cs.utexas.edu.
21. *Майерс Г.* Искусство тестирования программ. – Пер.с англ. М.: Финансы и статистика. – 1982. – 176 с.
22. *Иванников В.П., Дышлевый К.В., Мажелей С.Г., Садовская Д.Б., Шебуняев А.Б.* Распределенные объектно-ориентированные среды // М.: Труды ИСП РАН, 2000.–с.84–100
23. *Липаев В.В.* Тестирование программ. – М.: Радио и связь, 1986. – 295 с.
24. *Канер С., Фолк Д., Нгуен Е.К.* Тестирование программного обеспечения: Пер с англ. – Киев: DiaSoft. – 2000. – 544 с.
25. *Weyuker E.J., Ostrand T.J.* Theories of program testing and the application of revealing subdomains // IEEE Trans.Soft.Eng. – 1980. – 6. – №. 3, – P. 236–246.
26. *Андон Ф.И., Коваль Г.И., Лаврищева Е.М., Коротун Т.М., Суслов В.Ю,* Основы инженерии качества программных систем. – Киев: Академпериодика.–Второе изд.– 2007. – 680с.
27. http://research.microsoft.com/specsharp/
28. *ISO/IEC* 12207: 2002. Information technology – Software life cycle processes). Информационные технологи. – Процессы жизненного цикла программного обеспечения.
29. *Соммервил И.* Инженерия программного обеспечения.–6 издание.– Москва–Санкт–Петербург–Киев, 2002.–623 с.
30. *CASE–93.* Proceeding Sixth Intern. // Workshop on Computer Aided Software Engineering. – Singapure. – 1993. – July 19–23. – 418 p.
31. *Лаврищева Е.М., Коротун Т.М.* Построение процесса тестирования программных систем // Проблемы программирования.–2002.–№1–2.– С.272–281.
32. *Бабенко Л.П., Лавріщева К.М.* Основи программноі інженерії. – Киев: Знання, 2001. – 269 с.
33. *Коротун Т.М.* Модели и методы инженерии тестирования программных систем в условиях ограниченных ресурсов. – Автореф. дис. … канд.-физ.-мат. наук. – Киев: Ин-т кибернетики им. В.М. Глушкова НАН Украины, 2005. – 21 с.