МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

ПРОЕКТУВАННЯ, КОНСТРУЮВАННЯ ТА ТЕСТУВАННЯ ПРОГРАМНИХ СИСТЕМ

ВИБРАНІ ЛЕКЦІЇ

з дисципліни

„Технологія проектування програмних систем”

для студентів спеціальності

8.05010202 – системне програмування

денної форми навчання

Затверджено на засіданні кафедри
системного програмування,
протокол №2 від 21.03.2013 р.,
та Методичною радою ЧДТУ,
протокол № 68 від 08.04.2013 р.

Черкаси 2015

Укладачі: Рудницький Володимир Миколайович, д.т.н., професор,
Куницька Світлана Юріївна, к.т.н., ст. викладач,
Бабенко Віра Григорівна, к.т.н., доцент

Рецензент: Голуб С.В., д.т.н. професор

Проектування, конструювання та тестування програмних систем : вибрані лекції з дисципліни «Технологія проектування програмних систем» для студентів спеціальності 8.05010202 «Системне програмування» денної форми навчання/ Уклад. : В.М. Рудницький, С.Ю. Куницька, В.Г. Бабенко ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси : ЧДТУ, 2015. – 44с.

Вибрані лекції присвячені найважливішим питанням щодо теоретичних основ та сучасних підходів конструювання, проектування та тестування програмних систем. Особлива увага приділяється формуванню та вивченню наукових основ мов програмування, теоретичних та практичних методів побудови, тестування програмних систем. Освоєння теоретичних питань, що розкриті у вибраних лекціях, сприятимуть формуванню у студентів високих професійних якостей майбутнього фахівця, оволодінню новітніми технологіями проектування програмних систем в своїй практичній діяльності.

Для студентів спеціальності 8.05010202 «Системне програмування» денної форми навчання.

ЗМІСТ

ВСТУП

Сучасне використання інформаційних технологій потребує вивчення наукових основ мов програмування, практичних методів побудови автоматизованих систем керування, систем автоматизованого проектування та інших сфер використання комп'ютерів.

Метою викладання дисципліни „Технологія проектування програмних систем" є формування у студентів високих професійних якостей майбутнього фахівця, оволодіння новітніми технологіями проектування програмних систем в своїй практичній діяльності. Основу дисципліни складає засвоєння сучасних програмних технологій. Особлива увага надається практичній підготовці в розробці таких програмних систем, зокрема, груповій розробці програмних систем.

Значення дисципліни для реалізації вимог кваліфікаційної характеристики фахівця та вивчення наступних дисциплін полягає в тому, що вивчення дисципліни сприяє формуванню алгоритмічного мислення майбутнього фахівця, створює базу, яка необхідна при вивченні багатьох наступних дисциплін. Виходячи з цього, викладання дисципліни „Технологія проектування програмних систем" підпорядковане вирішенню двох основних задач: засвоюванню сучасних програмних технологій, що використовуються для створення масштабованих систем рівня підприємства; ознайомленню з найбільш поширеними шаблонами проектування, які використовуються для створення таких систем.

Теми викладені у вибраних лекціях сприятимуть формуванню та вивченню наукових основ мов програмування, теоретичних та практичних методів побудови, тестування програмних систем. Освоєння теоретичних питань, що розкриті у вибраних лекціях, сприятимуть формуванню у студентів високих професійних якостей майбутнього фахівця, оволодінню новітніми технологіями проектування програмних систем в своїй практичній діяльності.

РОЗДІЛ 1
КОНСТРУЮВАННЯ ПРОГРАМНИХ СИСТЕМ

ЛЕКЦІЯ 1. ОСНОВНІ ВИЗНАЧЕННЯ, МЕТОДИ, ПРОЦЕДУРИ, ЗАСОБИ, ЦІЛІ ТА СТРАТЕГІЇ ПРОГРАМНИХ СИСТЕМ.

План лекції

1. Визначення технології проектування, методи та засоби програмних систем.

2. Класичний життєвий цикл.

3. Макетування.

1. Визначення технології проектування програмних систем

Технологія проектування програмних систем, надалі конструювання програмного забезпечення (ТКПЗ) – це система інженерних принципів для створення економічного ПЗ, яке надійно і ефективно працює в комп'ютерах.

Розрізняють методи, засоби та процедури ТКПЗ.

Методи забезпечують вирішення наступних задач:

- планування та оцінка проекту;
- аналіз системних і програмних вимог;
- проектування алгоритмів, структур даних і програмних структур;
- кодування;
- тестування;
- супровід.

Засоби (утиліти) ТКПЗ забезпечують автоматизовану або автоматичну підтримку методів. З метою спільного застосування утиліти можуть об'єднуватися в системи автоматизованого конструювання ПЗ. Такі системи прийнято називати CASE-системами. Абревіатура CASE розшифровується як Computer Aided Software Engineering (програмна інженерія з комп'ютерною підтримкою).

Процедури є «клеєм», який з'єднує методи і утиліти так, що вони забезпечують безперервний технологічний ланцюжок розробки. Процедури визначають:

- порядок застосування методів і утиліт;

– формування звітів, форм за відповідними вимогам;

– контроль, який допомагає забезпечувати якість і координувати зміни;

– формування «віх», за якими керівники оцінюють прогрес.

Процес конструювання програмного забезпечення складається з послідовності кроків, що використовують методи, утиліти і процедури. Ці послідовності кроків часто називають парадигмами ТКПЗ.

Застосування парадигм ТКПЗ гарантує систематичний, впорядкований підхід до промислової розробки, використання та супроводу ПЗ. Фактично, парадигми вносять в процес створення ПЗ організуюче інженерне зерно, необхідність якого важко переоцінити.

Розглянемо найбільш популярні парадигми ТКПЗ.

2. Класичний життєвий цикл

Дуже часто класичний життєвий цикл називають *каскадною або водоспадною моделлю*, підкреслюючи, що розробка розглядається як послідовність етапів, причому перехід на наступний, ієрархічно нижній етап відбувається тільки після повного завершення робіт на поточному етапі (рис. 1) (за [1]).

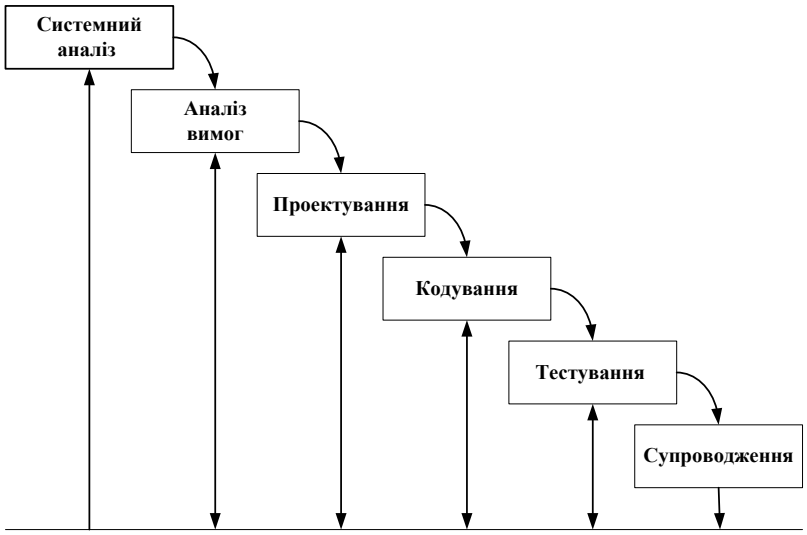

Рис. 1. Класичний життєвий цикл розробки ПЗ

Охарактеризуємо зміст основних етапів.

Мається на увазі, що розробка починається на системному рівні і проходить через аналіз, проектування, кодування, тестування і супровід. При цьому моделюються дії стандартного інженерного циклу.

Системний аналіз задає роль кожного елемента в комп'ютерній системі, взаємодію елементів один з одним. Оскільки ПЗ є лише частиною великої системи, то аналіз починається з визначення вимог до всіх системних елементів і призначення підмножини цих вимог програмному «елементу».

Необхідність системного підходу явно проявляється, коли формується інтерфейс ПЗ з іншими елементами (апаратурою, користувачами, базами даних). На цьому ж етапі починається рішення задачі планування проекту ПЗ. У ході планування проекту визначаються обсяг проектних робіт та їх ризик, необхідні трудовитрати, формуються робочі завдання і план-графік робіт.

Аналіз вимог відноситься до програмного елементу - програмного забезпечення. Уточнюються і деталізуються його функції, характеристики та інтерфейс.

Всі визначення документуються в специфікації аналізу. Тут же завершується вирішення задачі планування проекту.

Проектування полягає у створенні представлень:

– архітектури ПЗ;
– модульної структури ПЗ;
– алгоритмічної структури ПЗ;
– структури даних;
– вхідного і вихідного інтерфейсу (вхідних та вихідних форм даних).

Вихідні дані для проектування містяться в специфікації аналізу, тобто в ході проектування виконується трансляція вимог до ПЗ у множину проектних представлень. При вирішенні завдань проектування основна увага приділяється якості майбутнього програмного продукту.

Кодування полягає в переведенні результатів проектування в текст мови програмування.

Тестування – виконання програми для виявлення дефектів у функціях, логіці і формі реалізації програмного продукту.

Супровід – це внесення змін до ПЗ, яке експлуатується. Цілі змін:

– виправлення помилок;
– адаптація до змін зовнішнього для ПЗ середовища;
– вдосконалення ПЗ за вимогами замовника.

Супровід ПЗ складається в повторному застосуванні кожного з попередніх кроків (етапів) життєвого циклу до існуючої програми, але не в розробці нової програми.

Як і будь-яка інженерна схема, класичний життєвий цикл має переваги і недоліки.

Переваги класичного життєвого циклу

1) дає план і часовий графік по всіх етапах проекту;
2) упорядковує хід конструювання.

Недоліки класичного життєвого циклу:

1) реальні проекти часто вимагають відхилення від стандартної послідовності кроків;

2) цикл заснований на точному формулюванні вихідних вимог до ПЗ (реально на початку проекту вимоги замовника визначені лише частково);

3) результати проекту доступні замовнику тільки в кінці роботи.

3. Макетування

Досить часто замовник не може сформулювати детальні вимоги щодо введення, обробки або виведення даних для майбутнього програмного продукту. З іншого боку, розробник може сумніватися в підлаштуванні продукту під операційну систему, формі діалогу з користувачем або в ефективності реалізованого алгоритму. У цих випадках доцільно використовувати макетування.

Макетування – це процес створення моделі необхідного програмного продукту. Основна мета макетування – зняти невизначеності у вимогах замовника.

Модель може приймати одну з трьох форм:

1) паперовий макет або макет на основі ПК (зображує або малює людино-машинний діалог);

2) працюючий макет (виконує деяку частину необхідних функцій);

3) існуюча програма (характеристики якої потім повинні бути поліпшені).

Як показано на рис. 2, макетування базується на багаторазовому повторенні ітерацій, в яких беруть участь замовник та розробник.

Послідовність дій при макетуванні представлена на рис. 3.

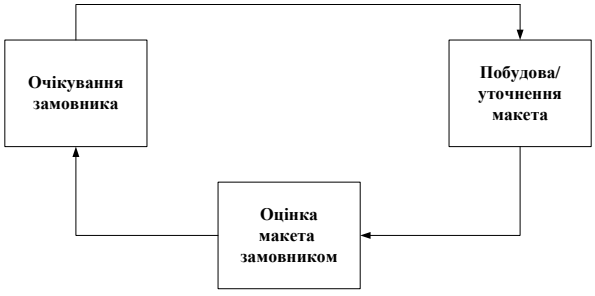

Рис. 2. Макетування

Рис. 3. Послідовність дій при макетуванні

Макетування починається зі збору й уточнення вимог до створюваного ПЗ. Розробник і замовник зустрічаються і визначають всі цілі ПЗ та встановлюють, які вимоги відомі, а які належить ще визначити.

Потім виконується швидке проектування. У ньому увага зосереджується на тих характеристиках ПЗ, які повинні бути видимі користувачеві.

Швидке проектування приводить до побудови макета. Макет оцінюється замовником і використовується для уточнення вимог до ПЗ. Ітерації повторюються до тих пір, поки макет не виявить всі вимоги замовника і, тим самим, не дасть можливість розробнику зрозуміти, що повинно бути зроблено.

Гідність макетування: забезпечує визначення повних вимог до ПЗ.

Недоліки макетування:

– замовник може прийняти макет за продукт;
– розробник може прийняти макет за продукт.

Контрольні питання

1. Дайте визначення технології конструювання програмного забезпечення.
2. Які етапи класичного життєвого циклу Ви знаєте?
3. Охарактеризуйте зміст етапів класичного життєвого циклу.
4. Поясніть переваги і недоліки класичного життєвого циклу.
5. Чим відрізняється класичний життєвий цикл від макетування?
6. Які існують форми макетування?

ЛЕКЦІЯ 2. СТРАТЕГІЇ ТА МОДЕЛІ КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План лекції

1. Стратегії конструювання ПЗ.
2. XP-процес.
3. Моделі якості процесів конструювання.

1. Стратегії конструювання програмного забезпечення

Існують 3 стратегії конструювання ПЗ:

– одноразовий прохід (водоспадна стратегія) - лінійна послідовність етапів конструювання;
– інкрементна стратегія. На початку процесу визначаються всі користувацькі та системні вимоги, частина, що залишилася конструювання

виконується у вигляді послідовності версій. Перша версія реалізує частину запланованих можливостей, наступна версія реалізує додаткові можливості і т. д., поки не буде отримана повна система;

– еволюційна стратегія. Система також будується у вигляді послідовності версій, але на початку процесу визначені не всі вимоги. Вимоги уточнюються в результаті розробки версій.

1.1. Інкрементна модель

Інкрементна модель є класичним прикладом інкрементної стратегії конструювання (рис. 4). Вона об'єднує елементи послідовної водоспадної моделі з ітераційною філософією макетування.

Кожна лінійна послідовність тут виконує інкремент ПЗ, що поставляється. Наприклад, ПЗ для обробки слів в 1-му інкременті реалізує функції базової обробки файлів, функції редагування та документування; у 2-му інкременті - більш складні можливості редагування та документування; в 3-му інкременті - перевірку орфографії та граматики; в 4-му інкременті - можливості компонування сторінки.

Перший інкремент приводить до отримання базового продукту, що реалізує базові вимоги (правда, багато допоміжних вимоги залишаються нереалізованими).

План наступного інкремента передбачає модифікацію базового продукту, що забезпечує додаткові характеристики та функціональність.

За своєю природою інкрементний процес ітераційний, але, на відміну від макетування, інкрементна модель забезпечує на кожному інкременті працюючий продукт.

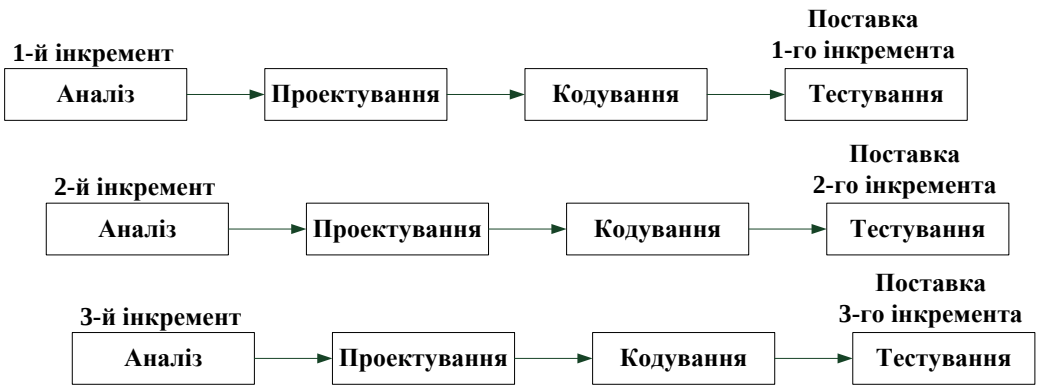

Рис. 4. Інкрементна модель

Сучасна реалізація інкрементного підходу - екстремальне програмування XP. Воно орієнтоване на дуже малі прирощення функціональності.

1.2. Швидка розробка додатків

Модель швидкої розробки додатків (Rapid Application Development) - другий приклад застосування інкрементной стратегії конструювання (рис. 5). RAD-модель забезпечує екстремально короткий цикл розробки. RAD - високошвидкісна адаптація лінійної послідовної моделі, в якій швидка

розробка досягається за рахунок використання компонентно-орієнтованого конструювання. Якщо вимоги повністю визначені, а проектна область обмежена, RAD-процес дозволяє групі створити повністю функціональну систему за дуже короткий час (60-90 днів).

RAD-підхід орієнтований на розробку інформаційних систем і виділяє наступні етапи:

– бізнес-моделювання. Моделюється інформаційний потік між бізнес-функціями. Шукається відповідь на наступні запитання: Яка інформація керує бізнес-процесом? Яка генерується інформація? Хто генерує її? Де інформація застосовується? Хто обробляє її?

– моделювання даних. Інформаційний потік, визначений на етапі бізнес-моделювання, відображається в набір об'єктів даних, які потрібні для підтримки бізнесу. Ідентифікуються характеристики (властивості, атрибути) кожного об'єкта, визначаються відношення між об'єктами;

– моделювання обробки. Визначаються перетворення об'єктів даних, що забезпечують реалізацію бізнес-функцій. Створюються описи обробки для додавання, модифікації, видалення або знаходження (виправлення) об'єктів даних;

– генерація додатка. Передбачається використання методів, орієнтованих на мови програмування 4-го покоління. Замість створення ПЗ за допомогою мов програмування 3-го покоління, RAD-процес працює з програмними компонентами, які повторно використовуються, або створює такі компоненти. Для забезпечення конструювання використовуються утиліти автоматизації;

– тестування та об'єднання. Оскільки застосовуються компоненти повторного використання, багато програмних елементів вже протестовані. Це зменшує час тестування (хоча все нові елементи повинні бути протестовані).

Застосування RAD можливо в тому випадку, коли кожна головна функція може бути завершена за 3 місяці. Кожна головна функція адресується окремій групі розробників, а потім інтегрується в цілу систему.

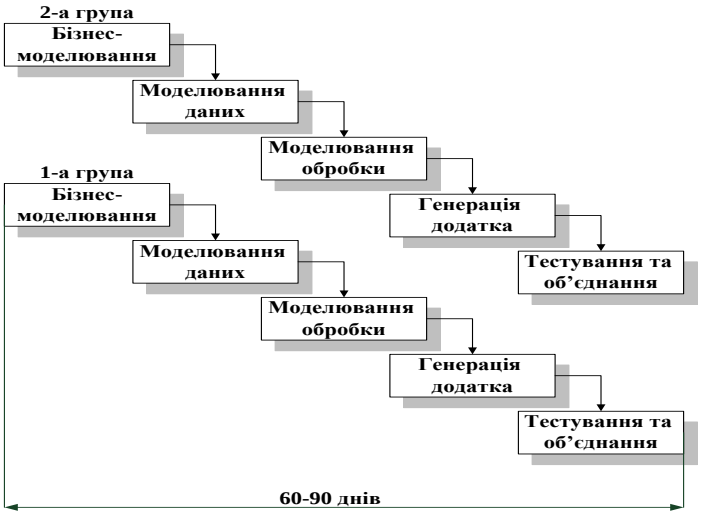

Рис. 5. Модель швидкої розробки додатків

Застосування RAD має і свої *недоліки та обмеження*.

1. Для великих проектів в RAD потрібні істотні людські ресурси (необхідно створити достатню кількість груп).

2. RAD застосовна тільки для таких додатків, до яких можна застосувати декомпозицію на окремі модулі і в яких продуктивність не є критичною величиною.

3. RAD не застосовна в умовах високих технічних ризиків (тобто при використанні нової технології).

1.3. Спіральна модель

Спіральна модель – класичний приклад застосування еволюційної стратегії конструювання. Спіральна модель базується на кращих властивостях класичного життєвого циклу та макетування, до яких додається новий елемент – аналіз ризику, відсутній в цих парадигмах.

Етапи спіральної моделі:

1 - початковий збір вимог і планування проекту;

2 - та ж робота, але на основі рекомендацій замовника;

3 - аналіз ризику на основі початкових вимог;

4 - аналіз ризику на основі реакції замовника;

5 - перехід до комплексної системи;

6 - початковий макет системи;

7 - наступний рівень макета;

8 - конструювання системи;

9 - оцінювання замовником.

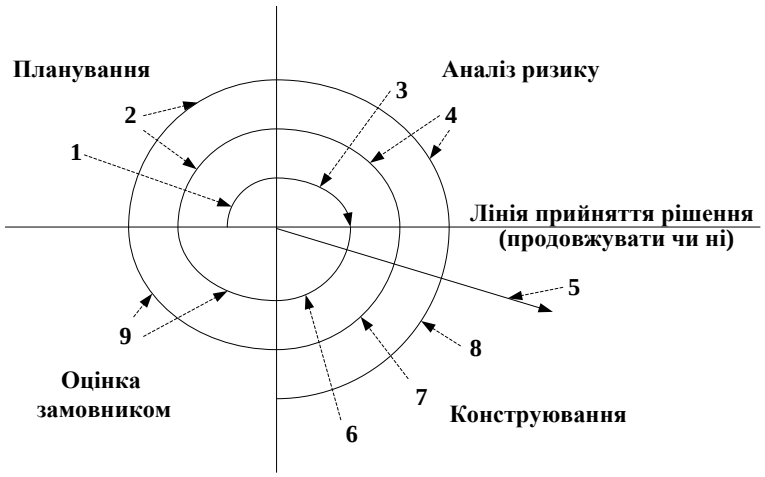

Рис. 6. Спіральна модель

Як показано на рис. 6, модель визначає чотири дії, які подаються чотирма квадратами спіралі.

1. Планування – визначення цілей, варіантів і обмежень.

2. Аналіз ризику – аналіз варіантів і розпізнавання / вибір ризику.

3. Конструювання - розробка продукту наступного рівня.

4. Оцінювання – оцінка замовником поточних результатів конструювання.

Інтегруючий аспект спіральної моделі очевидний при обліку радіального вимірювання спіралі. З кожною ітерацією по спіралі (просуванням від центру до периферії) будуються все більш повні версії ПЗ.

У першому витку спіралі визначаються початкові цілі, варіанти та обмеження, розпізнається і аналізується ризик. Якщо аналіз ризику показує невизначеність вимог, на допомогу розробнику і замовнику приходить макетування (що використовується в квадранті конструювання). Для подальшого визначення проблемних і уточнених вимог може бути використано моделювання. Замовник оцінює інженерну (конструкторську) роботу і вносить пропозиції щодо модифікації (квадрант оцінки замовником). Наступна фаза планування та аналізу ризику базується на пропозиціях замовника. У кожному циклі по спіралі результати аналізу ризику формуються у вигляді «продовжувати, не продовжувати». Якщо ризик дуже великий, проект може бути зупинений.

У більшості випадків рух по спіралі триває, з кожним кроком просуваючи розробників до більш загальної моделі системи. У кожному циклі по спіралі потрібне конструювання (нижній правий квадрант), яке може бути реалізоване класичним життєвим циклом або макетуванням. Зауважимо, що кількість дій з розробки (відбуваються в правому нижньому квадранті) зростає у міру просування від центру спіралі.

Переваги спіральної моделі:

1) найбільш реально (у вигляді еволюції) відображає розробку програмного забезпечення;

2) дозволяє явно враховувати ризик на кожному витку еволюції розробки;

3) включає крок системного підходу в ітераційну структуру розробки;

4) використовує моделювання для зменшення ризику і вдосконалення програмного виробу.

Недоліки спіральної моделі:

1) новизна (відсутня достатня статистика ефективності моделі);

2) підвищені вимоги до замовника;

3) труднощі контролю та управління часом розробки.

1.4. Компонентно-орієнтована модель

Компонентно-орієнтована модель є розвитком спіральної моделі і теж базується на еволюційній стратегії конструювання. У цій моделі конкретизується зміст квадранта конструювання - він відображає той факт, що в сучасних умовах нова розробка повинна базуватися на повторному використанні існуючих програмних компонентів (рис. 7).

Програмні компоненти, створені в реалізованих програмних проектах, зберігаються в бібліотеці. У новому програмному проекті, виходячи з вимог замовника, виявляються кандидати в компоненти. Далі перевіряється наявність цих кандидатів у бібліотеці. Якщо вони знайдені, то компоненти витягуються з бібліотеки і використовуються повторно. В іншому випадку

створюються нові компоненти, вони застосовуються в проекті і включаються в бібліотеку.

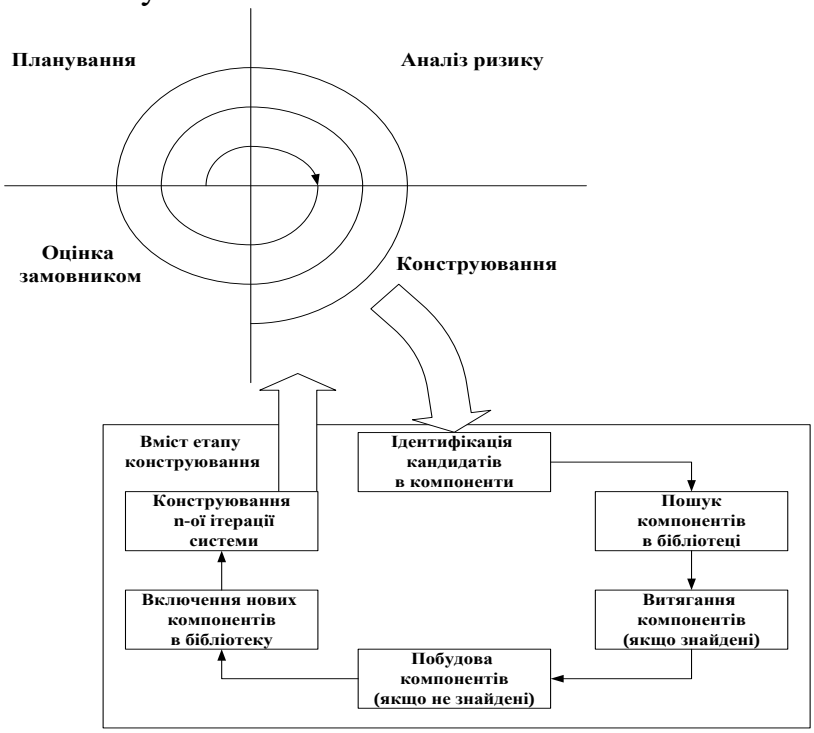

Рис. 7. Компонентно-орієнтована модель

Переваги компонентно-орієнтованої моделі:

1) зменшує на 30% час розробки програмного продукту;
2) зменшує вартість програмної розробки до 70%;
3) збільшує в півтора рази продуктивність розробки.

2. XP-процес

Екстремальне програмування (eXtreme Programming, XP) - полегшений (рухливий) процес (або методологія), головний автор якого - Кент Бек (1999). XP-процес орієнтований на групи малого та середнього розміру, що будують програмне забезпечення в умовах невизначених або швидко змінних вимог. XP-групу утворюють до 10 співробітників, які розміщуються в одному приміщенні.

Основна ідея XP - усунути високу вартість зміни, яка характерна для додатків з використанням об'єктів, патернів і реляційних баз даних. Тому XP-процес повинен бути високодинамічним процесом. XP-група має справу зі змінами вимог протягом всього ітераційного циклу розробки, причому цикл складається з дуже коротких ітерацій.

Чотирма базовими діями в XP-циклі є:

- кодування;
- тестування;
- втілення вимог замовника;
- проектування.

Динамізм забезпечується за допомогою чотирьох характеристик: безперервного зв'язку із замовником (і в межах групи), простоти (завжди вибирається мінімальне рішення), швидкого зворотного зв'язку (за допомогою модульного та функціонального тестування), сміливості у проведенні профілактики можливих проблем.

Принципи, що підтримуються в XP: мінімальність, простота, еволюційний цикл розробки, мала тривалість ітерації, участь користувача, оптимальні стандарти кодування і досягають «екстремальних значень» і є строго впорядкованим процесом.

Базис XP утворюють перераховані нижче дванадцять методів.

1. Гра планування (Planning game) - швидке визначення області дії наступної реалізації шляхом об'єднання ділових пріоритетів і технічних оцінок. Замовник формує область дії, пріоритетність і терміни з точки зору бізнесу, а розробники оцінюють і простежують просування (прогрес).

2. Часта зміна версій (Small releases) - швидкий запуск у виробництво простої системи. Нові версії реалізуються в дуже короткому (двотижневому) циклі.

3. Метафора (Metaphor) - вся розробка проводиться на основі простої, загальнодоступною історії про те, як працює вся система.

4. Просте проектування (Simple design) - проектування виконується настільки просто, наскільки це можливо в даний момент.

5. Тестування (Testing) - безперервне написання тестів для модулів, які повинні виконуватися бездоганно; замовники пишуть тести для демонстрації закінченості функцій. «Тестують, а потім кодуються» означає, що вхідним критерієм для написання коду є тестовий варіант, що «відмовив».

6. Реорганізація (Refactoring) - система реструктурується, але її поведінка не змінюється; мета - усунути дублювання, поліпшити взаємодію, спростити систему або додати в неї гнучкість.

7. Парне програмування (Pair programming) - весь код пишеться двома програмістами, що працюють на одному комп'ютері.

8. Колективне володіння кодом (Collective ownership) - будь-який розробник може покращувати будь-який код системи в будь-який час.

9. Безперервна інтеграція (Continuous integration) - система інтегрується і будується багато разів на день, по мірі завершення кожного завдання. Безперервне регресійне тестування, тобто повторення попередніх тестів, гарантує, що зміни вимог не приведуть до регресу функціональності.

10. 40-годинний тиждень (40-hour week) - як правило, працюють не більше 40 годин на тиждень. Не можна подвоювати робочий тиждень за рахунок понаднормових робіт.

11. Локальний замовник (On-site customer) - в групі весь час повинен знаходитися представник замовника, дійсно готовий відповідати на питання розробників.

12. Стандарти кодування (Coding standards) - повинні витримуватися правила, що забезпечують однакове представлення програмного коду у всіх частинах програмної системи.

Гра планування і часта зміна версій залежать від замовника, що забезпечує набір «історій» (коротких описів), що характеризують роботу, яка буде виконуватися для кожної версії системи. Версії генеруються кожні два тижні, тому розробники і замовник повинні прийти до згоди про те, які історії будуть здійснені в межах двох тижнів.

«Метафора» забезпечує глобальне «бачення» проекту.

Парне програмування - один з найбільш спірних методів в XP, воно впливає на ресурси, що важливо для менеджерів, які вирішують, чи буде проект використати XP. Парне програмування призводить до підвищення якості та зменшення часу циклу. Для узгодженої групи витрати збільшуються на 15%, а час циклу скорочується на 40-50%. Для Інтернет-середовища збільшення швидкості продажів покриває підвищення витрат.

Колективне володіння означає, що будь-який розробник може змінювати будь-який фрагмент коду системи в будь-який час. Безперервна інтеграція, безперервне регресійне тестування і парне програмування XP забезпечують захист від виникаючих при цьому проблем.

«Тестують, а потім кодуються» - ця фраза виражає акцент XP на тестуванні. Роздум про тестування на початку циклу життя - добре відома практика конструювання ПЗ.

Основним засобом управління XP є метрика, а середа метрик - «велика візуальна діаграма». Зазвичай використовують 3-4 метрики, причому такі, які видимі всій групі. Рекомендованою в XP метрикою є «швидкість проекту» - кількість історій заданого розміру, які можуть бути реалізовані в ітерації.

При прийнятті XP рекомендується освоювати його методи по одному, кожний раз вибираючи метод, орієнтований на саму важку проблему групи.

Розглянемо структуру «ідеального» XP-процесу. Основним структурним елементом процесу є XP-реалізація, в яку багато разів вкладається базовий елемент - XP-ітерація. До складу XP-реалізації та XP-ітерації входять три фази - дослідження, блокування, регулювання.

Дослідження (exploration) - це пошук нових вимог (історій, задач), які повинна виконувати система. Блокування (commitment) - вибір для реалізації конкретної підмножини з усіх можливих вимог (іншими словами, планування). Регулювання (steering) - проведення розробки, втілення плану в життя.

XP рекомендує: перша реалізація повинна мати тривалість 2-6 місяців, тривалість решти реалізацій - близько двох місяців, кожна ітерація триває приблизно два тижні, а чисельність групи розробників не перевищує 10 осіб.

Процес ініціюється початковою дослідницькою фазою. Фаза дослідження, з якої починається будь-яка реалізація та ітерація, має клапан

«пропуску», на цій фазі приймається рішення про доцільність подальшого продовження роботи.

Передбачається, що тривалість першої реалізації становить 3 місяці, тривалість другої – сьомої реалізацій - 2 місяці. Друга - сьома реалізації утворюють період супроводу, що характеризує природу XP-проекту. Кожна ітерація триває два тижні, за винятком тих, які відносять до пізньої стадії реалізації - «запуску у виробництво» (в цей час темп ітерації прискорюється).

Найбільш важка перша реалізація - пройти за три місяці від звичайного старту (скажімо, окремий співробітник не зафіксував ніяких вимог, не визначені обмеження) до поставки замовнику системи промислового якості дуже складно.

3. Моделі якості процесів конструювання

У сучасних умовах дуже важливо гарантувати високу якість вашого процесу конструювання ПЗ. Таку гарантію дає сертифікат якості процесу, що підтверджує його відповідність прийнятим міжнародним стандартам. Модель стандарту ISO 9001:2000 орієнтована на процеси розробки з будь-яких галузей людської діяльності. Стандарт ISO / IEC 15504 спеціалізується на процесах програмної розробки і відрізняється більш високим рівнем деталізації. Досить сказати, що обсяг цього стандарту перевищує 500 сторінок. Значна частина ідей ISO / IEC 15504 взята з моделі CMM.

Базовим поняттям моделі CMM вважається зрілість компанії. Незрілої називають компанію, де процес конструювання ПЗ та прийняті рішення залежать тільки від таланту конкретних розробників. Як наслідок, тут висока ймовірність перевищення бюджету або зриву термінів закінчення проекту.

Таким чином, модель CMM фіксує критерії для оцінки зрілості компанії і пропонує рецепти для поліпшення існуючих в ній процесів. Іншими словами, в ній не тільки сформульовані умови, необхідні для досягнення мінімальної організованості процесу, але і даються рекомендації щодо подальшого вдосконалення процесів.

Дуже важливо відзначити, що модель CMM орієнтована на побудову системи постійного поліпшення процесів. У ній зафіксовані п'ять рівнів зрілості (рис. 8) і передбачений плавний, поетапний підхід до вдосконалення процесів - можна поетапно отримувати підтвердження про поліпшення процесів після кожного рівня зрілості.

Початковий рівень (рівень 1) означає, що процес в компанії не формалізований. Він не може строго плануватися і відслідковуватися, його успіх носить випадковий характер. Результат роботи цілком і повністю залежить від особистих якостей окремих співробітників. При звільненні таких працівників проект зупиняється.

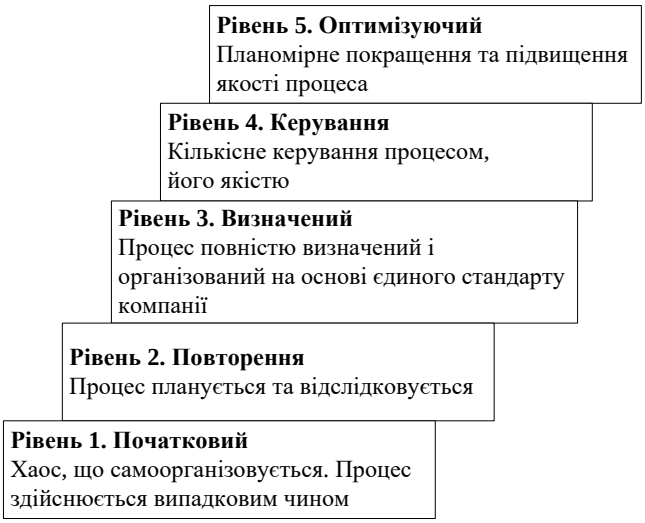

Рис. 8 П'ять рівнів зрілості моделі CMM

Для переходу на рівень повторів (рівень 2) необхідно впровадити формальні процедури для виконання основних елементів процесу конструювання. Результати виконання процесу відповідають заданим вимогам і стандартам. Основна відмінність від рівня 1 полягає в тому, що виконання процесу планується і контролюється. Засоби планування і управління, що застосовуються, дають можливість повторення раніше досягнутих успіхів.

Наступний, визначений рівень (рівень 3) вимагає, щоб всі елементи процесу були визначені, стандартизовані і задокументовані. Основна відмінність від рівня 2 полягає в тому, що елементи процесу рівня 3 плануються і управляються на основі єдиного стандарту компанії. Якість ПЗ, яке розробляється, уже не залежить від здібностей окремих особистостей.

З переходом на рівень керування (рівень 4) в компанії приймаються кількісні показники якості як програмних продуктів, так і процесу. Це забезпечує більш точне планування проекту та контроль якості його результатів. Основна відмінність від рівня 3 складається в більш об'єктивної, кількісної оцінки продукту і процесу.

Вищий, оптимізуючий рівень (рівень 5) має на увазі, що головним завданням компанії стає постійне поліпшення і підвищення ефективності існуючих процесів, введення нових технологій. Основна відмінність від рівня 4 полягає в тому, що технологія створення та супроводу програмних продуктів планомірно і послідовно удосконалюється.

Кожен рівень CMM характеризується областю ключових процесів (ОКП), причому вважається, що кожний наступний рівень включає в себе всі характеристики попередніх рівнів. Інакше кажучи, для 3-го рівня зрілості розглядаються ОКП 3-го рівня, ОКП 2-го рівня і ОКП 1-го рівня. Область ключових процесів утворюють процеси, які при спільному виконанні призводять до досягнення певного набору цілей. Наприклад, ОКП 5-го рівня утворюють процеси:

– запобігання дефектів;
– управління змінами технології;
– управління змінами процесу.

Якщо всі цілі ОКП досягнуті, компанії присвоюється сертифікат даного рівня зрілості. Якщо хоча б одна мета не досягнута, то компанія не може відповідати даному рівню СММ.

Контрольні питання

1. Чим відрізняються один від одного стратегії конструювання ПЗ?
2. Вкажіть подібності та відмінності класичного життєвого циклу та інкрементної моделі.
3. Поясніть переваги і недоліки інкрементної моделі.
4. Чим відрізняється модель швидкої розробки додатків від інкрементної моделі?
5. Поясніть переваги і недоліки моделі швидкої розробки додатків.
6. Вкажіть подібності та відмінності спіральної моделі і класичного життєвого циклу.
7. У чому полягає головна особливість спіральної моделі?
8. Чим відрізняється компонентно-орієнтована модель від спіральної моделі і класичного життєвого циклу?
9. Перерахуйте переваги і недоліки компонентно-орієнтованої моделі.
10. Перерахуйте характеристики XP-процесу.
17. Перерахуйте методи XP-процесу.
11. У чому полягає головна особливість XP-процесу?
12. Охарактеризуйте зміст гри планування в XP-процесі.
13. Яка особливість проектування в XP-процесі?
14. Яка особливість програмування в XP-процесі?
15. Яка особливість тестування в XP-процесі?
16. Чим відрізняється XP-реалізація від XP-ітерації?
17. Чим XP-реалізація схожа на XP-ітерацію?
18. Яка максимальна чисельність групи XP-розробників?
19. Які моделі якості процесів конструювання Ви знаєте?
20. Охарактеризуйте модель СММ.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

ЛЕКЦІЯ 3. ОСНОВИ ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ.

План лекції

1. Створення та етапи проектування програмних систем.
2. Особливості процесу синтезу програмних систем.
3. Особливості етапу проектування.

1. Створення та етапи проектування ПС

1. Початок проекту.

Перед плануванням проекту слід:

- встановити цілі і проблемну область проекту;
- обговорити альтернативні рішення;
- виявити технічні та управлінські обмеження.

2. Вимірювання, заходи та метрики.

Вимірювання процесу виконуються з метою його покращення, вимірювання продукту - для підвищення його якості. В результаті вимірювання визначається міра - кількісна характеристика якої-небудь властивості об'єкта. Шляхом безпосередніх вимірювань можуть визначатися тільки опорні властивості об'єкта. Всі інші властивості оцінюються в результаті обчислення тих чи інших функцій від значень опорних характеристик. Обчислення цих функцій проводяться за формулами, що дає числові значення і званим метриками.

3. Процес оцінки.

При плануванні програмного проекту треба оцінити людські ресурси (в людино-місяцях), тривалість (у календарних датах), вартість (у гривнях).

4. Аналіз ризику.

На цій стадії досліджується область невизначеності, наявна перед створенням програмного продукту. Аналізується її вплив на проект. Чи немає прихованих від уваги важких технічних проблем? Чи не стануть зміни, що проявилися в ході проектування, причиною неприпустимого відставання за термінами? У результаті приймається рішення - виконувати проект чи ні.

5. Планування.

Визначається набір проектних задач. Встановлюються зв'язки між задачами, оцінюється складність кожної задачі. Визначаються людські та інші ресурси. Створюється мережевий графік задач, проводиться його тимчасова розмітка.

6. Трасування й контроль.

Кожна задача, позначена в плані, відстежується керівником проекту. При відставанні у вирішенні задачі застосовуються утиліти повторного планування. За допомогою утиліт визначається вплив цього відставання на проміжну віху і загальний час конструювання. Під віхою розуміється тимчасова мітка, до якої прив'язане підведення проміжних підсумків.

В результаті повторного планування:

– можуть бути перерозподілені ресурси;
– можуть бути реорганізовані задачі;
– можуть бути переглянуті вихідні зобов'язання.

7. Планування проектних задач.

Основним завданням при плануванні є визначення WBS - Work Breakdown Structure (структури розподілу робіт). Вона складається за допомогою утиліти планування проекту. Типова WBS наведена на рис. 9.

Системний аналіз проводиться з метою:

1) з'ясування потреб замовника;
2) оцінки здійсненності системи;
3) виконання економічного та технічного аналізу;
4) розподілу функцій за елементами комп'ютерної системи (апаратурі, програмам, людям, базам даних і т. д.);
5) визначення вартості та обмежень планування;
6) створення системної специфікації.

У системній специфікації описуються функції, характеристики системи, обмеження розробки, вхідна та вихідна інформація.

Аналіз вимог дає можливість:

1) визначити функції та характеристики програмного продукту;
2) позначити інтерфейс продукту з іншими системними елементами;
3) визначити проектні обмеження програмного продукту;
4) побудувати моделі: процесу, даних, режимів функціонування продукту;
5) створити такі форми подання інформації та функцій системи, які можна використовувати в ході проектування.

Результати аналізу зводяться в специфікацію вимог до програмного продукту.

Як видно з типової структури, задачі щодо проектування і планування тестів можуть бути розпаралелені. Завдяки модульній природі ПЗ для кожного модуля можна передбачити паралельний шлях для детального (процедурного) проектування, кодування і тестування.

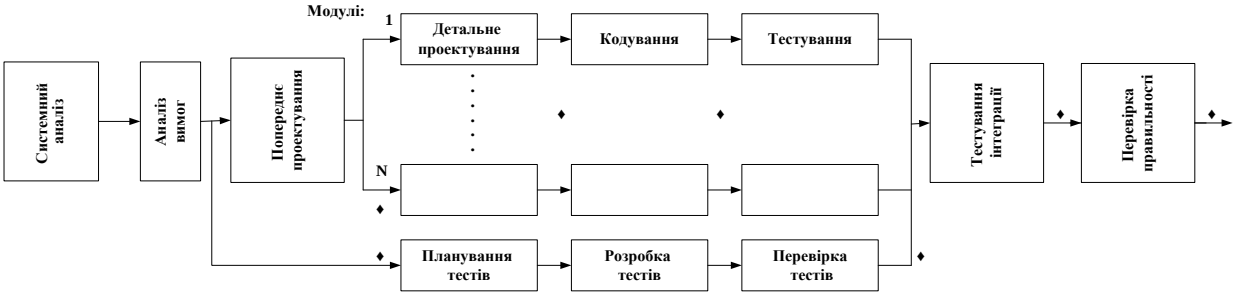

Рис. 9 Типова структура розподілення проектних робіт

Після отримання всіх модулів ПЗ вирішується задача тестування інтеграції - об'єднання елементів у єдине ціле. Далі проводиться тестування правильності, яке забезпечує перевірку відповідності ПЗ вимогам замовника.

Рекомендоване правило розподілу витрат проекту - 40-20-40:

– на аналіз і проектування припадає 40% витрат (з них на планування та системний аналіз - 5%);
– на кодування - 20%;
– на тестування і налагодження - 40%.

2. Особливості процесу синтезу програмних систем

Технологічний цикл конструювання програмної системи (ПС) включає три процеси - *аналіз, синтез і супровід.*

В ході *аналізу* шукається відповідь на питання: «Що повинна робити майбутня система?».

В процесі *синтезу* формується відповідь на питання: «Яким чином система буде реалізовувати пред'явлені до неї вимоги?».

Виділяють три етапи синтезу: проектування ПС, кодування ПС, тестування ПС (рис. 10).

Розглянемо інформаційні потоки процесу синтезу.

Етап проектування обгрунтовує вимоги до ПС, представлені інформаційної, функціональної і поведінкової моделями аналізу. Іншими словами, моделі аналізу поставляють етапу проектування вихідні відомості для роботи. Інформаційна модель описує інформацію, яку, на думку замовника, повинна обробляти ПС. Функціональна модель визначає перелік функцій обробки. Поведінкова модель фіксує бажану динаміку системи (режими її роботи). На виході етапу проектування - розробка даних, розробка архітектури та процедурна розробка ПС.

Розробка даних - це результат перетворення інформаційної моделі аналізу в структури даних, які будуть потрібні для реалізації програмної системи.

Розробка архітектури виділяє основні структурні компоненти та фіксує зв'язки між ними.

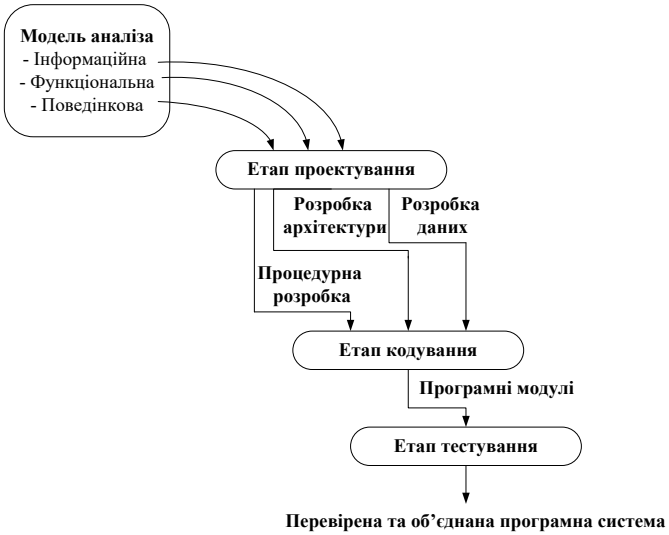

Рис. 10. Інформаційні потоки процесу синтезу ПС

Процедурна розробка описує послідовність дій у структурних компонентах, тобто визначає їх зміст.

Далі створюються тексти програмних модулів, проводиться тестування для об'єднання і перевірки ПС. На проектування, кодування і тестування припадає понад 75% вартості конструювання ПС. Прийняті тут рішення мають вирішальний вплив на успіх реалізації ПС та легкість, з якою ПС буде супроводжуватися.

Слід зазначити, що рішення, прийняті в ході проектування, роблять його допоміжним етапом до процесу синтезу. Важливість проектування можна визначити одним словом - якість. Проектування - етап, на якому «вирощується» якість розробки ПС. Проектування - єдиний шлях, що забезпечує правильну трансляцію вимог замовника в кінцевий програмний продукт.

3. Особливості етапу проектування

Проектування - ітераційний процес, за допомогою якого вимоги до ПС транслюються в інженерні представлення ПС. У проектуванні виділяють ступені:

- попереднє проектування - формує абстракції архітектурного рівня;
- детальне проектування - уточнює ці абстракції, додає подробиці алгоритмічного рівня;

- інтерфейсне проектування, мета якого – сформувати графічний інтерфейс користувача (GUI).

Розглянемо схему інформаційних зв'язків процесу проектування.

Попереднє проектування забезпечує:

- ідентифікацію підсистем;
- визначення основних принципів управління підсистемами, взаємодії підсистем.

Рис. 11. Інформаційні зв'язки процесу проектування

Попереднє проектування включає три типи діяльності:

1. *Структурування системи.* Система структурується на декілька підсистем, де під підсистемою розуміється незалежний програмний компонент. Визначаються взаємодії підсистем.
2. *Моделювання управління.* Визначається модель зв'язків управління між частинами системи.
3. *Декомпозиція підсистем на модулі.* Кожна підсистема розбивається на модулі. Визначаються типи модулів і міжмодульних з'єднання.

3.1. Структурування системи

Відомі чотири моделі системного структурування:

- модель сховища даних;
- модель клієнт-сервер;
- трирівнева модель;
- модель абстрактної машини.

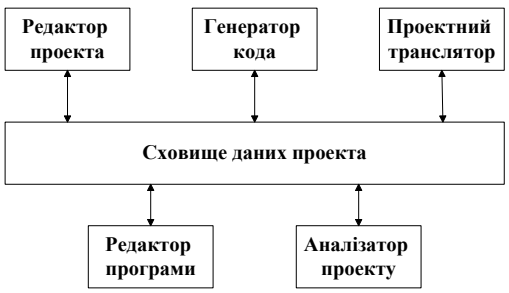

Рис. 12. Модель сховища даних

Рис. 13. Модель клієнт- сервер

У *моделі сховища даних* (рис. 12) підсистеми розділяють дані, що знаходяться в загальній пам'яті. Як правило, дані утворюють БД. Передбачається система управління цією базою.

Модель клієнт-сервер використовується для розподілених систем, де дані розподілені по серверах (рис. 13). Для передачі даних застосовують мережевий протокол, наприклад TCP / IP.

Трирівнева модель є розвитком моделі клієнт-сервер (рис. 14).

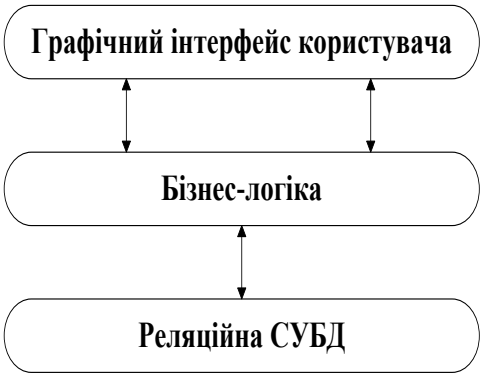

Рис. 14. Трирівнева модель

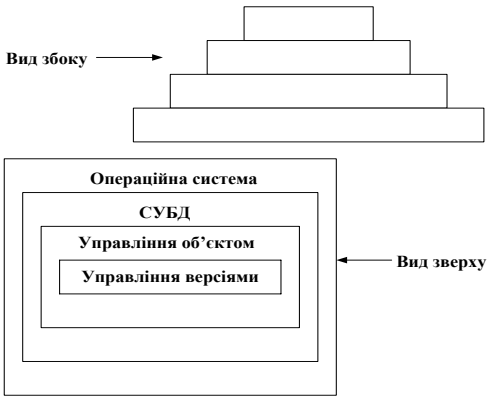

Рис. 15. Модель абстрактної машини

Рівень графічного інтерфейсу користувача запускається на машині клієнта. Бізнес-логіку утворюють модулі, що здійснюють функціональні обов'язки системи. Цей рівень запускається на сервері додатка. Реляційна СУБД зберігає дані, необхідні рівню бізнес-логіки. Цей рівень запускається на другому сервері - сервері бази даних.

Переваги трирівневої моделі:

- спрощується така модифікація рівня, яка не впливає на інші рівні;
- відділення прикладних функцій від функцій управління БД спрощує оптимізацію всієї системи.

Модель абстрактної машини відображає багатошарову систему (рис. 15) . Кожен поточний шар реалізується з використанням засобів, забезпечуваних шаром-фундаментом.

3.2. Моделювання управління

Відомі два типи моделей управління:

- модель централізованого управління;
- модель управління дією.

В моделі централізованого управління одна підсистема виділяється як системний контролер. Її обов'язки - керувати роботою інших підсистем. Розрізняють два різновиди моделей централізованого керування: *модель виклик-повернення* (рис. 16) і *модель менеджера* (рис. 17), яка використовується в системах паралельної обробки.

У моделі керування дією системи керують зовнішні події. Використовуються два різновиди моделі подійного керування: широкомовна модель і модель, керована перериваннями.

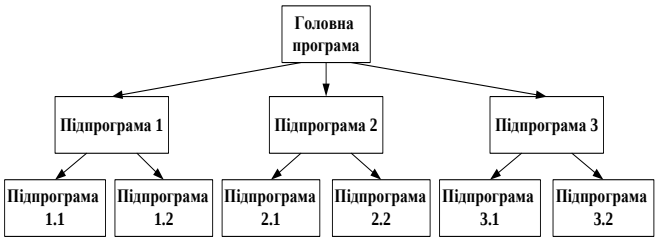

Рис. 16. Модель виклик-повернення

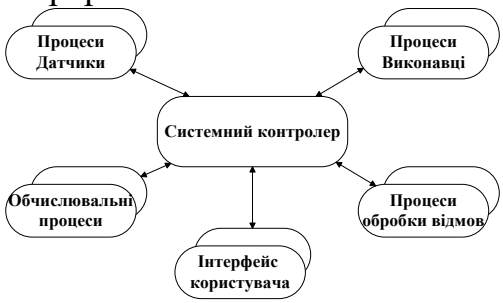

Рис. 17. Модель менеджера

В *широкомовної моделі* (рис. 18) кожна підсистема повідомляє оброблювача про свій інтерес до конкретних подій. Коли подія відбувається, оброблювач пересилає його підсистемі, яка може обробити цю подію. Функції управління в обробник не вбудовуються.

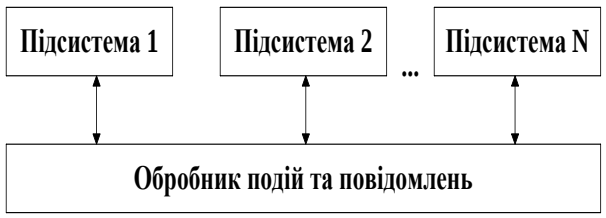

Рис. 18. Широкомовна модель

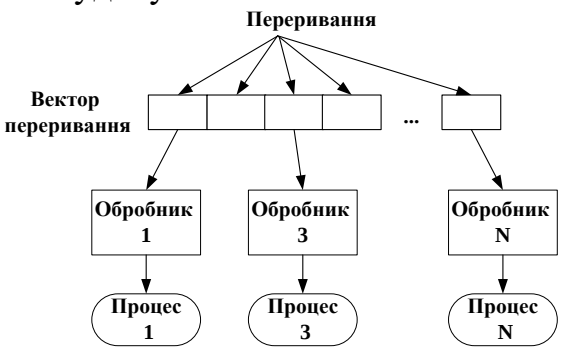

Рис. 19. Модель, керована перериваннями

У моделі, керованої перериваннями (рис. 19), всі переривання розбиті на групи - типи, які утворюють вектор переривань. Для кожного типу переривання є свій оброблювач. Кожен оброблювач реагує на свій тип переривання і запускає свій процес.

Контрольні питання

1. Охарактеризуйте етапи проектування програмних систем.
2. Яка мета синтезу програмної системи? Перерахуйте етапи синтезу.
3. Дайте визначення розробки даних, розробки архітектури та процедурної розробки.
4. Які особливості має етап проектування?
5. Рішення яких завдань забезпечує попереднє проектування?
6. Які моделі системного структурування ви знаєте?
7. Чим відрізняється модель клієнт-сервер від трирівневої моделі?
8. Які типи моделей управління ви знаєте?
9. Які існують різновиди моделей централізованого керування?
10. Поясніть різновиди моделей управління дією.

РОЗДІЛ 3
ТЕСТУВАННЯ ПРОГРАМНИХ СИСТЕМ

ЛЕКЦІЯ 4. СПОСОБИ ТЕСТУВАННЯ ПРОГРАМНИХ СИСТЕМ.

План лекції

1. Основні поняття та принципи тестування.
2. Структурне тестування. Поняття «білого ящика».

1. Основні поняття та принципи тестування

Тестування - процес виконання програми з метою виявлення помилок. Кроки процесу задаються тестами.

Кожен тест визначає:

- свій набір вихідних даних і умов для запуску програми;
- набір очікуваних результатів роботи програми.

Інша назва тесту - тестовий варіант. Повну перевірку програми гарантує *вичерпне тестування.* Воно вимагає перевірити всі набори вихідних даних, всі варіанти їх обробки і включає велику кількість тестових варіантів. Хорошим вважають тестовий варіант з високою ймовірністю виявлення ще не розкритої помилки. Успішним називають тест, який виявляє досі не розкриту помилку.

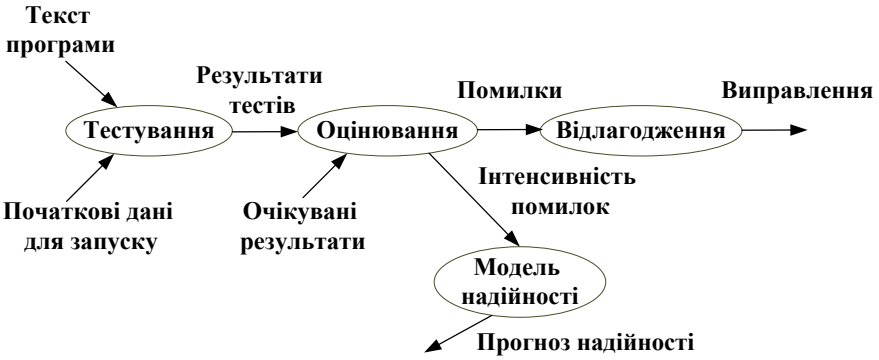

Рис. 20. Інформаційні потоки процесу тестування

Метою проектування тестових варіантів є систематичне виявлення різних класів помилок при мінімальних витратах часу і вартості.

Тестування забезпечує:

- виявлення помилок;
- демонстрацію відповідності функцій програми та її призначенню;
- демонстрацію реалізації вимог до характеристик програми;
- відображення надійності як індикатора якості програми.

Тестування не може показати відсутність дефектів (воно може показувати тільки присутність дефектів).

На вході процесу тестується три потоки:
- текст програми;
- вихідні дані для запуску програми;
- очікувані результати.

Виконуються тести, всі отримані результати оцінюються. Це означає, що реальні результати тестів порівнюються з очікуваними результатами. Коли виявляється розбіжність, фіксується помилка - починається налагодження. Процес налагодження непередбачуваний за часом. На пошук місця дефекту і виправлення може знадобитися година, день, місяць. Невизначеність у налагодженні призводить до великих труднощів у плануванні дій.

Після збору і оцінювання результатів тестування починається відображення якості та надійності ПЗ.

Існують 2 принципи тестування програми:
1) структурне тестування (тестування «білого ящика»);
2) функціональне тестування (тестування «чорного ящика»).

2. Структурне тестування

Відома внутрішня структура програми. Тобто, об'єктом тестування тут є не зовнішня, а тільки внутрішня поведінка програми. Перевіряється коректність побудови всіх елементів програми і правильність їх взаємодії один з одним. Зазвичай аналізуються керуючі зв'язку елементів, рідше - інформаційні зв'язки. Тестування за принципом «білого ящика» характеризується ступенем, в якій тести виконують або покривають логіку (вихідний текст) програми. Вичерпне тестування також ускладнене. Особливості цього принципу тестування розглянемо окремо.

Досліджуються: внутрішні елементи програми та зв'язки між ними (рис. 21).

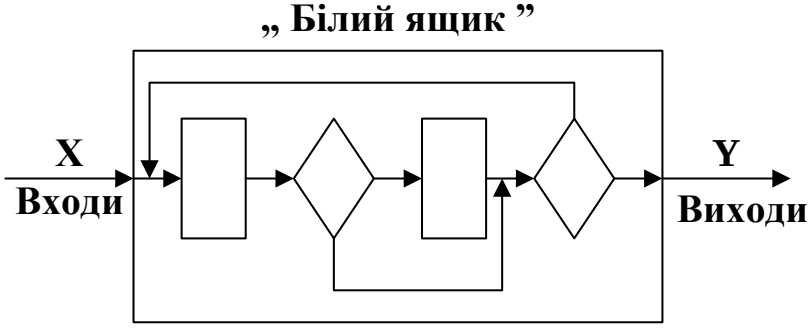

Рис. 21. Тестування «білого ящика»

2.1. Особливості тестування «білого ящика»

Зазвичай тестування «білого ящика» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреної, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління.

У цьому випадку формуються тестові варіанти, в яких:

- гарантується перевірка всіх незалежних маршрутів програми;
- проходяться гілки True, False для всіх логічних рішень;
- виконуються всі цикли (у межах їх меж і діапазонів);
- аналізується правильність внутрішніх структур даних.

Недоліки тестування «білого ящика»:

1. Вичерпне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.
2. Кількість незалежних маршрутів може бути дуже велика. Наприклад, якщо цикл в програмі виконується k разів, а всередині циклу мається n розгалужень, то кількість маршрутів обчислюється за формулою: $m = \sum_{i=1}^{k} n^i$.

При $n = 5$ і $k = 20$ кількість маршрутів $m = 10^{14}$. Приймемо, що на розробку, виконання та оцінку тесту по одному маршруту витрачається 1 мс. Тоді при роботі 24 години на добу 365 днів у році на тестування піде 3170 років.

3. В програмі можуть бути пропущені деякі маршрути.
4. Не можна виявити помилки, поява яких залежить від даних (це помилки, обумовлені виразами типу if abs (ab)<eps ..., if (a+b+c)/3 = a ...).

Переваги тестування «білого ящика» пов'язані з тим, що принцип «білого ящика» дозволяє врахувати особливості програмних помилок:

1. Кількість помилок мінімально в «центрі» і максимально на «периферії» програми.
2. Попередні припущення про ймовірність потоку управління або даних у програмі часто бувають некоректними. В результаті типовим може стати маршрут, модель обчислень по якому опрацьована мало.
3. При запису алгоритму ПЗ у вигляді тексту на мові програмування можливо внесення типових помилок трансляції (синтаксичних і семантичних).
4. Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Кожна з цих причин є аргументом для проведення тестування за принципом «білого ящика». Тести «чорного ящика» не зможуть реагувати на помилки таких типів.

2.2. Тестування циклів

Цикл - найбільш поширена конструкція алгоритмів, реалізованих в ПЗ. Тестування циклів проводиться за принципом «білого ящика», при перевірці циклів основна увага звертається на правильність конструкцій циклів.

Розрізняють *4 типи циклів:* прості, вкладені, об'єднані, неструктуровані. Структура циклів наведено на рис. 22.

Розглянемо прості цикли. Для перевірки простих циклів з кількістю повторень *n* може використовуватися один з наступних наборів тестів:

1) прогін всього циклу;
2) тільки один прохід циклу;
3) два проходи циклу;
4) *m* проходів циклу, де $m < n$,
5) $n - 1$, n, $n + 1$ проходів циклу.

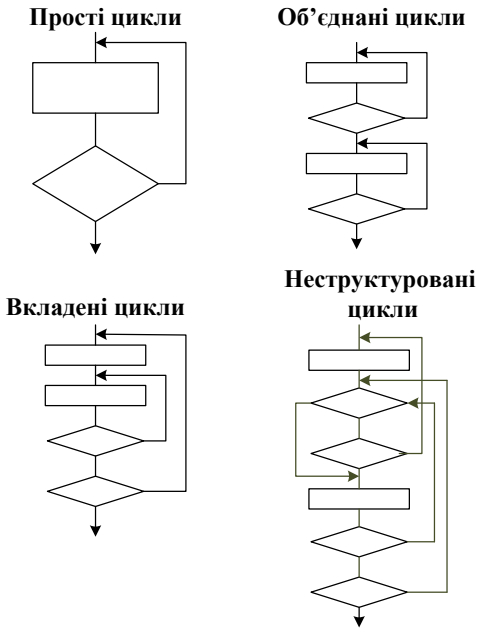

Рис. 22. Типові структури циклів

Для вкладених циклів за умови збільшення рівня вкладеності циклів, різко зростає кількість можливих шляхів. Це призводить до нереалізуючої кількості тестів. Для скорочення кількості тестів застосовується спеціальна методика, в якій використовуються такі поняття, як обємний і вкладений цикли (рис. 23).

Порядок тестування вкладених циклів ілюструється на рис. 24.

Рис. 23. Обємний і вкладений цикли

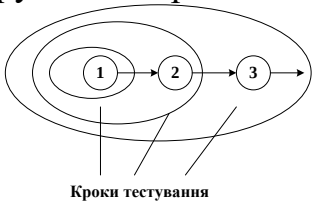

Рис. 24. Кроки тестування вкладених циклів

Визначені наступні *кроки тестування:*

1. Вибирається самий внутрішній цикл. Встановлюються мінімальні значення параметрів всіх інших циклів.
2. Для внутрішнього циклу проводяться тести простого циклу. Додаються тести для виключених значень і значень, що виходять за межі робочого діапазону.
3. Переходять в наступний по порядку осяжний цикл. Виконують його тестування. При цьому зберігаються мінімальні значення параметрів для всіх зовнішніх (охоплюючих) циклів і типові значення для всіх вкладених циклів.
4. Робота продовжується до тих пір, поки не будуть протестовані всі цикли.

Розглянемо об'єднані цикли. Якщо кожен із циклів незалежний від інших, то використовується техніка тестування простих циклів. За наявності залежності (наприклад, кінцеве значення лічильника першого циклу використовується як початкове значення лічильника другого циклу) використовується методика для вкладених циклів.

Неструктуровані цикли тестуванню не підлягають. Цей тип циклів повинен бути перероблений за допомогою структурованих програмних конструкцій.

2.3. Спосіб тестування потоків даних

У попередніх способах тести будувалися на основі аналізу керуючої структури програми. У даному способі аналізу піддається інформаційна структура програми. Роботу будь-якої програми можна розглядати як обробку потоку даних, переданих від входу в програму до її виходу.

Розглянемо приклад. Нехай потоковий граф програми має вигляд, представлений на рис. 25. У ньому суцільні дуги - це зв'язки з управління між операторами в програмі. Пунктирні дуги відзначають інформаційні зв'язки (зв'язки по потоках даних).

Окреслені тут інформаційні зв'язки відповідають наступним допущенням:

- у вершині 1 визначаються значення змінних *a, b;*
- значення змінної *a* використовується в вершині 4;
- значення змінної *b* використовується в вершинах 3, 6;
- у вершині 4 визначається значення змінної c, яка використовується в вершині 6.

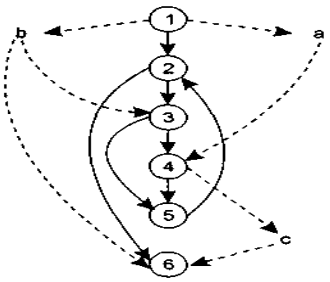

Рис. 25. Граф програми з керуючими та інформаційними зв'язками

У загальному випадку для кожної вершини графа можна записати:

- безліч визначень даних: DEF (i) = {x | i-а вершина містить визначення x};
- безліч використань даних: USE (i) = {x | i -а вершина використовує x}.

Під *визначенням даних* розуміють дії, які змінюють елемент даних. Ознака визначення - ім'я елемента стоїть в лівій частині оператора присвоєння: *x:=f*(...).

Використання даних - це застосування елементу у виразі, де відбувається звернення до елементу даних, але не зміна елемента. Ознака використання - ім'я елемента варто в правій частині оператора присвоювання: □=*f*(x).

Тут місце підстановки іншого імені відзначено прямокутником (прямокутник грає роль мітки-заповнювача).

Назвемо *DU-ланцюжком (ланцюжком визначення-використання)* конструкцію *[x, i, j],* де *i, j* - імена вершин; x визначена в *i-й* вершині *(x* є DEF*(i))*та використовується в *j-й* вершині *(x* є USE*(j))*.

У нашому прикладі існують наступні DU-ланцюжки:

[a, 1,4], [b, 1,3], [b, 1, 6], [з, 4, 6].

Спосіб *DU-тестування* вимагає охоплення всіх DU-ланцюжків програми. Таким чином, розробка тестів тут проводиться на основі аналізу життя всіх даних програми.

Очевидно, що для підготовки тестів потребується виділення маршрутів - шляхів виконання програми на керуючому графові. Критерій для вибору шляху - покриття максимальної кількості DU-ланцюжків.

Кроки способу DU-тестування:

1) побудова керуючого графа (УГ) програми;
2) побудова інформаційного графа (ІГ);
3) формування повного набору DU-ланцюжків;
4) формування повного набору відрізків шляхів в керуючому графові (відображенням набору DU-ланцюжків інформаційного графа, рис. 26);
5) побудова маршрутів - повних шляхів на керуючому графові, що покривають набір відрізків шляхів керуючого графа;
6) підготовка тестових варіантів.

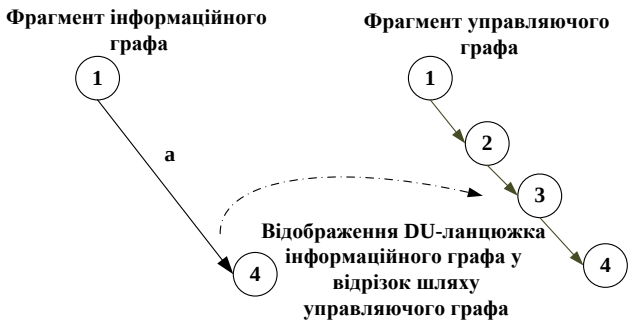

Рис. 26. Відображення DU-ланцюжки *в* відрізок шляху

Переваги DU-тестування:

- простота необхідного аналізу операційно-керуючої структури програми;
- простота автоматизації.

Недоліки DU –тестування - це труднощі у виборі мінімальної кількості максимально ефективних тестів.

Область використання DU-тестування: програми з вкладеними умовними операторами і операторами циклу.

2.4. Способи тестування умов

Мета цього сімейства способів тестування - будувати тестові варіанти для перевірки логічних умов програми. При цьому бажано забезпечити охоплення операторів з усіх гілок програми. Проста умова - булева змінна або вираз відносини.

Вираз відносини має вигляд:

E1 <оператор відносини> E2,

де El, E2 - арифметичні вираження, а в якості оператора відносини використовується один з наступних операторів : <, >, =, $\neq$, $\leq$, $\geq$.

Складена умова складається з декількох простих умов, булевих операторів і круглих дужок. Будемо застосовувати булеві оператори OR, AND (&), NOT. Умови, які не містять виразів відносини, називають булевими виразами.

Елементами умови є: булеві оператори, булева змінна, пара дужок (яка включає просту або складену умову), оператор відносини, арифметичний вираз. Ці елементи визначають типи помилок в умовах.

Якщо умова некоректна, то некоректний щонайменше один з елементів умови.

Отже, в умові можливі наступні типи помилок:

- помилка булевого оператора (наявність некоректних / відсутніх / надлишкових булевих операторів);
- помилка булевої змінної;
- помилка булевої дужки;
- помилка оператора відносини;
- помилка арифметичного виразу.

Спосіб тестування умов орієнтований на тестування кожної умови в програмі. Методики тестування умов мають *дві переваги*:

1) достатньо просто виконати вимірювання тестового покриття умови;

2) тестове покриття умов в програмі - це фундамент для генерації додаткових тестів програми.

Метою тестування умов є визначення не тільки помилок в умовах, але й інших помилок в програмах. Якщо набір тестів для програми А ефективний для виявлення помилок в умовах, що містяться в А, то ймовірно, що цей набір також ефективний для виявлення інших помилок в А. Крім того, якщо методика тестування ефективна для виявлення помилок в умові, то ймовірно, що ця методика буде ефективна для виявлення помилок в програмі.

Існує дві методики тестування умов:

1) найпростіша методика - *тестування гілок* . Тут для складеного умови *С* перевіряється:

- кожна проста умова (входить в нього);
- True-гілка;
- False-гілку.

2) інша методика - *тестування області визначення.* В ній для вираження ставлення вимагається генерація 3-4 тестів. Вираз виду:

E1 <оператор відносини> E2

перевіряється трьома тестами, які формують значення Е1 більшим, ніж Е2, рівним Е2 і меншим, ніж Е2. Якщо оператор відносини неправильний, а Е1 і Е2 коректні, то ці три тести гарантують виявлення помилки оператора відносини. Для визначення помилок в Е1 і Е2 тест повинен сформувати значення Е1 більшим чи меншим, ніж Е2, причому забезпечити якомога меншу різницю між цими значеннями.

Для булевих виразів з n змінними потрібно набір з 2^n тестів. Цей набір дозволяє виявити помилки булевих операторів, змінних і дужок, але практичний тільки при малому n. Втім, якщо в булевому виразі кожна булева змінна входить лише один раз, то кількість тестів легко зменшується.

Контрольні питання

1. Визначте поняття тестування.
2. Що таке тест? Поясніть зміст процесу тестування.
3. Що таке вичерпне тестування?
4. Які завдання вирішує тестування?
5. Яких завдань не вирішує тестування?
6. Які принципи тестування ви знаєте? У чому їх відмінність один від одного?
7. У чому полягає суть тестування «чорного ящика»?

8. У чому полягає суть тестування «білого ящика»?

9. Які особливості тестування «білого ящика»?

10. Які недоліки має тестування «білого ящика»?

11. Які переваги має тестування «білого ящика»?

12. Дайте характеристику способу тестування базового шляху.

13. Які особливості має потоковий граф?

14. Поясніть поняття незалежного шляху.

15. Дайте загальну характеристику способів тестування умов.

16. Які типи помилок в умовах ви знаєте?

17. Які методики тестування умов ви знаєте?

18. Поясніть суть способу тестування гілок і операторів відносин. Які він має обмеження?

19. Поясніть переваги, недоліки та область застосування способу тестування гілок і операторів відносин.

20. Поясніть суть способу тестування потоків даних.

21. Що таке безліч визначень даних?

22. Що таке безліч використань даних?

23. Поясніть кроки способу тестування потоків даних.

24. Поясніть переваги, недоліки та область застосування способу тестування потоків даних.

ЛЕКЦІЯ 5. ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ. ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

План лекції

1. Другий спосіб тестування – функціональне тестування. Поняття «чорного ящика».

2. Організація процесу тестування програмного забезпечення.

1. Функціональне тестування

Відомі функції програми. Досліджується робота кожної функції на всій області визначення. Як показано на рис. 27, основне місце використання тестів «чорного ящика» - інтерфейс ПЗ.

Ці тести демонструють:

- як виконуються функції програм;
- як приймаються вихідні дані;
- як виробляються результати;
- як зберігається цілісність зовнішньої інформації.

При тестуванні «чорного ящика» розглядаються системні характеристики програм, ігнорується їх внутрішня логічна структура. Вичерпне тестування, як правило, неможливе. Наприклад, якщо в програмі 10 вхідних величин і кожна приймає по 10 значень, то потрібно 10^{10} тестових варіантів. Відзначимо також, що тестування «чорного ящика» не реагує на багато особливостей програмних помилок.

1.1. Особливості тестування «чорного ящика»

Тестування «чорного ящика» дозволяє отримати комбінації вхідних даних, які забезпечують повну перевірку всіх функціональних вимог до програми. Програмний виріб тут розглядається як «чорний ящик», чию поведінку можна визначити тільки дослідженням його входів і відповідних виходів. При такому підході бажано мати:

- набір, утворений такими вхідними даними, які приводять до аномалій поведінки програми (назвемо його *ІТ);*
- набір, утворений такими вихідними даними, які демонструють дефекти програми (назвемо його *ОТ).*

Як показано на рис. 28, будь-який спосіб тестування «чорного ящика» повинен:

- виявити такі вхідні дані, які з високою ймовірністю належать набору ІТ;
- сформулювати такі очікувані результати, які з високою ймовірністю є елементами набору ОТ.

У багатьох випадках визначення таких тестових варіантів ґрунтується на попередньому досвіді інженерів тестування. Вони використовують своє знання і розуміння галузі визначення для ідентифікації тестових варіантів, які ефективно виявляють дефекти. Тим не менш систематичний підхід до виявлення тестових даних, обговорюваний в даній главі, може використовуватися як корисне доповнення до евристичного знання.

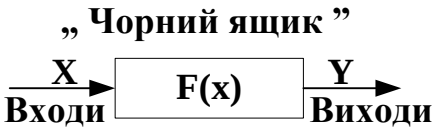

Рис. 27. Тестування «чорного ящика»

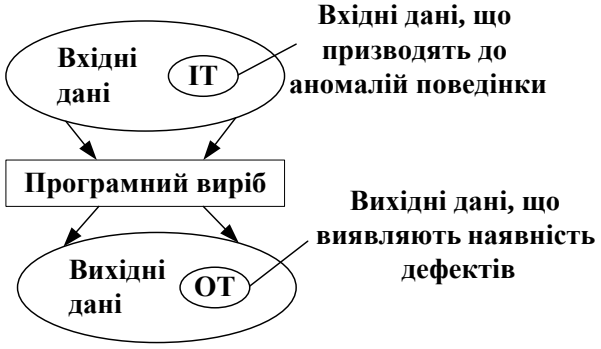

Рис. 28. Тестування «чорного ящика»

Принцип «чорного ящика» не альтернативний принципу «білого ящика». Скоріше це доповнює підхід, який виявляє інший клас помилок.

Тестування «чорного ящика» забезпечує пошук наступних категорій помилок:

1) некоректних або відсутніх функцій;

2) помилок інтерфейсу;

3) помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних;

4) помилок характеристик (необхідна ємність пам'яті і т.д.);

5) помилок ініціалізації і завершення.

Подібні категорії помилок способами «білого ящика» не виявляються.

На відміну від тестування «білого ящика», яке виконується на ранній стадії процесу тестування, тестування «чорного ящика» застосовують на пізніх стадіях тестування. При тестуванні «чорного ящика» нехтують керуючою структурою програми. Тут увага концентрується на інформаційній галузі визначення програмної системи.

Техніка «чорного ящика» орієнтована на *вирішення наступних* завдань:

- скорочення необхідної кількості тестових варіантів (через перевірки не статичних, а динамічних аспектів системи);
- виявлення класів помилок, а не окремих помилок.

1.2. Спосіб діаграм причин-наслідків

Діаграми причинно-наслідкових зв'язків - спосіб проектування тестових варіантів, який забезпечує формальний запис логічних умов і відповідних дій. Використовується автоматний підхід до вирішення завдання.

Кроки способа:

1) для кожного модуля перераховуються причини (умови введення або класи еквівалентності умов введення) та слідства (дії або умови виведення). Кожній причині і наслідку присвоюється свій ідентифікатор;

2) розробляється граф причинно-наслідкових зв'язків;

3) граф перетвориться в таблицю рішень;

4) стовпці таблиці рішень перетворюються в тестові варіанти.

Зобразимо базові символи для запису графів причин і наслідків (cause-effect graphs) .

Зробимо попередні *зауваження:*

1) причини будемо позначати символами c_i, а слідства - символами e_i;

2) кожен вузол графа може перебувати в стані 0 або 1 (0 - стан відсутня, 1 - стан присутня).

Функція *тотожність* (рис. 29) встановлює, що якщо значення $c_1 \in 1$, то і значення $e_1 \in 1$, в іншому випадку значення $e_1 \in 0$.

Функція *не* (рис. 30) встановлює, що якщо значення $c_1 \in 1$, то значення $e_1 \in 0$; у противному випадку значення $e_1 \in 1$.

Функція *або* (рис. 31) встановлює, що якщо c_1 або c_2 є 1, то e_1 є 1, в іншому випадку e_1 є 0.

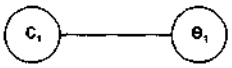

Рис. 29. Функція *тотожність*

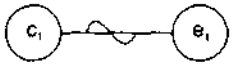

Рис. 30. Функція *не*

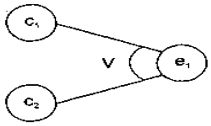

Рис. 31. Функція *або*

Функція *і* (рис. 32) встановлює, що якщо і c_1 і c_2 є 1, то e_1 є 1, в іншому випадку e_1 є 0.

Часто певні комбінації причин неможливі через синтаксичні або зовнішні обмеження. Використовуються перераховані нижче позначення обмежень.

Обмеження *E* (виключає, Exclusive, рис. 33) встановлює, що Е має бути істинним, якщо хоча б одна з причин - *a* чи *b* - приймає значення 1 *(a* і *b* не можуть приймати значення 1 одночасно) .

Обмеження *I* (включає, Inclusive, рис. 34) встановлює, що принаймні одна з величин, *a, b,* або с, завжди повинна бути рівною 1*(a,b* і *c* не можуть приймати значення 0 одночасно).

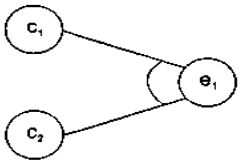

Рис. 32. Функція *і*

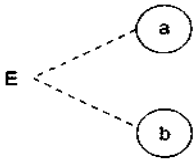

Рис. 33. Обмеження *E* (виключає, Exclusive)

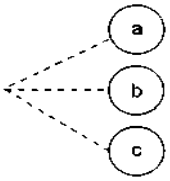

Рис. 34. Обмеження *I* (включає, Inclusive)

Обмеження *O* (одне і тільки одне, Only one, рис. 35) встановлює, що одна і тільки одна з величин *a* чи *b* повинна бути дорівнює 1.

Обмеження *R* (вимагає, Requires, рис. 36) встановлює, що якщо *a* приймає значення 1, то і *b* повинна приймати значення 1 (не можна, щоб *a* дорівнювало 1, а *b* - 0). Часто виникає необхідність в обмеженнях для наслідків.

Обмеження *M* (приховує, Masks, рис. 37) встановлює, що якщо слідство *a* має значення 1, то наслідок *b* повинно прийняти значення 0.

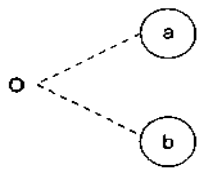

Рис. 35. Обмеження *O* (одне і тільки одне, Only one)

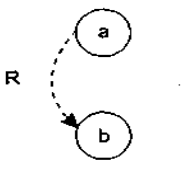

Рис. 36. Обмеження R (вимагає, Requires)

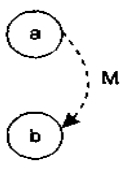

Рис. 37. Обмеження M (приховує, Masks)

2. Організація процесу тестування програмного забезпечення

Методика тестування ПС може бути представлена у вигляді розгортаючої спіралі. На початку здійснюється *тестування елементів (модулів), що* перевіряє результати етапу *кодування* ПС. На другому кроці виконується *тестування інтеграції,* орієнтоване на виявлення помилок етапу *проектування* ПС. На третьому обороті спіралі проводиться *тестування правильності, що* перевіряє коректність етапу *аналізу вимог* до ПС. На заключному витку спіралі проводиться *системне тестування, що* виявляє дефекти етапу *системного аналізу* ПС.

Охарактеризуємо кожен крок процесу тестування:

1. *Тестування елементів.* Мета - індивідуальна перевірка кожного модуля. Використовуються способи тестування «білого ящика».
2. *Тестування інтеграції.* Мета - тестування зборки модулів в програмну систему. В основному застосовують способи тестування «чорного ящика».
3. *Тестування правильності.* Мета - перевірити реалізацію в програмній системі всіх функціональних і поведінкових вимог, а також вимоги ефективності. Використовуються виключно способи тестування «чорного ящика».
4. *Системне тестування.* Мета - перевірка правильності об'єднання і взаємодії всіх елементів комп'ютерної системи, реалізації всіх системних функцій.

Рис. 38. Спіраль процесу тестування ПС

Організація процесу тестування у вигляді еволюційної розгортаючої спіралі забезпечує максимальну ефективність пошуку помилок.

Науковий підхід полягає в застосуванні математичної моделі відмов. Наприклад, для логарифмічної моделі Пуассона формула розрахунку поточної інтенсивності відмов має вигляд:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \qquad (1)$$

де $\lambda(t)$ - поточна інтенсивність програмних відмов (кількість відмов в одиницю часу); λ_0 - початкова інтенсивність відмов (на початку тестування);

p – експоненційне зменшення інтенсивності відмов за рахунок виявлених й усунутих помилок; *t-час* тестування.

За допомогою рівняння (1) можна передбачити зниження помилок в ході тестування, а також час, потрібний для досягнення допустимо низької інтенсивності відмов.

2.1. Тестування елементів

Об'єктом тестування елементів є найменша одиниця проектування ПС - модуль. Для виявлення помилок в рамках модуля тестуються його найважливіші керуючі шляху. Відносна складність тестів і помилок визначається як результат обмежень галузі тестування елементів. Принцип тестування - «білий ящик», крок може виконуватися для набору модулів паралельно.

Тестуванню піддаються:

- інтерфейс модуля;
- внутрішні структури даних;
- незалежні шляхи;
- шляхи обробки помилок;
- граничні умови.

Інтерфейс модуля тестується для перевірки правильності введення-виведення тестової інформації. Якщо немає впевненості в правильному введенні-виведенні даних, немає сенсу проводити інші тести.

Дослідження внутрішніх структур даних гарантує цілісність даних, що зберігається.

Тестування незалежних шляхів гарантує однократне виконання всіх операторів модуля. При тестуванні шляхів виконання виявляються наступні категорії помилок: помилкові обчислення, некоректні порівняння, неправильний потік управління.

Найбільш загальними помилками обчислень є:

1) неправильний або незрозумілий пріоритет арифметичних операцій;
2) змішана форма операцій;
3) некоректна ініціалізація;
4) неузгодженість в представленні точності;
5) некоректне символічне представлення виразів.

2.2. Тестування інтеграції

Тестування інтеграції підтримує збірку цільної програмної системи. Мета збірки і тестування інтеграції: взяти модулі, протестовані як елементи, і побудувати програмну структуру, необхідну проектом.

Тести проводяться для виявлення помилок інтерфейсу. Перерахуємо деякі категорії помилок інтерфейсу:

- втрата даних при проходженні через інтерфейс;

- відсутність в модулі необхідного посилання;
- несприятливий вплив одного модуля на інший;
- підфункції при об'єднанні не утворюють необхідну головну функцію;
- окремі (допустимі) неточності при інтеграції виходять за допустимий рівень;
- проблеми при роботі з глобальними структурами даних.

2.3. Тестування правильності

Після закінчення тестування інтеграції програмна система зібрана в єдиний корпус, інтерфейсні помилки виявлені і відкориговані. Тепер починається останній крок програмного тестування - *тестування правильності.* Мета - підтвердити, що функції, описані у специфікації вимог до ПС, відповідають очікуванням замовника.

Підтвердження правильності ПС виконується за допомогою тестів «чорного ящика», що демонструють відповідність вимогам. При виявленні відхилень від специфікації вимог створюється список недоліків. Як правило, відхилення і помилки, виявлені при підтвердженні правильності, вимагають зміни термінів розробки продукту.

Важливим елементом підтвердження правильності є перевірка конфігурації ПС. Конфігурацією ПС називають сукупність усіх елементів інформації, що виробляються в процесі конструювання ПС.

Мінімальна конфігурація ПС включає наступні базові елементи:

1) системну специфікація;
2) план програмного проекту;
3) специфікацію вимог до ПС; працюючий або паперовий макет;
4) попереднє керівництво користувача;
5) специфікація проектування;
6) лістінги вихідних текстів програм;
7) план і методику тестування; тестові варіанти і отримані результати;
8) керівництва по роботі та інсталяції;
9) .exe-код виконуваної програми;
10) опис бази даних;
11) керівництво користувача по налаштуванню;
12) документи супроводу; звіти про проблеми ПС ; запити супроводу; звіти про конструкторські зміни;
13) стандарти та методики конструювання ПС.

2.4. Системне тестування

Системне тестування передбачає вихід за рамки області дії програмного проекту і проводиться не тільки програмним розробником. Класична проблема системного тестування - вказівка причини. Вона

виникає, коли розробник одного системного елемента звинувачує розробника іншого елемента в причині виникнення дефекту.

Для захисту від подібного звинувачення розробник програмного елемента повинен:

1) передбачити засоби обробки помилок, які тестують всі вводи інформації від інших елементів системи;

2) провести тести, моделюючі невдалі дані або інші потенційні помилки інтерфейсу ПС;

3) записати результати тестів, щоб використовувати їх як доказ невинуватості у випадку «зазначення причини»;

4) взяти участь в плануванні та проектуванні системних тестів, щоб гарантувати адекватне тестування ПС.

Системні тести повинні перевіряти, що всі системні елементи правильно об'єднані і виконують призначені функції.

2.5. Мистецтво налагодження

Налагодження - це локалізація і усунення помилок, якщо тестовий варіант виявляє помилку, то процес налагодження знищує її.

Процес налагодження намагається зіставити симптом з причиною, внаслідок чого призводить до виправлення помилки. Можливі *два результати процесу* налагодження:

1) причина знайдена, виправлена, знищена;

2) причина не знайдена, тобто відладчик може припускати причину. Для перевірки цієї причини він просить розробити додатковий тестовий варіант, який допоможе перевірити припущення. Таким чином, запускається ітераційний процес корекції помилки.

Можливі різні способи прояву помилок:

1) програма завершується нормально, але видає неправильні результати;

2) програма зависає;

3) програма завершується з переривання;

4) програма завершується, видає очікувані результати, але збережені дані зіпсовані (це самий неприємний варіант).

Характер прояву помилок також може мінятися. Симптом помилки може бути:

- постійним;
- мерехтливим;
- пороговим (проявляється при перевищенні деякого порога в обробці - 200 літаків на екрані відстежуються, а 201-го - немає);
- відкладеним (проявляється тільки після виправлення маскуючих помилок).

Розрізняють *дві групи методів* налагодження:

1) аналітичні - базуються на аналізі вихідних даних для тестових прогонів;

2) експериментальні - базуються на використанні допоміжних засобів налагодження (налагоджування друку, трасування), що дозволяють уточнити характер поведінки програми при тих чи інших вихідних даних.

Загальна *стратегія налагодження* - зворотне проходження від поміченого симптому помилки до вихідної аномалії (місцем в програмі, де помилка здійснена).

Мета налагодження - знайти оператор програми, при виконанні якого правильні аргументи приводять до неправильних результатів. Якщо місце прояви симптому помилки не є шуканої аномалією, то один з аргументів оператора повинен бути невірним. Тому треба перейти до дослідження попереднього оператора, який виробив цей невірний аргумент. У підсумку покрокове зворотне простежування призводить до шуканого помилкового місця.

Основна *перевага аналітичних методів налагодження* полягає в тому, що вихідна програма залишається без змін.

В експериментальних методах для простежування виконується:

1. Видача значень змінних у зазначених точках.
2. Трасування змінних (видача їх значень при кожній зміні).
3. Трасування потоків управління (імен викликаємих процедур, міток, на які передається управління, номерів операторів переходу).

Перевага експериментальних методів налагодження полягає в тому, що основна рутинна робота з аналізу процесу обчислень перекладається на комп'ютер. Багато трансляторів мають вбудовані засоби налагодження для отримання інформації про хід виконання програми.

Недолік експериментальних методів налагодження - в програму вносяться зміни, при виключенні яких можуть з'явитися помилки. Втім, деякі системи програмування створюють спеціальний налагоджувальний екземпляр програми, а в основній екземпляр не вміщаються.

Контрольні питання

1. Поясніть особливості тестування циклів.
2. Які методики тестування простих циклів ви знаєте?
3. Які кроки тестування вкладених циклів?
4. У чому суть способу діаграм причин-наслідків?
5. Дайте загальну характеристику графа причинно-наслідкових зв'язків.
6. Які обмеження використовуються в графі причин і наслідків?
7. Поясніть кроки способу діаграм причин-наслідків.
8. Поясніть суть методики тестування програмної системи.

9. Коли і навіщо виконується тестування елементів? Який етап конструювання воно перевіряє?
10. Коли і навіщо виконується тестування інтеграції? Який етап конструювання воно перевіряє?
11. Коли і навіщо виконується тестування правильності? Який етап конструювання воно перевіряє?
12. Коли і навіщо виконується системне тестування? Який етап конструювання воно перевіряє?
13. Поясніть суть тестування елементів.
14. У чому мета тестування інтеграції?
15. Які категорії помилок інтерфейсу ви знаєте?
16. В чому суть тестування правильності?
17. Які елементи включає мінімальна конфігурація програмної системи?
18. В чому суть системного тестування?
19. Що таке відладка?

ЛІТЕРАТУРА

1. *Орлов С.* Технологии разработки программного обеспечения: Учебник. — СПб.: Питер, 2002. — 464 с.: ил.
2. *Бадд Т.* Объектно-ориентированное программирование в действии. - СПб. «Питер».: 1997. - 320 с.
3. *Арлоу Д., Нейштадт А.* UML 2 и Унифицированный процесс: практический объектно-ориентированный анализ и проектирование. Символ. - М.: 2007. - 624 с.
4. *Буч Г., Рамбо Д., Джекобсон А.* UML. Руководство пользователя. - М.: ДМК, 2000.
5. *Калянов Г.* CASE. Структурный системный анализ. - М.:ЛОРИ, 1996. — 242 с.
6. *Вендров А.М.* CASE-технологии. Современные методы и средства проектирования информационных систем. - М., 1997. — 346 с.
7. *Фаулер М.* Архитектура корпоративных программных приложений. Пер. с англ. – М.: Издательский дом “Вильямс”, 2004. – 544 с.
8. *Федотова Д. Э., Семенов Ю. Д., Чижик К. Н.* CASE-технологии: Практикум. - М., 2005. - 60 с.
9. *Роджерсон Д.* Основы COM. - М.: Русская редакция, 1997. - 370 с.
10. *Вендров А.М.* Практикум по проектированию программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2002.