МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВСП «ФАХОВИЙ КОЛЕДЖ ПРОМИСЛОВОЇ АВТОМАТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
ОДЕСЬКОГО НАЦІОНАЛЬНОГО ТЕХНОЛОГІЧНОГО УНІВЕРСИТЕТУ»

ЗАТВЕРДЖУЮ
Заступник директора з
навчально-методичної роботи
ФКПАІТ ОНТУ
з навчально-методичної роботи
_____ Вікторія ОКСАНІЧЕНКО
_____._____.20___ року

Методичні вказівки для самостійної роботи

з дисципліни Конструювання програмного забезпечення
(Назва дисципліни)

галузі знань 12 «Інформаційні технології»
(шифр і назва галузі знань)

спеціальності 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

освітня програма Інженерія програмного забезпечення
(назва освітньо-професійної програми)

Одеса 2022

Методичні вказівки для самостійної роботи з дисципліни «Конструювання програмного забезпечення» для підготовки фахових молодших бакалаврів за спеціальністю 121 «Інженерія програмного забезпечення».

Укладач: викладач ФКПАІТ ОНТУ Т. П. Костиренко

Розглянуто та затверджено на засіданні циклової комісії комп'ютерних наук та інженерії програмного забезпечення ФКПАІТ ОНТУ

Протокол від ___._____.20___ року, № ____

Голова циклової комісії

_____Тетяна КОСТИРЕНКО

(підпис) (власне ім'я та прізвище)

___._____.20___ року

Методичні вказівки для самостійної роботи студентів розроблені згідно навчальної програми з предмету конструювання програмного забезпечення.

Предметом вивчення навчальної дисципліни є основи конструювання програмного забезпечення та уніфікована мова моделювання програмного забезпечення UML.

Метою вивчення навчальної дисципліни є оволодіння студентами теоретичних знань та набуття практичних навиків з дисципліни конструювання програмного забезпечення, необхідного для спеціальної підготовки та майбутньої професійної діяльності.

Основними завданнями вивчення дисципліни є формування у студентів базових уявлень про конструювання програмного забезпечення та основи моделювання програмного забезпечення; інтелектуальний розвиток особистості, розвиток у студентів логічного мислення і просторової уяви, алгоритмічної, інформаційної та графічної культури, пам’яті, уваги, інтуїції; формування у студентів компетенцій за фахом.

Зміст

Блок змістових модулів 1 – Мови конструювання

Змістовий модуль 1.1 - Організація процесу конструювання. Керівництво програмним проектом

Тема №1: Ваговиті і полегшені процеси

(2 години)

Мета: Ознайомитись з історією виникнення великовагових та полегшених процесів, розглянути їх основні недоліки.

План:

1. Великовагові й полегшені процеси:
    a. великовагові (heavyweight) процеси;
    b. полегшені (lightweight) процеси;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Чим відрізняються великовагові процеси від полегшених процесів?
2. Чим відрізняються великовагові процеси від прогнозуючих процесів?
3. Чим відрізняються рухливі процеси від полегшених процесів?
4. Перелічите гідності й недоліки великовагових процесів.
5. Перелічите гідності й недоліки полегшених процесів.
6. Приведіть приклади великовагових процесів.
7. Приведіть приклади полегшених процесів.

Великовагові й полегшені процеси

В XXI столітті потреби суспільства в програмному забезпеченні інформаційних технологій досягли екстремальних значень. Програмна індустрія буквально "захлинається" від потоку найрізноманітніших замовлень. "Більше

процесів розробки, гарних і різних!" - скандують замовники. "Зараз, зараз! Тільки про це й думаємо!" - відповідають розроблювачі.

Традиційно для впорядкування й прискорення програмних розробок пропонувалися, що строго впорядковують *великовагові (heavyweight) процеси*. У цих процесах прогнозується весь обсяг майбутніх робіт, тому вони називаються прогнозуючими (predictive) процесами. Порядок, який повинен виконувати при цьому людина-розроблювач, надзвичайно строгий - "крок вправо, крок уліво - віртуальний розстріл!". Іншими словами, людські слабості в розрахунки не ухвалюються, а обсяг необхідної документації здатний відняти спокій і сон в "совісного" розроблювача.

В останні роки з'явилася група нових, *полегшених (lightweight) процесів*. Тепер їх називають рухливими (agile) процесами. Вони привабливі відсутністю бюрократизму, характерного для великовагових (прогнозуючих) процесів. Нові процеси повинні втілити в життя розумний компроміс між занадто строгою дисципліною й повною її відсутністю. Інакше кажучи, порядку в них досить для того, щоб одержати розумну віддачу від розроблювачів.

Рухливі процеси вимагають меншого обсягу документації й орієнтовані на людину. У них явно зазначене на необхідність використання природних якостей людської натури (а не на застосування дій, спрямованих всупереч цим якостям).

Більше того, рухливі процеси враховують особливості сучасного замовника, а саме часті зміни його вимог до програмного продукту. Відомо, що для прогнозуючих процесів часті зміни вимог подібні смерті. На відміну від них, рухливі процеси адаптують зміни вимог і навіть виграють від цього. Словом, рухливі процеси мають адаптивну природу.

Таким чином, у сучасній інфраструктурі програмної інженерії існують два сімейства процесів розробки:

- сімейство прогнозуючих (великовагових) процесів;
- сімейство адаптивних (рухливих, полегшених) процесів.

У кожного сімейства є свої гідності, недоліки й область застосування:

 адаптивний процес використовують при частих змінах вимог, нечисленній групі висококваліфікованих розроблювачів і грамотному замовнику, який згодний брати участь у розробці;

 прогнозуючий процес застосовують при фіксованих вимогах і численній групі розроблювачів різної кваліфікації.

Тема №2: XP - процес

(2 години)

Мета: Ознайомитись з основною ідеєю XP-проектування. Розглінути методи екстремального проектування, а також ознайомитись з моделями якості процесів конструювання.

План:

1. XP-Процес;
2. Методи XP-процесу;
3. Моделі якості процесів конструювання.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Перелічите характеристики Хр- Процесу.
2. Перелічите методи Хр- Процесу.
3. У чому полягає головна особливість Хр- Процесу?
4. Охарактеризуйте зміст гри планування в Хр- Процесі.
5. Охарактеризуйте призначення метафори в Хр- Процесі.
6. Яка особливість проектування в Хр- Процесі?
7. Яка особливість програмування в Хр- Процесі?
8. Що таке реорганізація?
9. Що таке колективне володіння?

10. Яка особливість тестування в Хр- Процесі?
11. Чим відрізняється Хр- Реалізація від Хр- Ітерації?
12. Чим Хр- Реалізація схожа на Хр- Ітерацію?
13. Яка тривалість Хр- Реалізації?
14. Яка тривалість Хр- Ітерації?
15. Яка максимальна чисельність групи Хр- Розроблювачів?
16. Які моделі якості процесів конструювання ви знаєте?
17. Охарактеризуйте модель СММ.
18. Охарактеризуйте рівень зрілості знайомій вам фірми.

ХР-Процес

Екстремальне програмування (extreme Programming, ХР) - полегшений (рухливий) процес (або методологія), головний автор якого - Кент Бек (1999). Хр-Процес орієнтований на групи малого й середнього розміру, що будують програмне забезпечення в умовах невизначених або швидко мінливих вимог.Хр-Групу утворюють до 10 співробітників, які розміщаються в одному приміщенні.

Основна ідея ХР - усунути високу вартість зміни, характерну для додатків з використанням об'єктів, пат тернів (Паттерн є розв'язком типової проблеми в певному контексті) і реляційних баз даних. Тому ХР - Процес повинен бути высокодинамичним процесом. ХР- Група має справу зі змінами вимог на всьому протязі ітераційного циклу розробки, причому цикл складається з дуже коротких ітерацій. Чотирма базовими діями в ХР- Циклі є: кодування, тестування, вислуховування замовника й проектування. Динамізм забезпечується за допомогою чотирьох характеристик: безперервному зв'язку із замовником (і в межах групи), простоти (завжди вибирається мінімальний розв'язок), швидкому зворотному зв'язка (за допомогою модульного й функціонального тестування), сміливості в проведенні профілактики можливих проблем.

Більшість принципів, підтримуваних у ХР (мінімальність, простота, еволюційний цикл розробки, мала тривалість ітерації, участь користувача, оптимальні стандарти кодування і т.д.), продиктовані здоровим глуздом і застосовуються в будь-якому впорядкованому процесі. Просто в ХР ці принципи, як показано в табл. 1, досягають "екстремальних значень".

Практика здорового глузду	XP-Экстремум	XP-Реалізація
Перевірки коду	Код перевіряється увесь час	Парне програмування
Тестування	Тестування виконується увесь час, навіть за допомогою замовників	Тестування модуля, функціональне тестування
Проектування	Проектування є частиною щоденної діяльності кожного розроблювача	Реорганізація (refactoring)
Простота	Для системи вибирається найпростіший проектний розв'язок, що підтримує її поточну функціональність	Найпростіша річ, яка могла б працювати
Архітектура	Кожний постійно працює над уточненням архітектури	Метафора
Тестування інтеграції	Інтегрується й тестується кілька раз у день	Безперервна інтеграція
Короткі ітерації	Ітерації є гранично короткими, тривають секунди, хвилини, годинник, а не тижня, місяці або роки	Гра планування

Той, хто ухвалює принцип "мінімального розв'язку" за хакерство, помиляється, у дійсності XP - строго впорядкований процес. Прості розв'язки, що мають вищий пріоритет, у цей час розглядаються як найцінніші частини системи, на відміну від проектних розв'язків, які поки не потрібні, а можуть (в умовах зміни вимог і операційного середовища) і взагалі не знадобитися.

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 строки з погляду бізнесу, а розроблювачі оцінюють і прослідковують 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) - уся розробка проводиться на основі простій, загальнодоступної історії про те, як працює вся система.

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

5. Тестування (Testing) - безперервне написання тестів для модулів, які повинні виконуватися бездоганно; замовники пишуть тести для демонстрації закінченості функцій. "Тестуй, а потім кодуй" означає, що вхідним критерієм для написання коду є ", щовідмовив" тестовий варіант.

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

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

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

9. Безперервна інтеграція (Continuous integration) - система інтегрується й будується багато раз у день, у міру 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 більш 40 годин на тиждень. Не можна подвоювати робочий тиждень за рахунок понаднормових робіт.

11. Локальний замовник (On-Site customer) - у групі увесь час повинен перебувати представник замовника, дійсно готовий відповідати на запитання розроблювачів.

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

Гра планування й 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 розроблювачі й замовник повинні дійти згоди про те, які історії будуть здійснено в межах двох тижнів. Повну функціональність, необхідну замовникові, характеризує

пул історій; але для наступної двотижневої ітерації з пулу вибирається підмножина історій, найбільш важливе для замовника. У будь-який час у пул можуть бути додані нові історії, таким чином, вимоги можуть швидко змінюватися. Однак процеси двотижневої генерації засновані на найбільш важливих функціях, що входять у поточний пул, отже, мінливість управляється. Локальний замовник забезпечує підтримку цього стилю ітераційної розробки.

"Метафора" забезпечує глобальне "бачення" проекту. Вона могла б розглядатися як высокоуровневая архітектура, але XP підкреслює бажаність проектування при мінімізації проектної документації. Точніше кажучи, XP пропонує безперервне перепроектування (за допомогою реорганізації), при якому немає потреби в деталізованій проектній документації, а для інженерів супроводу єдиним надійним джерелом інформації є програмний код. Звичайно після написання коду проектна документація викидається. Проектна документація зберігається тільки в тому випадку, коли замовник тимчасово втрачає здатність придумувати нові історії. Тоді систему поміщають в "нафталін" і пишуть керівництво сторінок на п'ять- десять по "нафталіновому" варіанту системи. Використання реорганізації приводить до реалізації найпростішого розв'язку, що задовольняє поточну потребу. Зміни у вимогах змушують відмовлятися від усіх "загальних розв'язків".

Парне програмування - один з найбільш спірних методів у XP, воно впливає на ресурси, що важливо для менеджерів, що вирішують, чи буде проект використовувати XP. Може здатися, що парне програмування подвоює ресурси, але дослідження довели: парне програмування приводить до підвищення якості й зменшенню часу циклу. Для погодженої групи витрати збільшуються на 15%, а час циклу скорочується на 40-50%. Для Інтернет- Середовища збільшення швидкості продажів покриває підвищення витрат. Співробітництво поліпшує процес вирішення проблем, поліпшення якості суттєво знижує витрати супроводу, які перевищують вартість додаткових ресурсів по всьому циклу розробки.

Колективне володіння означає, що будь-який розроблювач може змінювати будь-який фрагмент коду системи в будь-який час. Безперервна інтеграція,

безперервне регресійне тестування й парне програмування XP забезпечують захист від виникаючих при цьому проблем.

"Тестуй, а потім кодуй" - ця фраза виражає акцент XP на тестуванні. Вона відбиває принцип, по якому спочатку планується тестування, а тестові варіанти розробляються паралельно аналізу вимог, хоча традиційний підхід полягає в тестуванні "чорного ящика". Міркування про тестування на початку циклу життя - добре відома практика конструювання ПО (правда, рідко здійснювана практично).

Основним засобом керування XP є метрика, а середовище метрик - "більша візуальна діаграма". Звичайно використовують 3-4 метрики, причому такі, які видимі всій групі. Рекомендованої 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, щораз вибираючи метод, орієнтований на саму важку проблему групи. Звичайно, усі ці методи є "не більш ніж правилами" - група може в будь-який момент поміняти їх (якщо її співробітники досяглися принципової угоди із приводу внесених змін). Захисники XP визнають, що XP виявляє сильний соціальний вплив, і не кожний може прийняти його. Разом з тим, XP - це методологія, що забезпечує переваги тільки при використанні закінченого набору базових методів.

Розглянемо структуру "ідеального" XP- Процесу. Основним структурним елементом процесу є XP- Реалізація, у яку багаторазово вкладається базовий елемент - XP- Ітерація. До складу XP- Реалізації й XP- Ітерації входять три фази - дослідження, блокування, регулювання. Дослідження (exploration) - це пошук нових вимог (історій, завдань), які повинна виконувати система. Блокування (commitment) - вибір для реалізації конкретної підмножини із 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 інших реалізацій - близько двох місяців, кожна ітерація триває приблизно два тижні, а чисельність групи розроблювачів не перевищує 10 людей.Xp- Процес для проекту із сьома реалізаціями, здійснюваний за 15 місяців, показаний на мал. 1.

Процес ініціюється початковою дослідницькою фазою.

Фаза дослідження, з якої починається будь-яка реалізація й ітерація, має клапан "пропуску", на цій фазі ухвалюється розв'язок про доцільність подальшого продовження роботи.

Передбачається, що тривалість першої реалізації становить 3 місяця, тривалість другий - сьомий реалізацій - 2 місяця. 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" (у цей час темп ітерації прискорюється).

Найбільш важка перша реалізація - пройти за три місяці від звичайного старту (скажемо, окремий співробітник не зафіксував ніяких вимог, не певні обмеження) до поставки замовникові системи промислової якості дуже складно.

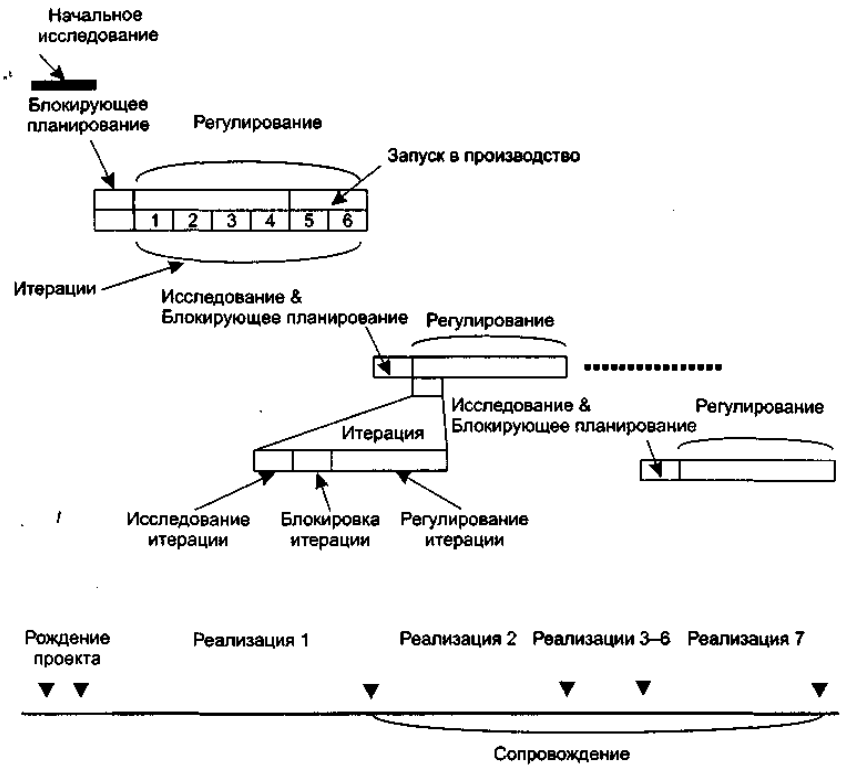


Рисунок 1 - Ідеальний XP- Процес

Моделі якості процесів конструювання

У сучасних умовах, умовах твердої конкуренції, дуже важливо гарантувати висока якість вашого процесу конструювання ПО. Таку гарантію дає сертифікат якості процесу, що підтверджує його відповідність прийнятим міжнародним стандартам. Кожний такий стандарт фіксує свою модель забезпечення якості.

Найбільш авторитетні моделі стандартів ISO 9001:2000, ISO/ IEC 15504 і модель зрілості процесу конструювання ПО (Capability Maturity Model - CMM) Інституту програмної інженерії при американському університеті Карнегі- Меллон.

Модель стандарту ISO 9001:2000 орієнтована на процеси розробки з будь-яких областей людської діяльності. Стандарт ISO/IEC 15504 спеціалізується на процесах програмної розробки й відрізняється більш високим рівнем деталізації. Досить сказати, що обсяг цього стандарту перевищує 500 сторінок. Значна частина ідей ISO/IEC 15504 узята з моделі CMM.

Базовим поняттям моделі CMM уважається зрілість компанії [61], [62]. Незрілої називають компанію, де процес конструювання ПО й прийняті розв'язки залежать тільки від таланта конкретних розроблювачів. Як наслідок, тут висока ймовірність перевищення бюджету або зриву строків закінчення проекту.

Напроти, у зрілій компанії працюють ясні процедури керування проектами й побудови програмних продуктів. У міру необхідності ці процедури уточнюються й розбудовуються. Оцінки тривалості й витрат розробки точні, ґрунтуються на накопиченому досвіді. Крім того, у компанії є й діють корпоративні стандарти на процеси взаємодії із замовником, процеси аналізу, проектування, програмування, тестування й впровадження програмних продуктів. Усе це створює середовище, що забезпечує якісну розробку програмного забезпечення.

Дуже важливо відзначити, що модель CMM орієнтована на побудову системи постійного поліпшення процесів. У ній зафіксовано п'ять рівнів зрілості (мал. 2) і передбачений плавний, поетапний підхід до вдосконалювання процесів - можна поетапно одержувати підтвердження про поліпшення процесів після кожного рівня зрілості.

Рисунок 2 - П'ять рівнів зрілості моделі CMM

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

Для переходу на *повторюваний рівень* (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 й контролюється. Застосовувані засоби планування й керування дають можливість повторення раніше досягнутих успіхів.

Наступний, *певний рівень* (рівень 3) вимагає, щоб усі елементи процесу були визначені, стандартизовані й задокументовані. Основна відмінність від рівня 2 полягає в тому, що елементи процесу рівня 3 плануються й управляються на основі єдиного стандарту компанії. Якість розроблювального ПО вже не залежить від здатностей окремих особистостей.

З переходом на *керований рівень* (рівень 4) у компанії ухвалюються кількісні показники якості як програмних продуктів, так і процесу. Це забезпечує більш точне планування проекту й контроль якості його результатів. Основна

відмінність від рівня 3 полягає в більш об'єктивній, кількісній оцінці продукту й процесу.

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

Кожний рівень CMM характеризується областю ключових процесів (ОКП), причому вважається, що кожний наступний рівень містить у собі всі характеристики попередніх рівнів. Інакше кажучи, для 3- го рівня зрілості розглядаються ОКП 3- го рівня, ОКП 2- го рівня й ОКП 1- го рівня. Область ключових процесів утворюють процеси, які при спільному виконанні приводять до досягнення певного набору цілей. Наприклад, ОКП 5- го рівня утворюють процеси:

- запобігання дефектів;
- керування змінами технології;
- керування змінами процесу.

Якщо всі мети ОКП досягнуті, компанії привласнюється сертифікат даного рівня зрілості. Якщо хоча б одна мета не досягнута, то компанія не може відповідати даному рівню CMM.

Тема №3: Процес керівництва проєктом

(2 години)

Мета: Виявити значення процесу керівництва проєктом, розглянути основні етапи процесу керівництва проєктом.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:

a. Повідомлення теми та мети заняття;

b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Керівництво програмним проєктом:
    a. Початок проєкту;
    b. Виміри міри і метрики;
    c. Процес оцінки;
    d. Аналіз ризику;
    e. Планування;
    f. Трасування і контроль.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:

1. Ознайомитись з теоретичним матеріалом теми 3.
2. Вивчити основні поняття лекції.
3. Відповісти на контрольні питання.

Контрольні питання:

1. Коли відбувається керівництво програмним проектом?
2. Що необхідно зробити перед плануванням проекту?
3. Що таке міра і як вона визначається?
4. Що таке метрика?
5. Для чого проводяться виміри процесу, а для чого виміри продукту?
6. Що оцінюється в процесі оцінки?
7. В процесі аналізу ризиків яке рішення ухвалюється?
8. Що визначається в процесі планування?
9. Що використовується при відставанні в рішенні задачі?

Керівництво програмним проєктом - перший шар процесу конструювання ПЗ. Термін "шар" підкреслює, що керівництво визначає сутність процесу розробки від його початку до кінця. Принцип керівництва ілюструє мал. 1.

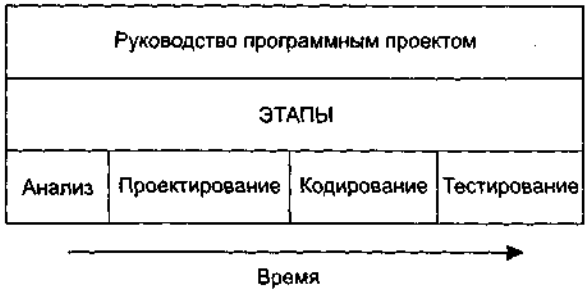


Рисунок 1 – Керівництво в процесі конструювання ПЗ

На цьому малюнку прямокутник позначає процес конструювання, у ньому виділені етапи, а вгорі, над кожним з етапів, розміщений шар діяльності "керівництво програмним проектом".

Для проведення успішного проекту потрібно зрозуміти обсяг майбутніх робіт, можливий ризик, необхідні ресурси, що стоять завдання, що прокладаються віхи, необхідні зусилля (вартість), план робіт, якому бажане випливати. Керівництво програмним проектом забезпечує таке розуміння. Воно починається перед технічною роботою, триває в міру розвитку ПЗ від ідеї до реальності й досягає найвищого рівня до кінця робіт.

Початок проєкту

Перед плануванням проєкту необхідно:

- установити цілі й проблемну область проекту;
- обговорити альтернативні розв'язки;
- виявити технічні й управлінські обмеження.

Виміри, міри й метрики

Виміру допомагають зрозуміти як процес розробки продукту, так і сам продукт. Виміри процесу проводяться з метою його поліпшення, виміри продукту - для підвищення його якості. У результаті вимірів визначається *міра* - кількісна характеристика якої-небудь властивості об'єкта. Шляхом безпосередніх вимірів можуть визначатися тільки опорні властивості об'єкта. Усі інші властивості оцінюються в результаті обчислення тих або інших функцій від значень опорних характеристик. Обчислення цих функцій проводяться по формулах, що дають числові значення й називаються *метриками*.

В IEEE Standard Glossary of Software Engineering Terms метрика визначена як міра ступеня володіння властивістю, що має числове значення. У програмній інженерії поняття міра і метрика дуже часто розглядають як синоніми.

Процес оцінки

При плануванні програмного проекту треба оцінити людські ресурси (у людино-місяцях), тривалість (у календарних датах), вартість (у тисячах доларів). Звичайно виходять із минулого досвіду. Якщо новий проект по розміру й функціям схожий на попередній проект, цілком імовірно, що будуть потрібні такі ж ресурси, час і гроші.

Аналіз ризику

На цій стадії досліджується область невизначеності, що є в наявності перед створенням програмного продукту. Аналізується її вплив на проект. Чи немає схованих від уваги важких технічних проблем? Не чи стануть зміни, що виявилися в ході проектування, причиною неприпустимого відставання по строках? У результаті ухвалюється рішення - виконувати проект чи ні.

Планування

Визначається набір проектних завдань. Установлюються зв'язки між завданнями, оцінюється складність кожного завдання. Визначаються людські й інші ресурси. Створюється сітковий графік завдань, проводиться його тимчасова розмітка.

Трасування й контроль

Кожне завдання, позначена в плані, відслідковується керівником проекту. При відставанні в розв'язку завдання застосовуються утиліти повторного планування. За допомогою утиліт визначається вплив цього відставання на проміжну віху й загальний час конструювання. Під віхою розуміється тимчасова мітка, до якої прив'язано підведення проміжних підсумків.

У результаті повторного планування:

- можуть бути перерозподілені ресурси;
- можуть бути реорганізовані завдання;
- можуть бути переглянуті вихідні зобов'язання.

Тема №4: Планування проєктних задач

(2 години)

Мета: Визначити поняття структури розподілу робіт. Ознайомитись з основними поняттями розмірно-орієнтованих та функціонально-орієнтованих метрик.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Планування проектних задач;
2. Розмірно-орієнтовані метрики;
3. Функціонально-орієнтовані метрики.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми 4.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Що таке WBS?
2. З якою метою проводиться системний аналіз?
3. Які можливості дає аналіз вимог?
4. Які оцінки використовують при розрахунку границь часу виконання рішень?
5. Що таке розмірно-орієнтована метрика?

6. На чому ґрунтуються розмірно-орієнтовані метрики?
7. Що таке функціонально-орієнтовані метрики?
8. Які характеристики використовують функціонально-орієнтовані метрики?

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. 1.

Першими виконуваними завданнями є системний аналіз і аналіз вимог. Вони закладають фундамент для наступних паралельних завдань.

Системний аналіз проводиться з метою:

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

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

Аналіз вимог дає можливість:

1) визначити функції й 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 вистави інформації й функцій системи, які можна використовувати в ході проектування.

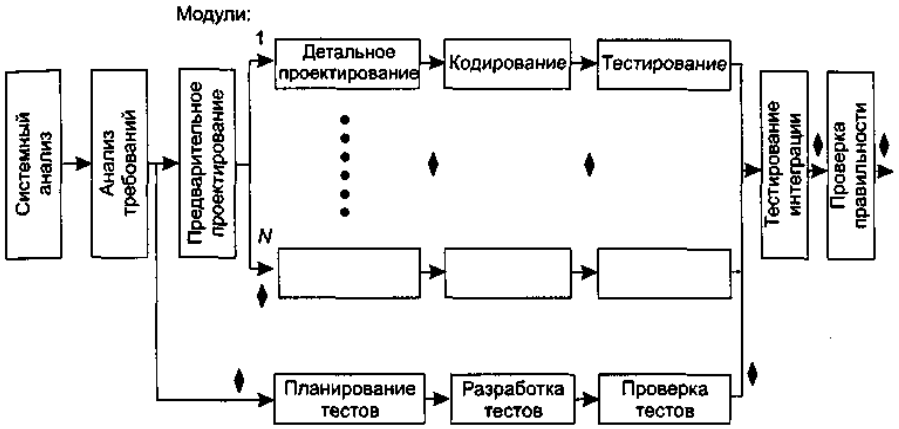


Рисунок 1 – Типова структура розподілу проектних робіт

Результати аналізу зводяться в *специфікацію вимог* до програмного продукту.

Як видне з типової структури, завдання по проектуванню й плануванню тестів можуть бути розпаралеленні. Завдяки модульній природі ПЗ для кожного модуля можна передбачити паралельний шлях для детального (процедурного) проектування, кодування й тестування. Після одержання всіх модулів ПЗ вирішується завдання тестування інтеграції - об'єднання елементів у єдине ціле. Далі проводиться тестування правильності, яке забезпечує перевірку відповідності ПЗ вимогах замовника.

Ромбиками на мал. 1 позначені *віхи* - процедури контролю проміжних результатів. Дуже важливо, щоб віхи були розставлені через регулярні інтервали (уздовж усього процесу розробки ПЗ). Це дасть керівникові можливість регулярно одержувати інформацію про поточний стан справ. Віхи поширюються й на документацію як на один з результатів успішного розв'язку завдання.

Паралельність дій підвищує вимоги до планування. Тому що паралельні завдання виконуються асинхронно, планувальник повинен визначити міжзадачні залежності. Це гарантує "безперервність руху до об'єднання". Крім того, керівник проекту повинен знати завдання, що лежать на критичному шляху. Для того щоб увесь проект був виконаний у строк, необхідно виконувати в строк усі критичні завдання.

Основний важіль у плануючих методах - обчислення границь часу виконання завдання.

Звичайно використовують наступні оцінки:

1. Ранній час початку розв'язку завдання $T_{\min}^{in}$ (за умови, що всі попередні завдання вирішені в найкоротший час).

2. Пізніше час початку розв'язку завдання $T_{\max}^{in}$ (ще не викликає загальну затримку проекту).

3. Ранній час кінця розв'язку завдання $T_{\min}^{out}$.

$$T_{\min}^{out} + T_{\min}^{in} + T_{реш}$$

4. Пізніше час кінця розв'язку завдання $T_{\max}^{out}$.

$$T_{\max}^{out} + T_{\max}^{in} + T_{реш}$$

5. Загальний резерв - кількість надлишків і втрат планування завдань у часі, що не приводять до збільшення тривалості критичного шляху $T_{к.п}$.

Усі ці значення дозволяють керівникові (планувальникові) кількісно оцінити успіх у плануванні, виконанні завдань.

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

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

Розмірно-Орієнтовані метрики

Розмірно-Орієнтовані метрики прямо вимірюють програмний продукт і процес його розробки. Ґрунтуються розмірно-орієнтовані метрики на Loc- Оцінках (Lines Of Code). Loc- Оцінка - це кількість рядків у програмному продукті.

Вихідні дані для розрахунків цих метрик зводяться в таблицю (табл. 1).

Таблиця 1. Вихідні дані для розрахунків Loc- Метрик

Проект	**Витрати люд.-міс**	**Вартість тыс. $**	**KLOC тис. LOC**	**Прогр. док- ти, сторінок**	***Помилки***	**Люди**
aaa01	24	168	12,1	365	29	3
bbb02	62	440	27,2	1224	86	5
ccc03	43	314	20,2	1050	64	6

Таблиця містить дані про проекти за останні кілька років. Наприклад, запис про проект ааа01 показує: 12 100 рядків програми були розроблені за 24 людино-місяця й коштували $168 000. Крім того, по проекту ааа01 було розроблено 365 сторінок документації, а протягом першого року експлуатації було зареєстровано 29 помилок. Розробляли проект ааа01 три людини.

На основі таблиці обчислюються розмірно-орієнтовані метрики продуктивності і якості (для кожного проекту):

$$\text{Производительность} = \frac{\text{Длина}}{\text{Затраты}}\left[\frac{\text{тыс.LOC}}{\text{чел.-мес}}\right];$$

$$\text{Качество} = \frac{\text{Ошибки}}{\text{Длина}}\left[\frac{\text{Единиц}}{\text{тыс.LOC}}\right];$$

$$\text{Удельная стоимость} = \frac{\text{Стоимость}}{\text{Длина}}\left[\frac{\text{Тыс\$}}{\text{LOC}}\right];$$

$$\text{Документированность} = \frac{\text{Страниц Документа}}{\text{Длина}}\left[\frac{\text{Страниц}}{\text{тыс.LOC}}\right].$$

Гідності розмірно-орієнтованих метрик:

1) широко поширені;

2) прості й легко обчислюються.

Недоліки розмірно-орієнтованих метрик:

1) залежні від мови програмування;

2) вимагають вихідних даних, які важко одержати на початковій стадії проекту;

3) не пристосовані до не процедурних мов програмування.

Функціонально-орієнтовані метрики

Функціонально-орієнтовані метрики побічно вимірюють програмний продукт і процес його розробки. Замість підрахунку Loc-Оцінки при цьому розглядається не розмір, а функціональність або корисність продукту.

Використовується 5 інформаційних характеристик.

1. *Кількість зовнішніх вводів.* Підраховуються всі введення користувача, по яких надходять різні прикладні дані. Уведення повинні бути відділені від запитів, які підраховуються окремо.

2. *Кількість зовнішніх виводів*. Підраховуються всі висновки, по яких до користувача надходять результати, обчислені програмним додатком. У цьому контексті висновки означають звіти, екрани, роздруківки, повідомлення про помилки. Індивідуальні одиниці даних усередині звіту окремо не підраховуються.

3. *Кількість зовнішніх запитів*. Під запитом розуміється діалогове введення, яке приводить до негайної програмної відповіді у формі діалогового висновку. При цьому діалогове введення в додатку не зберігається, а діалоговий висновок не вимагає виконання обчислень. Підраховуються всі запити - кожний ураховується окремо.

4. *Кількість внутрішніх логічних файлів*. Підраховуються всі логічні файли (тобто логічні групи даних, які можуть бути частиною бази даних або окремим файлом).

5. *Кількість зовнішніх інтерфейсних файлів*. Підраховуються всі логічні файли з інших додатків, на які посилається даний додаток.

Вводи, виводи й запити відносять до категорії транзакція. *Транзакція* - це елементарний процес, що різниться користувачем, що й переміщає дані між зовнішнім середовищем і програмним додатком. У своїй роботі транзакції використовують внутрішні й зовнішні файли. Прийняті наступні визначення.

Зовнішній ввід - елементарний процес, що переміщає дані із зовнішнього середовища в додаток. Дані можуть надходити з екрана введення або з іншого додатка. Дані можуть використовуватися для відновлення внутрішніх логічних файлів. Дані можуть містити як керуючу, так і ділову інформацію. Керуючі дані не повинні модифікувати внутрішній логічний файл.

Зовнішній вивід - елементарний процес, що переміщає дані, обчислені в додатку, у зовнішнє середовище. Крім того, у цьому процесі можуть обновлятися внутрішні логічні файли. Дані створюють звіти або вихідні файли, що посилають іншим додаткам. Звіти й файли створюються на основі внутрішніх логічних файлів і зовнішніх інтерфейсних файлів. Додатково цей процес може використовувати дані, що вводяться, їх утворюють критерії пошуку й параметри, не підтримувані внутрішніми логічними файлами. дані, що вводяться, надходять ззовні, але носять тимчасовий характер і не зберігаються у внутрішньому логічному файлі.

Зовнішній запит - елементарний процес, що працює як з, що вводяться, так і з виведеними даними. Його результат - дані, що вертаються із внутрішніх логічних файлів і зовнішніх інтерфейсних файлів. Вхідна частина процесу не модифікує внутрішні логічні файли, а вихідна частина не несе даних, що обчислюються додатком (у цьому й полягає відмінність запиту від висновку).

Внутрішній логічний файл - розпізнавана користувачем група логічно зв'язаних даних, яка розміщена усередині додатка й обслуговується через зовнішні введення.

Зовнішній інтерфейсний файл - розпізнавана користувачем група логічно зв'язаних даних, яка розміщена усередині іншого додатка й підтримується ім. Зовнішній файл даного додатка є внутрішнім логічним файлом в іншому додатку.

Кожної з виявлених характеристик ставиться у відповідність складність. Для цього характеристиці призначається низький, середній або високий ранг, а потім формується числова оцінка рангу.

Для транзакцій ранжирування засноване на кількості посилань на файли й кількості типів елементів даних. Для файлів ранжирування засноване на кількості типів елементів-записів і типів елементів даних, що входять у файл.

Тип елемента-запису - підгрупа елементів даних, розпізнавана користувачем у межах файлу.

Тип елемента даних - унікальне не рекурсивне (неповторювальне) поле, розпізнаване користувачем.

Тема №5: Виконання оцінки проєкта на основі LOC- та FP-метрик

(2 години)

Мета: Розглянути процес виконання оцінки проекту на основі LOC- і FP-Метрик.

План:

1. Виконання оцінки в ході керівництва проєктом:
2. Виконання оцінки проекту на основі LOC- і FP- Метрик;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Які размерно- орієнтовані метрики ви знаєте?
2. Для чого використовують размерно- орієнтовані метрики?
3. Визначите гідності й недоліки размерно- орієнтованих метрик.
4. Чи можна перейти від FP- Оцінок до LOC- Оцінкам?
5. Охарактеризуйте кроки оцінки проекту на основі LOC- і FP- Метрик.
6. Чим відрізняється найбільш точний підхід від найменш точного?

Виконання оцінки в ході керівництва проектом

Процес керівництва програмним проектом починається з безлічі дій, поєднуваних загальною назвою планування проекту. Перше із цих дій - виконання оцінки. Воно закладає фундамент для інших дій по плануванню проекту. При оцінці проекту надзвичайно висока ціна помилок. Дуже важливо провести оцінку з мінімальним ризиком.

Виконання оцінки проекту на основі LOC- і FP- Метрик

Ціль цієї діяльності - сформувати попередні оцінки, які дозволять:

- пред'явити замовникові коректні вимоги за вартістю й витратам на розробку програмного продукту;
- скласти план програмного проекту.

При виконанні оцінки можливі два варіанти використання LOC- і Fp- Даних:

- у якості оцінних змінн, що визначають розмір кожного елемента продукту;
- у якості метрик, зібраних за минулі проекти й вхідних у метричний базис фірми.

Обговоримо кроки процесу оцінки.

 Крок 1. Область призначення проектованого продукту розбивається на ряд функцій, кожну з яких можна оцінити індивідуально:

$$f1, f2, \ldots, fn.$$

 Крок 2. Для кожної функції fi, планувальник формує кращу $LOC_{лучші}$ ($FP_{лучші)}$, гіршу $LOC_{худші}$ ($FP_{худші}$) і ймовірну оцінку $LOC_{вероятні}$ ($FP_{вероятні}$). Використовуються досвідчені дані (з метричного базису) або інтуїція. Діапазон значення оцінок відповідає ступеню передбаченої невизначеності.

 Крок 3. Для кожної функції/ відповідно до -розподілом обчислюється очікуване значення LOC- (або FP-) оцінки:

$$LOC_{ожі} = (LOC_{лучші} + LOC_{худші} + 4x\ LOC_{вероятні}) / 6.$$

 Крок 4. Визначається значення LOC- або FP- Продуктивності розробки функції.

Використовується один із трьох підходів:

1) для всіх функцій ухвалюється та сама метрика середньої продуктивності ПРОИЗВСР, узята з метричного базису;

2) для і- і функції на основі метрики середньої продуктивності обчислюється величина, що настроюється, продуктивності:

$$ПРОИЗВ_i = ПРОИЗВ_{ср} x (LOC_{ср} / LOC_{ожі}),$$

де $LOC_{ср}$ - середня LOC- Оцінка, узята з метричного базису (відповідає середній продуктивності);

3) для і- і функції величина, що настроюється, продуктивності обчислюється по аналогові, узятому з метричного базису:

$$ПРОИЗВ_i = ПРОИЗВ_{ані} x (LOC_{ані} / LOC_{ожі}).$$

Перший підхід забезпечує мінімальну точність (при максимальній простоті обчислень), а третій підхід - максимальну точність (при максимальній складності обчислень).

 Крок 5. Обчислюється загальна оцінка витрат на проект: для першого підходу

$$ЗАТРАТЫ = (\sum_{i=1}^{n} LOC_{ожі}) / ПРОИЗВ_{ср} [чел.-мес];$$

для другого й третього підходів

$$\text{ЗАТРАТЫ} = \sum_{i=1}^{n} (\text{LOC}_{\text{ожі}} / \text{ПРОИЗВ}_i)[\text{чел.-мес}] .$$

 Крок 6. Обчислюється загальна оцінка вартості проекту: для першого й другого підходів

$$\text{СТОИМОСТЬ} = (\sum_{i=1}^{n} \text{LOC}_{\text{ожі}}) \times \text{УД\_СТОИМОСТЬ}_{\text{ср}} ,$$

де УД_СТОИМОСТЬСР - метрика середньої вартості одному рядка, узята з метричного базису.

для третього підходу

$$\text{СТОИМОСТЬ} = \sum_{i=1}^{n} (\text{LOC}_{\text{ожі}} \times \text{УД\_СТОИМОСТЬ}_{\text{ані}}),$$

де УД_Стоимость$_{\text{ані}}$ - метрика вартості одному рядка аналога, узята з метричного базису. Приклад застосування даного процесу оцінки приведемо нижче.

Тема №6: Попередня оцінка програмного продукту

(2 години)

Мета: Розглянути процес попередньої оцінки програмного проекту на основі конкретного прикладу.

План:

1. Попередня оцінка програмного проекту;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Виконати завдання.

Завдання:

1. Зробити попередню оцінку програмного продукту для будь-якої організації, з роботою якої Ви знайомі.

Попередня оцінка програмного проекту

У якості ілюстрації застосування методики оцінки, викладеної в розділі "Виконання оцінки проекту на основі LOC- і FP- Метрик", розглянемо конкретний приклад. Припустимо, що зробило замовлення від концерну "СУПЕРАВТО". Необхідно створити ПЗ для робочої станції дизайнера автомобіля (РДА). Замовник визначив проблемну область проекту у своїй специфікації:

- ПЗ РДА повинне формувати 2- і 3- мірні зображення для дизайнера;
- дизайнер повинен вести діалог із РДА й управляти їм за допомогою стандартизованого графічного користувацького інтерфейсу;
- геометричні дані й прикладні дані повинні втримуватися в базі даних РДА;
- модулі проектного аналізу робочої станції повинні формувати дані для широкого класу дисплеїв SVGA;
- ПП РДА повинне управляти й звістки діалог з наступними периферійними обладнаннями: миша, дигитайзер (графічний планшет для ручного введення), плоттер (графобудівник), сканер, струминний і лазерний принтери.

Насамперед треба деталізувати проблемну область. Слід виділити базові функції ПЗ й окреслити кількісні границі. Очевидно, потрібно визначити, що таке "стандартизований графічний користувацький інтерфейс", якими повинні бути розмір і інші характеристики бази даних РДА і т.д..

Будемо вважати, що ця робота пророблена й що ідентифіковані наступні основні функції ПЗ:

1. Засобу керування користувацьким інтерфейсом СУПИ.
2. Аналіз двомірної графіки А2Г.
3. Аналіз тривимірної графіки А3Г.
4. Керування базою даних УБД.
5. Засобу комп'ютерної дисплейної графіки КДГ.
6. Керування периферією УП.
7. Модулі проектного аналізу МПА.

Тепер потрібно оцінити кожну з функцій кількісно, за допомогою LOC-Оцінки. По кожній функції експерти надають краще, гірше і ймовірне значення. Очікувану LOC- Оцінку реалізації функції визначаємо по формулі

$LOC_{ожі} = (LOC_{лучші} + LOC_{худші} + 4 \times LOC_{вероятні})/6$,

результати розрахунків затягаємо в табл. 1.

Таблиця 1 - Початкова таблиця оцінки проекту

Функция	**Лучш. [LOC]**	**Вероят. [LOC]**	**Худш. [LOC]**	**Ожид. [LOC]**	**Уд. стоимость [$/LOC]**	**Стоимо-сть[$]**	**Произв. [LOC/ [чел-мес]**	**Затраты [чел-мес]**
СУПИ	1800	2400	2650	2340				
А2Г	4100	5200	7400	5380				
АЗГ	4600	6900	8600	6800				
УБД	2950	3400	3600	3350				
КДГ	4050	4900	6200	4950				
УП	2000	2100	2450	2140				
МПА	6600	8500	9800	8400				
Итого				33360				

Для визначення питомої вартості й продуктивності звернемося в архів фірми, де зберігаються дані метричного базису, зібрані по вже виконаних проектах. Припустимо, що з метричного базису витягнуті дані по функціях- аналогам, представлені в табл. 2.

Видне, що найбільшу питому вартість має рядок функції керування периферією (потрібні специфічні й конкретні знання по різноманітних периферійних обладнаннях), найменшу питому вартість - рядок функції керування користувацьким інтерфейсом (застосовуються широко відомі розв'язки).

Таблиця 2 - Дані з метричного базису фірми

Функция	**$LOC_{ані}$**	**$УД\_СТОИМОСТЬ_{ані}$[$ / LOC]**	**$ПРОИЗВ_{ані}$[LOC/чел-мес]**
СУПИ	585	14	1260
А_Г	3000	20	440
УБД	1117	18	720
КДГ	2475	22	400
УП	214	28	1400
МПА	1400	18	1800

Уважається, що питома вартість рядка є константою й не змінюється від реалізації до реалізації. Отже, вартість розробки кожної функції розраховуємо по формулі

$$Стоимость_i = LOC_{ожі} \text{ x } УД\_Стоимость_{ані}.$$

Для обчислення продуктивності розробки кожної функції виберемо найточніший підхід - підхід продуктивності, що настроюється:

$$ПРОИЗВ_i = Произв_{ані} \text{ x } (LOC_{ані} / LOC_{ожі}).$$

Відповідно, витрати на розробку кожної функції будемо визначати по вираженню

$$ВИТРАТИ_i = (LOC_{ожі} / ПРОИЗВ_i)[чол./міс].$$

Тепер ми маємо всі необхідні дані для завершення розрахунків. Заповнимо до кінця таблицю оцінки нашого проекту (табл. 3).

Таблиця 3 - Кінцева таблиця оцінки проекту

Функция	**Лучш.**	**Вероят.**	**Худш.**	**Ожид. [LOC]**	**Уд. стоимость [S/LOC]**	***Стоимость* [$]**	**Произв. [LOC/ чел.-мес]**	**Затраты [чел-мес]**
СУПИ	1800	2400	2650	2340	14	32760	315	7,4
А2Г	4100	5200	7400	5380	20	107600	245	21,9
АЗГ	4600	6900	8600	6800	20	136000	194	35,0
УБД	2950	3400	3600	3350	18	60300	240	13,9
КДГ	4050	4900	6200	4950	22	108900	200	24,7
УП	2000	2100	2450	2140	28	59920	140	15,2
МПА	6600	8500	9800	8400	18	151200	300	28,0
Итого				33360		656680		146

Враховуючи важливість отриманих результатів, перевіримо розрахунки за допомогою Fp- Покажчиків. На даному етапі оцінювання розумно допустити, що всі інформаційні характеристики мають середній рівень складності. У цьому випадку результати експертної оцінки ухвалюють вид, представлений у табл. 4 та 5.

Таблиця 4 - Оцінка інформаційних характеристик проекту

Характеристика	**Лучш.**	**Вероят.**	**Худш.**	**Ожид.**	**Сложность**	**Количество**
Вводы	20	24	30	24	x 4	96
Выводы	12	15	22	16	x 5	80
Запросы	16	22	28	22	x 4	88
Логические файлы	4	4	5	4	x 10	40
Интерфейсные файлы	2	2	3	2	x 7	14
Общее количество						318

Таблиця 5 - Оцінка системних параметрів проекту

Коэффициент регулировки сложности		**Оценка**
F_1	Передачи данных	2
F_2	Распределенная обработка данных	0
F_3	Производительность	4
F_4	Распространенность используемой конфигурации	3
F_5	Скорость транзакций	4
F_6	Оперативный ввод данных	5
F_7	Эффективность работы конечного пользователя	5
F_8	Оперативное обновление	3
F_9	Сложность обработки	5
F_{10}	Повторная используемость	4
F_{11}	Легкость инсталляции	3
F_{12}	Легкость эксплуатации	4
F_{13}	Разнообразные условия размещения	5
F_{14}	Простота изменений	5

Таким чином, одержуємо:

FP = Загальна кількість x (0,65+ 0,01 x) = 318 x 1,17 = 372.

Використовуючи значення продуктивності, узяте в метричному базисі фірми,

Продуктивність = 2,55 [FP / чіл.-мес],

обчислюємо значення витрат і вартості:

Витрати = FP / Продуктивність = 145,9 [чіл.-мес],

Вартість = Витрати x $4500 = $656500.

Отже, результати перевірки показали гарну вірогідність результатів. Але ми не будемо зупинятися на досягнутому й організуємо ще одну перевірку, за допомогою моделі COCOMO II.

Приймемо, що всі масштабні фактори й фактори витрат мають номінальні значення. У силу цього показник ступеня B = 1,16, а множник виправлення Мр= 1. Крім того, будемо вважати, що автоматична генерація коду й повторне використання компонентів не передбачаються. Отже, ми має право застосувати формулу

ВИТРАТИ = A x РАЗМЕРВ[чіл.-мес]

і одержуємо:

ВИТРАТИ = 2,5(33,3)1,16 =145,87 [чіл.-мес].

Відповідно, номінальна тривалість проекту рівна

Тривалість = [3,0 x (ВИТРАТИ)(0,33+0,2(B-1,01))]=3(145,87)0,36 = 18[мес].

Підведемо підсумки. Виконана попередня оцінка програмного проекту. Для мінімізації ризику оцінювання використано три методики, що довели коректність отриманих результатів.

Тема №7: Аналіз чутливості програмного проєкту

(2 години)

Мета: Розглянути можливості моделі COCOMO II в завданнях аналізу чутливості програмного проекту до зміни умов розробки.

План:

1. Аналіз чутливості програмного проекту;
2. Сценарій зниження зарплати;
3. Сценарій нарощування пам'яті;
4. Сценарій використання нового мікропроцесора;
5. Сценарій зменшення засобів на завершення проекту.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;

2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. У чому полягає призначення моделі етапу пост-архітектури COCOMO II?
2. Чим відрізняється основне рівняння моделі етапу пост-архітектури від аналогічного рівняння моделі раннього етапу проектування?
3. Що таке фактори витрат моделі етапу пост-архітектури і як вони обчислюються?
4. Як визначається тривалість розробки в моделі COCOMO II?
5. Що таке аналіз чутливості програмного проекту?
6. Як застосувати модель COCOMO II до аналізу чутливості?

Аналіз чутливості програмного проекту

COCOMO II - авторитетна й багатопланова модель, що дозволяє вирішувати найрізноманітніші завдання керування програмним проектом.

Розглянемо можливості цієї моделі в завданнях аналізу чутливості - чутливості програмного проекту до зміни умов розробки.

Будемо вважати, що корпорація "Сверхмобильныесвязи" замовила розробку ПЗ для вбудованої космічної системи обробки повідомлень. Очікуваний розмір ПЗ - 10 KLOC, використовується серійний мікропроцесор. Приймемо, що масштабні фактори мають номінальні значення (показник ступеня B = 1,16) і що автоматична генерація коду не передбачається. До проведення розробки залучаються головний аналітик і головний програміст високої кваліфікації, тому середня зарплата в команді складе $ 6000 на місяць. Команда має річний досвід роботи із цією проблемною областю й півроку працює з потрібною апаратною платформою.

У термінах COCOMO II проблемну область (область застосування продукту) класифікують як "операції із приладами" з наступним описом: вбудована система для високошвидкісного мультиприоритетного обслуговування вилучених ліній зв'язку, що забезпечує можливості діагностики.

Оцінку пост-архітектурних факторів витрат для проекту зведемо в табл. 1.

З таблиці випливає, що збільшення витрат в 1,3 рази через дуже високу складність продукту врівноважується їхнім зменшенням внаслідок високої кваліфікації аналітика й програміста, а також активного використання програмних утиліт.

Таблиця 1 - Оцінка пост-архітектурних факторів витрат

Фактор	**Описание**	**Оценка**	**Множитель**
RELY	Требуемая надежность ПО	Номинал.	1
DATA	Размер базы данных — 20 Кбайт	Низкая	0,93
CPLX	Сложность продукта	Очень высок.	1,3
RUSE	Требуемая повторная используемость	Номинал.	1
DOCU	Документирование жизненного цикла	Номинал.	1
TIME	Ограничения времени выполнения (70%)	Высокая	1,11
STOR	Ограничения оперативной памяти (45 из 64 Кбайт, 70%)	Высокая	1,06
PVOL	Изменчивость платформы (каждые 6 месяцев)	Номинал.	1
ACAP	Возможности аналитика (75%)	Высокая	0,83
PCAP	Возможности программиста (75%)	Высокая	0,87
AEXP	Опыт работы с приложением (1 год)	Номинал.	1
PEXP	Опыт работы с платформой (6 месяцев)	Низкая	1,12
LTEX	Опыт работы с языком и утилитами (1 год)	Номинал.	1
PCON	Непрерывность персонала (1 2% в год)	Номинал.	1
TOOL	Активное использование программных утилит	Высокая	0,86
SITE	Мультисетевая разработка (телефоны)	Низкая	1,1
SCED	Требуемый график разработки	Номинал.	1
Множитель поправки M_p			1,088

Розрахуємо витрати й вартість проекту:

ВИТРАТИ = Ахразмерbхмр=2,5(10)1,16х1,088=36х1,088= 39[чіл.-мес],

ВАРТІСТЬ = ВИТРАТИ х $6000 = $234 000.

Такі стартові умови програмного проекту. А тепер обговоримо кілька сценаріїв можливого розвитку подій.

Сценарій зниження зарплати

Покладемо, що замовник розв'язав заощадити на зарплаті розроблювачів. Важіль - зниження кваліфікації аналітика й програміста. Відповідно, зарплата співробітників знижується до $5000. Оцінки їх можливостей стають номінальними, а відповідні множники витрат ухвалюють одиничні значення:

$EM_{ACAP}=EM_{PCAP}=1$.

Наслідком такого розв'язку є зростання множника виправлення Мр= 1,507, а також витрат і вартості:

ВИТРАТИ = 36х 1,507 = 54 [чіл.-мес],

ВАРТІСТЬ = ВИТРАТИ х$5000 = $270 000,

Програш_ в_вартості = $36 000.

Сценарій нарощування пам'яті

Покладемо, що розроблювач запропонував наростити пам'ять - купити за $1000 чип ОЗУ ємністю 96 Кбайт (замість 64 Кбайт). Це міняє обмеження пам'яті (використовується не 70%, а 47%), після чого фактор STOR знижується до номінального:

EM_{STOR}=1-> M_p =1,026,

ВИТРАТИ = 36х1,026 = 37 [чіл.-мес],

ВАРТІСТЬ = ВИТРАТИ х $6000 = $222000,

Виграш_ в_вартості = $ 12 000.

Сценарій використання нового мікропроцесора

Покладемо, що замовник запропонував використовувати новий, більш дешевий МП (дешевше на $1000). До чого це приведе? Досвід роботи з його мовою й утилітами знижується від номінального до дуже низького й EM_{LTEX} = 1,22, а розроблені для нього утиліти (компілятори, асемблери й отладчики) примітивні й ненадійні (у результаті фактор TOOL знижується від високого до дуже низького й EM_{TOOL}= 1,24):

M_p = (1,088 / 0,86) х 1,22 х 1,24 = 1,914,

ВИТРАТИ = 36х1,914 = 69[чіл.-мес],

ВАРТІСТЬ = ВИТРАТИ х $6000 = $414000,

Програш_в_вартості = $180000.

Сценарій зменшення засобів на завершення проекту

Покладемо, що до розробки прийнятий сценарій з нарощуванням пам'яті:

ВИТРАТИ = 36 х 1,026 = 37 [чіл.-мес],

ВАРТІСТЬ = ВИТРАТИ х $6000 = $222000.

Крім того, припустимо, що завершився етап аналізу вимог, на який було витрачено $22 000 (10% від бюджету). Після цього на завершення проекту залишилося $200 000.

Допустимо, що в цей момент "підступний" замовник повідомляє про відсутність у нього достатніх коштів і про надання на завершення розробки тільки $170 000 (15%-ное зменшення оплати).

Для розв'язку цієї проблеми треба встановити можливі зміни факторів витрат, що дозволяють зменшити оцінку витрат на 15%.

Перший розв'язок: зменшення розміру продукту (за рахунок виключення деяких функцій). Нам треба визначити розмір мінімізованого продукту. Будемо виходити з того, що витрати повинні поменшатися з 37 до 31,45 чіл.-мес. Розв'яжемо рівняння:

2,5 (НовыйРазмер)1,16= 31,45 [чіл.-мес].

Очевидно, що

(НовыйРазмер)1,16 = 12,58,

(НовыйРазмер)1,16 = 12,581/1,16 = 8,872 [KLOC].

Інші розв'язки:

- зменшити необхідну надійність із номінальної до низкою. Це скорочує вартість проекту на 12% (EM_{RELY} змінюється з 1 до 0,88). Такий розв'язок приведе до збільшення витрат і труднощів при застосуванні й супроводі;
- підвищити вимоги до кваліфікації аналітиків і програмістів (з високих до дуже високих). При цьому вартість проекту зменшується на 15-19%. Завдяки програмістові вартість може поменшатися на (1 - 0,74/0,87) х 100% = 15%. Завдяки аналітикові вартість може понизитися на (1 - 0,67/0,83) х 100% = 19%. Основні труднощі - пошук фахівців такого класу (готових працювати за ті ж гроші);
- підвищити вимоги до досвіду роботи з додатком (з номінальних до дуже високих) або вимоги до досвіду роботи із платформою (з низьких до високих). Підвищення досвіду роботи з додатком скорочує вартість проекту на (1-0,81) х 100% = 19%; підвищення досвіду роботи із платформою скорочує вартість проекту на (1 - 0,88/1,12) х 100% = 21,4%. Основні труднощі - пошук експертів (фахівців такого класу);

- підвищити рівень мультисетевой розробки з низького до високого. При цьому вартість проекту зменшується на (1 - 0,92/1,1) х 100% = 16,4%;
- послабити вимоги до режиму роботи в реальному часі. Припустимо, що 70%-ное обмеження за часом виконання пов'язане з бажанням замовника забезпечити обробку одного повідомлення за 2 мс. Якщо ж замовник погодиться на збільшення середнього часу обробки з 2 до 3 мс, те обмеження за часом стане рівно (2 мс/3 мс) х 70% = 47%, у результаті чого фактор TIME поменшається з високого до номінального, що приведе до економії витрат на (1- 1/1,11) х 100%= 10%;
- облік інших факторів витрат не має змісту. Деякі фактори (розмір бази даних, обмеження оперативної пам'яті, необхідний графік розробки) уже мають мінімальні значення, для інших важко очікувати швидкого поліпшення (використання програмних утиліт, досвід роботи з мовою й утилітами), треті мають оптимальні значення (необхідна повторна используемость, документування вимог життєвого циклу). На деякі розроблювач майже не може вплинути (складність продукту, мінливість платформи). Нарешті, життєві несподіванки чи ледь дозволять поліпшити прийняте значення фактора "безперервність персоналу".

Яке ж розв'язок слід вибрати? Найбільш доцільний розв'язок - виключення окремих функцій продукту. Другим (по перевазі) розв'язком є підвищення рівня мультисетевой розробки (однаково це прийде зробити найближчим часом). У якості третього розв'язку можна розглядати ослаблення вимог до режиму роботи в реальному часі. Прийняття ж інших розв'язків залежить від наявності необхідних фахівців або засобів розробки. Втім, остаточний розв'язок повинний вибиратися в процесі переговорів із замовником, коли враховуються всі міркування.

Висновки.

1. Фактори витрат впливають на вихідні параметри програмного проекту.
2. Модель COCOMO II пропонує широкий спектр факторів витрат, що враховують більшість реальних ситуацій в "житті" програмного проекту.
3. Модель COCOMO II забезпечує переклад якісного обґрунтування розв'язку менеджера на кількісні рейки, тим самим підвищуючи об'єктивність прийнятого розв'язку.

Блок змістових модулів 2 – Моделі конструювання

Змістовий модуль 2.1 Базис мови візуального моделювання

Тема №8: Діаграми в UML

(4 години)

Мета: Закріпити матеріал отриманий на лекції та розглянути основні відомості про види діаграм, що використовуються в UML.

План:

1. Діаграми в UML:
    a. Діаграма варіантів використання;
    b. Діаграма класів;
    c. Діаграми поведінки системи;
    d. Діаграми реалізації.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

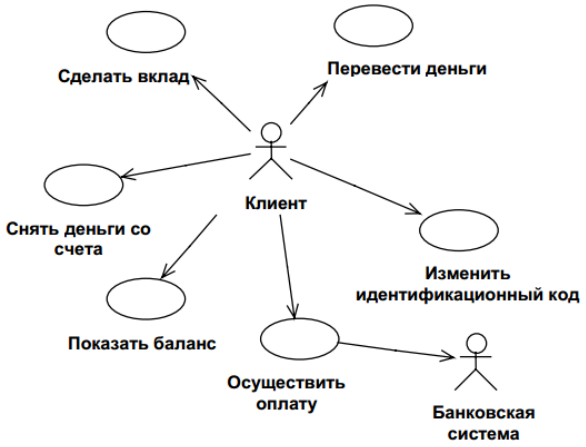
1. Чому потрібно будувати різні діаграми при моделюванні системи?
2. Які діаграми відповідають статичному уявленню про систему?
3. Ви розробляєте комп'ютерну програму для гри в шахи. Яка діаграма UML була б корисної в цьому випадку? Чому?

Розробка моделі будь-якої системи (не тільки програмної) завжди передує її створенню або відновленню. Це необхідно хоча б для того, щоб ясніше уявити собі розв'язуване завдання. Продумані моделі дуже важливі й для взаємодії усередині команди розроблювачів, і для взаєморозуміння із замовником. Зрештою , це дозволяє переконатися в "архітектурній погодженості" проекту до того, як він буде реалізований у коді.

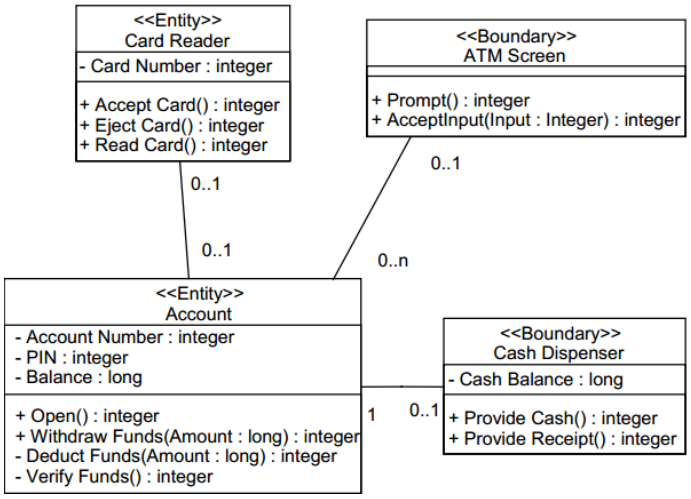
Ми будуємо моделі складних систем, тому що не можемо описати їх повністю, "оглянути одним поглядом". Тому ми виділяємо лише істотні для конкретного завдання властивості системи й будуємо її модель, що відображає ці властивості. Метод об'єктно-орієнтованого аналізу дозволяє описувати реальні складні системи найбільш адекватним образом. Але зі збільшенням складності систем виникає потреба в гарній технології моделювання. Як ми вже говорили в попередній лекції, у якості такої "стандартної" технології використовується уніфікована мова моделювання (Unified Modeling Language, UML), який є графічною мовою для специфікації, візуалізації, проектування й документування систем. За допомогою UML можна розробити докладну модель створюваної системи, що відображає не тільки її концепцію, але й конкретні особливості реалізації. У рамках Uml- Моделі всі вистави про систему фіксуються у вигляді спеціальних графічних конструкцій, що одержали назву діаграм.

Види діаграм:

- діаграми варіантів використання (use case diagrams) - для моделювання бізнес-процесів організації й вимог до створюваної системи);



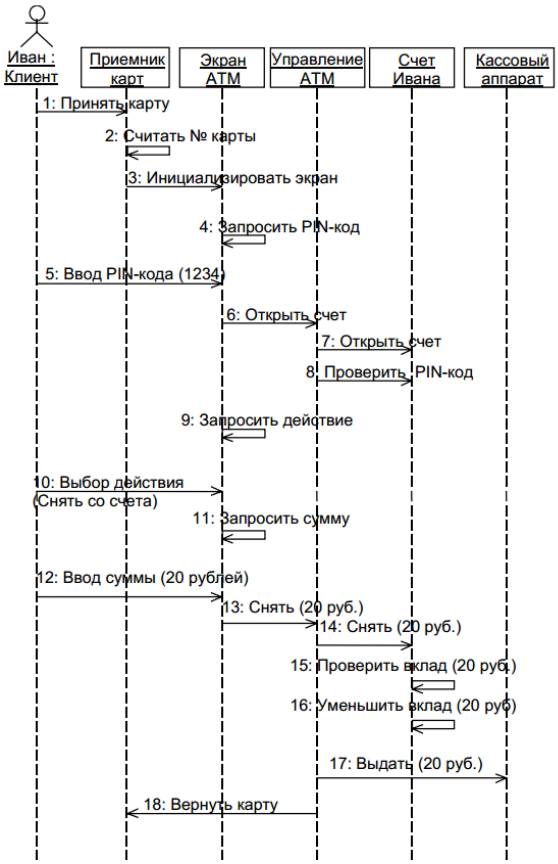
- діаграми класів (class diagrams) - для моделювання статичної структури класів системи й зв'язків між ними;



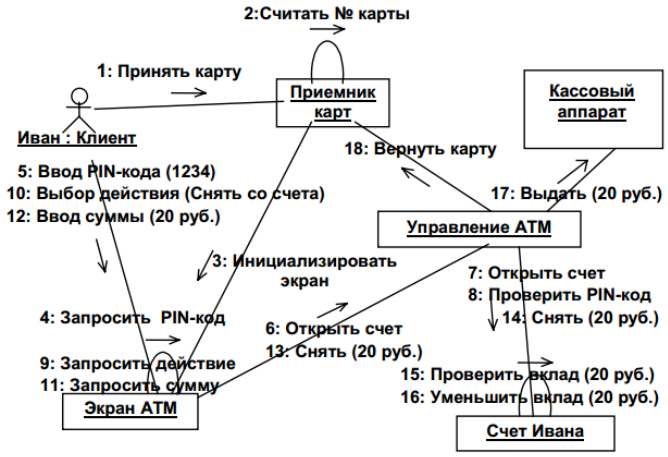
- діаграми поведінки системи (behavior diagrams):

• діаграми взаємодії (interaction diagrams):

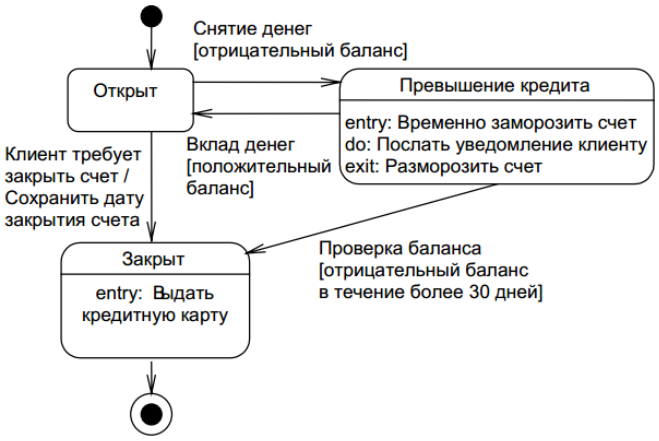
♦ діаграми послідовності (sequence diagrams) і



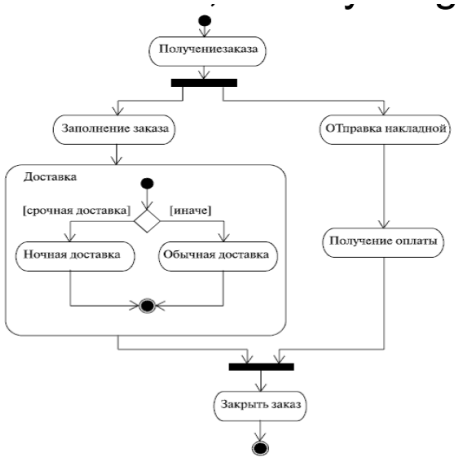
♦ кооперативні діаграми (collaboration diagrams) - для моделювання процесу обміну повідомленнями між об'єктами;



• діаграми станів (statechart diagrams) - для моделювання поведінки об'єктів системи при переході з одного стану в інше;

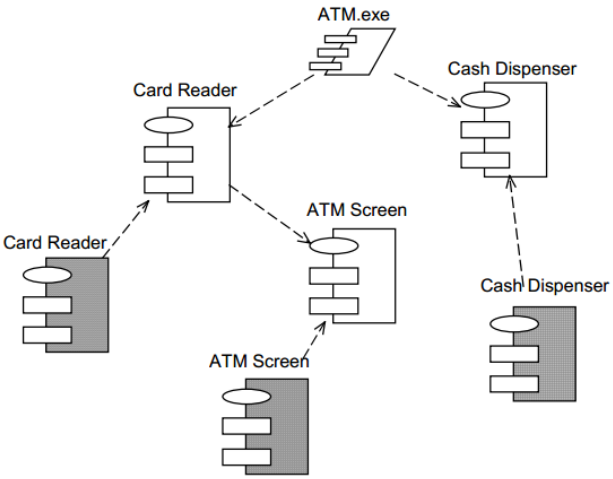


• діаграми діяльностей (activity diagrams) - для моделювання поведінки системи в рамках різних варіантів використання, або моделювання діяльностей;

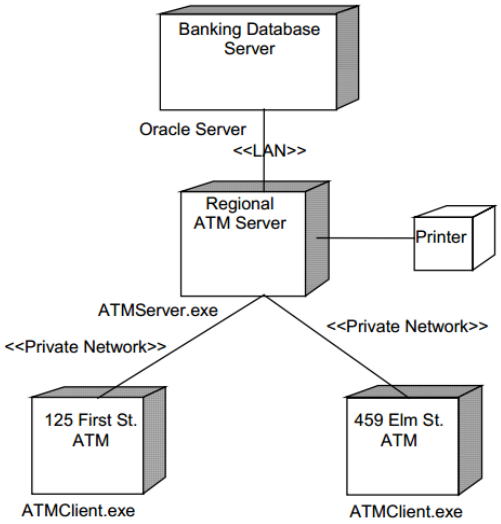


- діаграми реалізації (implementation diagrams):

• діаграми компонентів (component diagrams) - для моделювання ієрархії компонентів (підсистем) системи;



• діаграми розміщення (deployment diagrams) - для моделювання фізичної архітектури системи.



Тема №9: Відносини в мові UML

(8 години)

Мета: Закріпити матеріал отриманий на лекції та отримати досвід встановлення відносин між класами.

План:

1. Відносини між класами:
2. Залежність;
3. Асоціація;
4. Агрегація і композиція;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання та виконати завдання.

Контрольні питання:

1. Які три принципи лежать в основі ООП?
2. Що таке інтерфейс? На якому з базових принципів ООП заснований механізм інтерфейсів?
3. Що таке n- арная асоціація?
4. У чому різниця між агрегацією й композицією?
5. Що таке клас асоціації?

Завдання:

1. Розробіть класи для наступних предметних областей: казка про курочку рябу, Автосалон, театр.
2. Згрупуйте класи попарно використовуючи різні види відносин.

Відносини між класами

Жоден з об'єктів навколишнього нас миру не існує сам по собі. Птаха літають тому, що є повітря, на яке опираються їхні крила. Кожний з нас пов'язаний з масою інших людей різноманітними родинними, професійними й іншими

зв'язками, що припускають різні типи відносин. Точно так само й класи зв'язані між собою. І щоб повною мірою опанувати ООП, нам необхідно зрозуміти суть цих відносин і навчитися їх ідентифікувати.

Ми сказали, що об'єкти перебувають у певних відносинах один з одним. Один з типів таких відносин - це *залежність*. Думаємо, суть такого відношення зрозуміла вже з його назви - залежність виникає тоді, коли реалізація класу одного об'єкта залежить від специфікації операцій класу іншого об'єкта. І якщо зміниться специфікація операцій цього класу, нам неминуче прийде вносити зміни й у залежний клас. Приведемо простий приклад, знов-таки взятий з нашої повсякденності. Іноді до нас у руки попадають відеофайли, відтворити які "з лету" не вдається. Чому? Правильно, тому що на комп'ютері не встановлені відповідні кодеки. Тобто операція "Відтворення", реалізована програмою- медиаплеером, залежить від операції "Декомпресія", реалізованої кодеком. Якщо специфікація операції "Декомпресія" зміниться, прийде міняти код медиаплеера, інакше він просто не зможе працювати з якимось кодеком і, у найкращому разі, завершить свою роботу з помилкою. А от так залежність між класами зображується в UML.

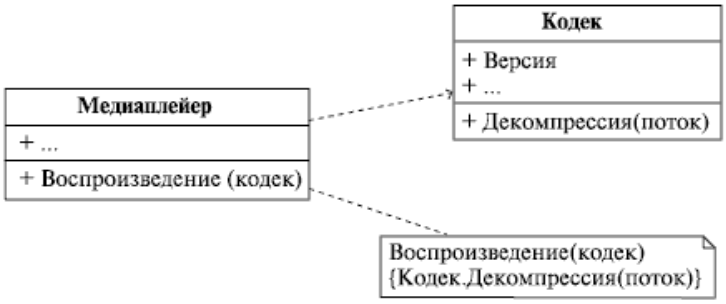


Рисунок 1 – Відношення залежності

Варто відзначити, що залежності на діаграмах зображують далеко не завжди, а тільки в тих випадках, коли їх відображення є важливим для розуміння моделі. Часто залежності лише маються на увазі, тому що логічно випливають із природи класів.

Інший вид відносин між об'єктами - це *асоціація*. Це просто зв'язок між об'єктами, по якій можна між ними переміщатися. Асоціація може мати ім'я, що показує природу відносин між об'єктами, при цьому в імені може вказуватися

напрямок читання зв'язки за допомогою трикутного маркера. Односпрямована асоціація може зображуватися стрілкою. Проілюструємо сказане прикладами.

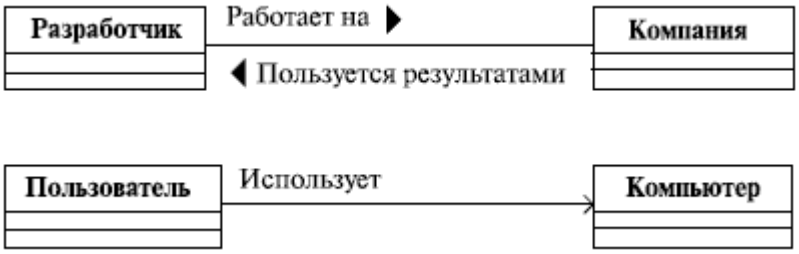


Рисунок 2 – Відношення асоціація

Крім напрямку асоціації, ми можемо вказати на діаграмі ролі, які каждый клас відіграє в даному відношенні, і кратність, тобто кількість об'єктів, зв'язаних відношенням.

Рисунок 3 – Ролі на асоціації

І щодо ролей, і щодо кратності на цій діаграмі всі зрозуміло - людей може взагалі не працювати, працювати в одній або більш компаніях, а от компанії в кожному разі потрібний хоча б один співробітник. До речі, про кратність. Асоціація може поєднувати три й більш класи. У цьому випадку вона називається n- арной і зображується ромбом на перетинанні ліній, як показано на цій діаграмі, запозиченої нами з Zicom Mentor.

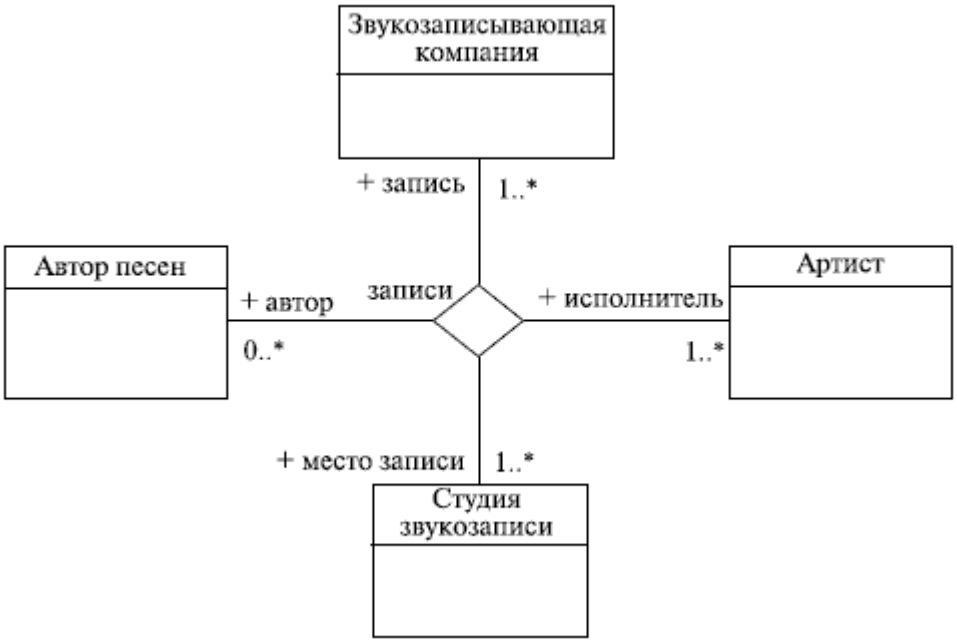


Рисунок 3 - n- арна асоціація

Раніше ми говорили, що асоціація - це "просто зв'язок" між об'єктами. Насправді , у реальності зв'язку бувають "просто зв'язками" украй рідко. Звичайно при ближчому розгляді під асоціацією розуміється більш складне відношення між класами, наприклад, зв'язок типу " частина-ціле". Такий вид асоціації називається асоціацією з агрегуванням. У цьому випадку один клас має більш високий статус (ціле) і складається з нижчих за статусом класів (частин). При цьому виділяють простої й композитне агрегування й говорять про властиво *агрегації й композиції*. Проста агрегація припускає, що частини, відділені від цілого, можуть продовжувати своє існування незалежне від нього. Під композитним же агрегуванням розуміється ситуація, коли ціле володіє своїми частинами і їх час життю відповідає часу життя цілого, тобто незалежно від цілого частини існувати не можуть. Приклади цих видів асоціацій і їх позначень в UML можна побачити на наступній діаграмі.

Рисунок 4 – Агрегація і композиція

Приклади, як нам здається, дуже прості й зрозумілі. Вінчестер можна вийняти з комп'ютера й установити в новий комп'ютер або в Usb- Кишеню, тобто існування жорсткого диска з розбиранням системного блоку не закінчується. А от кнопки без вікон звичайно існувати не можуть - із закриттям вікна кнопки також зникають.

І, нарешті, ще одна важлива річ, що стосується асоціації. У відношенні між двома класами сама асоціація теж може мати властивості й, отже, теж може бути представлена у вигляді класу. Приклад простий

Рисунок 5 – Представлення асоціації як класу

Тема №10: Діаграма Use Case

(4 години)

Мета: Закріпити матеріал отриманий на лекції та отримати практичні навики побудови діаграми Use Case.

План:

1. Діаграми Use Case;
2. Робота з елементами Use Case;
3. Специфікація елементів Use Case;
4. Приклад діаграми Use Case.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання та виконати завдання.

Контрольні питання:

1. Для чого використовують діаграму прецедентів?
2. Як відбувається робота з елементами Use Case?
3. Що таке специфікація елементів Use Case?

Завдання:

1. Розробіть діаграму Use Case для наступних предметних областей:
   a. СТО;
   b. Бібліотека;
2. Розробити специфікації елементів для отриманих діаграм.

Діаграма Use Case визначає поведінка системи з погляду користувача. Діаграма Use Case розглядається як головний засіб для первинного моделювання динаміки системи, використовується для з'ясування вимог до розроблювальної системи, фіксації цих вимог у формі, яка дозволить проводити подальшу розробку.

У російській літературі діаграми Use Case часто називають діаграмами прецедентів, або діаграмами варіантів використання.

До складу діаграм Use Case входять елементи Use Case, актори, а також відносини залежності, узагальнення й асоціації. Як і інші діаграми, діаграми Use Case можуть включати примітки й обмеження. Крім того, діаграми Use Case можуть містити пакети, використовувані для угруповання елементів моделі у великі фрагменти.

Робота з елементами Use Case

Елемент Use Case описує, що повинна робити система, але не визначає, як вона повинна це робити. При моделюванні це дозволяє відокремлювати зовнішня вистава системи від її внутрішньої вистави.

Поведінка елемента Use Case описується потоком подій. Початковий опис виконується в текстовій формі, прозорої для користувача системи. У потоці подій виділяють:

- основний потік і альтернативні потоки поведінки;
- як і коли стартує й закінчується елемент Use Case;
- коли елемент Use Case взаємодіє з акторами;
- якими даними обмінюються актор і система.

Для уточнення й формалізації потоків подій використовують діаграми послідовності. Звичайно одна діаграма послідовності визначає головний потік в елементі Use Case, а інші діаграми - потоки виключень.

У загальному випадку один елемент Use Case описує набір послідовностей, у якому кожна послідовність представляє можливий потік подій. Кожна послідовність називається сценарієм. Сценарій - конкретна послідовність дій, яка ілюструє поведінку. Сценарії є для елемента Use Case майже тим же, чим є екземпляри для класу. Говорять, що сценарій - це екземпляр елемента Use Case.

Специфікація елементів Use Case

Специфікація елемента Use Case - основне джерело інформації для виконання аналізу й проектування системи. Дуже важливо, щоб зміст специфікації був презентовано в повній і конструктивній формі. У загальному випадку специфікація включає головний потік, подпотоки й альтернативні потоки

поведінки. У якості шаблону специфікації представимо опис елемента Use Case "Купувати авіаквиток" для моделі інформаційної системи авіакаси.

Передумова: перед початком цього елемента Use Case повинен бути виконаний елемент Use Case "Заповнити базу даних авіарейсів".

Головний потік

Цей елемент Use Case починається, коли покупець реєструється в системі й уводить свій пароль. Система перевіряє, чи правильний пароль (E-1), і пропонує покупцеві вибрати одне з дій: СТВОРИТИ, ВИЛУЧИТИ, ПЕРЕВІРИТИ, ВИКОНАТИ, ВИХІД.

1. Якщо обрана дія СТВОРИТИ, виконується подпоток S-1: створити замовлення авіаквитка.

2. Якщо обрана дія ВИЛУЧИТИ, виконується подпоток S-2: вилучити замовлення авіаквитка.

3. Якщо обрана дія ПЕРЕВІРИТИ, виконується подпоток S-3: перевірити замовлення авіаквитка.

4. Якщо обрана дія ВИКОНАТИ, виконується подпоток S-4: реалізувати замовлення авіаквитка.

5. Якщо обрана дія ВИХІД, елемент Use Case закінчується.

Подпотоки

S-1: створити замовлення авіаквитка. Система відображає діалогове вікно, що містить поля для пункту призначення й дати польоту. Покупець уводить пункт призначення й дату польоту (E-2). Система відображає параметри авіарейсів (E-3). Покупець уибирає авіарейс. Система зв'язує покупця з обраним авіарейсом (E-4). Повернення до початку елемента Use Case.

S-2: вилучити замовлення авіаквитка. Система відображає параметри замовлення. Покупець підтверджує розв'язок про ліквідацію замовлення (E-5). Система видаляє зв'язок з покупцем (E-6). Повернення до початку елемента Use Case.

S-3: перевірити замовлення авіаквитка. Система виводить (E-7) і відображає параметри замовлення авіаквитка: номер рейсу, пункт призначення, дата, час,

місце, ціну. Коли покупець указує, що він закінчив перевірку, виконується повернення до початку елемента Use Case.

S-4: реалізувати замовлення авіаквитка. Система запитує параметри кредитної карти покупця. Покупець уводить параметри своєї кредитної карти (E-8). Повернення до початку елемента Use Case.

Альтернативні потоки

E-1: уведений неправильний Id- Номер покупця. Покупець може повторити введення Id- Номера або припинити елемент Use Case.

E-2: уведені неправильні пункт призначення/дата польоту. Покупець може повторити введення пункту призначення/дати польоту або припинити елемент Use Case.

E-3: немає підходящих авіарейсів. Покупець інформується, що тепер такий політ неможливий. Повернення до початку елемента Use Case.

E-4: не може бути створений зв'язок між покупцем і авіарейсом. Інформація зберігається, система створить цей зв'язок пізніше. Елемент Use Case триває.

E-5: уведений неправильний номер замовлення. Покупець може повторити введення правильного номера замовлення або припинити елемент Use Case.

E-6: не може бути вилучений зв'язок між покупцем і авіарейсом. Інформація зберігається, система буде видаляти цей зв'язок пізніше. Елемент Use Case триває.

E-7: система не може вивести інформацію замовлення. Повернення до початку елемента Use Case.

E-8: некоректні параметри кредитної карти. Покупець може повторити введення параметрів карти або припинити елемент Use Case.

Таким чином, у даній специфікації зафіксоване, що усередині елемента Use Case перебуває один основний потік і дванадцять допоміжних потоків дій. Надалі розроблювач може ухвалити рішення щодо виділення із цього елемента Use Case самостійних елементів Use Case. Очевидно, що якщо самостійний елемент Use Case містить подпоток, те його слід підключати до базового елемента Use Case відношенням include. У свою чергу, самостійний елемент Use Case з

альтернативним потоком підключається до базового елемента Use Case відношенням extend.

Приклад діаграми Use Case

Найбільші труднощі при побудові діаграм Use Case викликає застосування відносин включення й розширення. Дуже важливо розібратися у відмінних рисах цих відносин, специфіці взаємодії елементів Use Case, що з'єднуються з їхньою допомогою.

Приклад діаграми Use Case, у якій використані відносини включення й розширення, наведений на мал. 1.

У цій діаграмі один базовий елемент Use Case Сеанс банкомата, який взаємодіє з актором Клієнт. До базового елемента Use Case підключено два розширювальні елементи Use Case (Стан, Зняти) і два, що включаються елемента Use Case (Ідентифікація клієнта, Перевірка рахунку). У свою чергу, до елемента Use Case Ідентифікація клієнта підключений елемент, що включається, Use Case Перевірити вірогідність, а до елемента Use Case Зняти - розширювальний елемент Use Case Захоплення карти (він же розширює елемент Use Case Перевірити вірогідність).

Бачимо, що елемент Use Case Сеанс банкомата має дві крапки розширення (діалог можливий, видача квитанції), а елементи Use Case Зняти й Перевірити вірогідність - по одній крапці розширення (перевірка зняття й перевірка відповідно). У крапки розширення можлива вставка поведінки з розширювального елемента Use Case. Вставка відбувається, якщо виконується умова розширення:

- для розширювального елемента Use Case Стан - це умова запит стану;
- для розширювального елемента Use Case Зняти - це умова запит зняття;
- для розширювального елемента Use Case Захоплення карти - це умова список підозр.

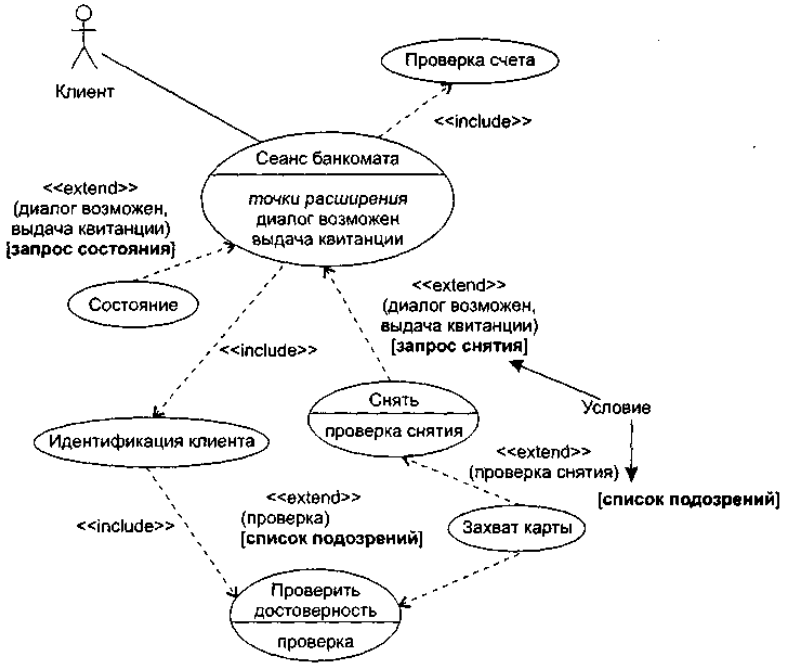


Рисунок 1 - Використання включення й розширення

Для розширюваного (базового) елемента Use Case ці умови є зовнішніми, тобто формованими поза ним. Іншими словами, елементу Use Case Сеанс банкомата нічого не відомо про умови запит стану й запит зняття, а елементам Use Case Зняти й Перевірити вірогідність - про умову список підозр. Умови розширення є наслідками подій, що відбуваються в зовнішньому середовищі.

Стрілки розширення в діаграмі підписані. Крім стереотипу, тут зазначені:

- у круглих дужках - імена крапок розширення;
- у квадратних дужках - умова розширення.

Опис розширювального елемента Use Case розділене на сегменти, кожний сегмент обслуговує одну крапку розширення базового елемента Use Case.

Кількість сегментів розширювального елемента Use Case дорівнює кількості крапок розширення базового елемента Use Case. Перший сегмент розширювального елемента Use Case починається з умови розширення, умова записується тільки один раз, його дія поширюється й на всі інші сегменти.

Поведінка базового елемента Use Case задається внутрішнім потоком подій, аж до крапки розширення. У крапці розширення можливе виконання розширювального елемента Use Case, після чого відновляється робота внутрішнього потоку.

Специфікації елементів Use Case розглянутої діаграми мають такий вигляд:

Елемент Use Case Сеанс банкомата

include (Ідентифікація клаієнта)
include (Перевірка рахунку)
(діалог допустимий)
надрукувати заголовок квитанції
(видача квітанції)
кінець сеансу
	//включення
	//включення
	//перша крапка розширення
	
	//друга крапка розширення

Розширювальний елемент Use Case Стан

сегмент	//початок першого сегменту
прийняти запит стану	//умова розширення
відобразити інформацію о стані рахунку	
сегмент	//друга точка розширення
кінец сеансу	

Розширювальний елемент Use Case Зняти

сегмент	//початок першого сегменту
прийняти запит на зняття	//умова расширення
Визначити суму	
(перевірка зняття)	//точка расширення
сегмент	//початок другого сегменту
Надрукувати знімаємо суму	
Видати готівку	

Розширювальний елемент Use Case Захоплення карти

сегмент принйяти список підозр захопити картку кінец сеансу	//початок єдиного сегменту //умова расширення

Елемент, що включається, Use Case Ідентифікація клієнта

Отримати ім’я клієнта include (Перевірити) отримати номер рахунку клієнта	//включення

Елемент, що включається, Use Case Перевірка рахунку

Встановити поєднання з БД рахунків Отримати стан і обмеження рахунку

Елемент, що включається, Use Case Перевірити вірогідність

Встановити з’єднання з БД клієнтів Отримати параметри клієнта (перевірка) //точка расширення

Опишемо можливу поведінку моделі, що задається цієї діаграмою.

Актор Клієнт ініціює дії базового елемента Use Case Сеанс банкомата. На першому кроці запускається елемент, що включається, Use Case Ідентифікація клієнта. Цей елемент Use Case одержує ім'я клієнта й запускає елемент Use Case Перевірити вірогідність, у результаті чого встановлюється з'єднання з базою даних клієнтів і виходять параметри клієнта.

Якщо до цього моменту виконується умова розширення список підозр, то "спрацьовує" розширювальний елемент Use Case Захоплення карти, карта заарештовується й робота системи припиняється.

А якщо ні, то відбувається повернення до елемента Use Case Ідентифікація клієнта, який одержує номер рахунку клієнта й повертає керування базовому елементу Use Case.

Базовий елемент Use Case переходить до другого кроку роботи - викликає елемент, що включається, Use Case Перевірка рахунку, який установлює з'єднання з базою даних рахунків і одержує стан і обмеження рахунку.

Керування знову вертається до базового елемента Use Case. Базовий елемент Use Case переходить до першої крапки розширення діалог можливий. У цій крапці можливе підключення одного із двох розширювальних елементів Use Case.

Покладемо, що до цього моменту виконується умова розширення запит стану, тому запускається перший сегмент елемента Use Case Стан. У результаті відображається інформація про стан рахунку й керування передається базовому елементу Use Case. У базовому елементі Use Case друкується заголовок квитанції й забезпечується перехід до другої крапки розширення видача квитанції.

Оскільки в активному стані продовжує перебувати розширювальний елемент Use Case Стан, запускається його другий сегмент - у квитанції друкується інформація про стан рахунку.

Востаннє керування вертається до базового елемента Use Case - завершується сеанс роботи банкомата.

Тема №11: Правила побудови діаграми класів

(2 години)

Мета: Розглянути рекомендації для побудови діаграми класів та отримати власний досвід побудови діаграм класів.

План:

1. Діаграма класів;
2. Рекомендації з побудови діаграм класів.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Виконати завдання.

Завдання:

1. Побудуйте діаграму класів для наступних предметних областей:
    a. Таксопарк;
    b. Музей;
    c. Казка «Колобок».

Центральне місце в ООАП займає розробка логічної моделі системи у вигляді діаграми класів. Нотація класів у мові UML проста й інтуїтивно зрозуміла всім, хто коли-небудь мав досвід роботи з Case- Инструментариями. Схожа нотація застосовується й для об'єктів - екземплярів класу, з тим відмінністю, що до імені класу додається ім'я об'єкта й увесь напис підкреслюється.

Нотація UML надає широкі можливості для відображення додаткової інформації (абстрактні операції й класи, стереотипи, загальні й частки методи, деталізовані інтерфейси, параметризованные класи). При цьому можливо використання графічних зображень для асоціацій і їх специфічних властивостей, таких як відношення агрегації, коли складовими частинами класу можуть виступати інші класи.

Діаграма класів (class diagram) служить для вистави статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування. Діаграма класів може відбивати, зокрема , різні взаємозв'язки між окремими сутностями предметної області, такими як об'єкти й підсистеми, а також описує їхню внутрішню структуру й типи відносин. На даній діаграмі не вказується інформація про тимчасові аспекти функціонування системи. Із цього погляду діаграма класів є подальшим розвитком концептуальної моделі проектованої системи.

Діаграма класів являє собою деякий граф, вершинами якого є елементи типу "класифікатор", які зв'язані різними типами структурних відносин. Слід помітити, що діаграма класів може також містити інтерфейси, пакети, відносини й навіть окремі екземпляри, такі як об'єкти й зв'язки. Коли говорять про дану діаграму, мають на увазі статичну структурну модель проектованої системи. Тому діаграму класів прийнято вважати графічним представленому таких структурних взаємозв'язків логічної моделі системи, які не залежать або інваріантні від часу.

Діаграма класів складається з безлічі елементів, які в сукупності відбивають декларативні знання про предметну область. Ці знання інтерпретуються в базових поняттях мови UML, таких як класи, інтерфейси й відносини між ними і їх складовими компонентами. При цьому окремі компоненти цієї діаграми можуть утворювати пакети для вистави більш загальної моделі системи. Якщо діаграма класів є частиною деякого пакета, то її компоненти повинні відповідати елементам цього пакета, включаючи можливі посилання на елементи з інших пакетів.

У загальному випадку пакет статичної структурної моделі може бути представлено у вигляді однієї або декількох діаграм класів. Декомпозиція деякої вистави на окремі діаграми виконується з метою зручності й графічної візуалізації структурних взаємозв'язків предметної області. При цьому компоненти діаграми відповідають елементам статичної семантичної моделі. Модель системи, у свою чергу, повинна бути погоджена із внутрішньою структурою класів, яка описується мовою UML.

Рекомендації з побудови діаграм класів

Процес розробки діаграми класів займає центральне місце в ООАП складних систем. Від уміння правильно вибрати класи й установити між ними взаємозв'язку часто залежить не тільки успіх процесу проектування, але й продуктивність виконання програми. Як показує практика ООП, кожний програміст у своїй роботі прагне тією чи іншою мірою використовувати вже накопичений особистий досвід при розробці нових проектів. Це обумовлене бажанням звести нове завдання до вже вирішених, щоб мати можливість використовувати не тільки перевірені фрагменти програмного коду, але й окремі компоненти в цілому (бібліотеки компонентів).

Такий стереотипний підхід дозволяє суттєво скоротити строки реалізації проекту, однак прийнятний лише в тому випадку, коли новий проект концептуально й технологічно не занадто відрізняється від попередніх. А якщо ні, то платою за скорочення строків проекту може стати його реалізація на застарілій технологічній базі. Що стосується властиво об'єктної структуризації предметної області, те тут доречно дотримуватися тих рекомендацій, які накопичені в ООП.

При визначенні класів, атрибутів і операцій і завданні їх імен і типів перед вітчизняними розроблювачами завжди встає мимовільне питання: який з мов використовувати в якості природнього, російський або англійський? З одного боку, використання рідної мови для опису моделі є найбільш природнім способом її вистави й найбільшою мірою відбиває комунікативну функцію моделі системи. З іншого боку, розробка моделі є лише одним з етапів розробки відповідної системи, а застосування інструментальних засобів для її реалізації в абсолютній більшості випадків вимагає використання англомовних термінів. Саме тому виникає характерна неоднозначність, з якої, очевидно, зовсім незнайома англомовна аудиторія.

Відповідаючи на поставлений вище питання, слід зазначити, що найбільше доцільно дотримуватися наступних рекомендацій. При побудові діаграми варіантів використання концептуальною моделлю, що є найбільш загальною, проектованої системи, застосування російськомовних термінів є не тільки виправданим з погляду опису структури предметної області, але й ефективним з погляду

комунікативної взаємодії із замовником і користувачами. При побудові інших типів діаграм слід дотримуватися розумного компромісу.

Зокрема , на початкових етапах розробки діаграм доцільність використання російськомовних термінів цілком очевидна й виправдана. Однак, у міру готовності графічної моделі для реалізації у вигляді програмної системи й передачі її для подальшої роботи програмістам, акцент може зміщатися убік використання англомовних термінів, які тією чи іншою мірою відбивають особливості мови програмування, на якій передбачається реалізація даної моделі.

Більше того, використання Case- Инструментариев для автоматизації ООАП, найчастіше , накладає свої власні вимоги на мову специфікації моделей. Саме із цієї причини більшість прикладів у літературі даються в англомовній виставі, а при їхньому перекладі на росіянин може бути втрачена не тільки точність формулювань, але й семантика відповідних понять.

Після розробки діаграми класів процеси ООАП може бути продовжений у двох напрямках. З одного боку, якщо поведінка системи тривіально, те можна приступитися до розробки діаграм кооперації й компонентів. Однак для складних динамічних систем поведінка представляє найважливіший аспект їх функціонування. Деталізація поведінки здійснюється послідовно при розробці діаграм станів, послідовності й діяльності.

Тема №12: Діаграма послідовності дій

(4 години)

Мета: Закріпити матеріал отриманий на лекції та отримати практичні навики побудови діаграми послідовності дій.

План:

1. Діаграми послідовності:
    a. лінія життя об'єкта;
    b. фокус керування;
        i. паралельні лінії життя;
        ii. розгалуження.

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання та виконати завдання.

Контрольні питання:

1. Для чого використовується діаграма послідовності?
2. Що загального в діаграмі послідовності й діаграмі співробітництва? Чим вони відрізняються друг від друга?
3. Як відображається порядок передачі повідомлень у діаграмі послідовності?
4. Коли зручніше застосовувати діаграми послідовності?

Завдання:

1. Побудувати діаграму послідовності для наступних предметних областей:
    a. СТО;
    b. Бібліотека.

Діаграма послідовності - різновид діаграм взаємодії. Відбиваючи сценарій поведінки в системі, ця діаграма забезпечує більш наочна вистава порядку передачі повідомлень. Правда, вона не дозволяє показати такі деталі, які видні на діаграмі співробітництва (структурні характеристики об'єктів і зв'язків).

Графічно діаграма послідовності - різновид таблиці, який показує об'єкти, розміщені уздовж осі X, і повідомлення, упорядковані за часом уздовж осі Y.

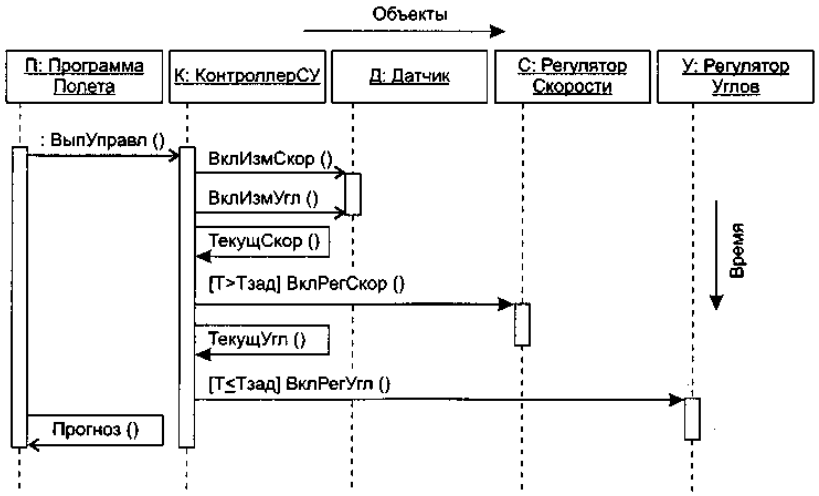


Рисунок 1 - Діаграма послідовності системи керування польотом

Як показано на мал. 1, об'єкти, що брати участь у взаємодії, містяться на вершині діаграми, уздовж осі X. Звичайно ліворуч розміщається об'єкт, що ініціює взаємодія, а праворуч - об'єкти по зростанню підпорядкованості. Повідомлення, що посилають і прийняті об'єктами, містяться уздовж осі Y у порядку зростання часу від вершини до підстави діаграми. Використовуються ті ж синтаксис і позначення синхронізації, що й у діаграмах співробітництва. Таким чином, забезпечується проста візуальна вистава потоку керування в часі.

Від діаграм співробітництва діаграми послідовності відрізняють дві важливі характеристики.

Перша характеристика - *лінія життя об'єкта*.

Лінія життя об'єкта - це вертикальна пунктирна лінія, яка позначає період існування об'єкта. Більшість об'єктів існують протягом усього взаємодії, їх лінії життя тягнуться від вершини дощенту діаграми. Втім, об'єкти можуть створюватися в ході взаємодії. Їхні лінії життя починаються з моменту приймання повідомлення "create". Крім того, об'єкти можуть знищуватися в ході взаємодії. Їхні лінії життя закінчуються з моменту приймання повідомлення "destroy". Як презентовано на мал. 2, знищення лінії життя відзначається позначкою X наприкінці лінії:

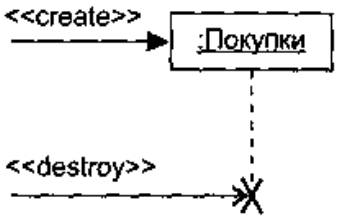


Рисунок 2 - Створення й знищення об'єкта

Друга характеристика - *фокус керування*.

Фокус керування - це високий тонкий прямокутник, що відображає період часу, протягом якого об'єкт виконує дія (свою або підлеглу процедуру). Вершина прямокутника відзначає початок дії, а підстава - його завершення. Момент завершення може маркіруватися повідомленням повернення, яке показується пунктирною стрілкою. Можна показати вкладення фокуса керування (наприклад, рекурсивний виклик власної операції). Для цього другий фокус керування рисується небагато правее першого (мал. 3).

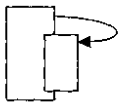


Рисунок 3 - Вкладення фокусів керування

Для відображення "умовності" лінія життя може бути розділена на кілька паралельних ліній життя. Кожна окрема лінія відповідає умовному розгалуженню у взаємодії. Далі в деякій крапці лінії життя можуть бути знову злиті (мал. 4).

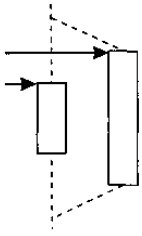


Рисунок 4 - Паралельні лінії життя

Розгалуження показується безліччю стрілок, що йдуть із однієї крапки. Кожна стрілка відзначається сторожовою умовою (мал. 5).

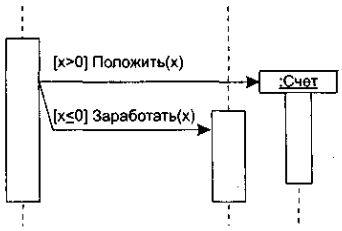


Рисунок 5 – Розгалуження

Тема №13: Діаграма схем станів

(4 години)

Мета: Закріпити матеріал отриманий на лекції та розглянути поняття складного стану та під-стану.

План:

1. Діаграма станів;
2. Складений стан і під-стан;
3. Складні переходи;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Для чого використовуються діаграми станів?
2. Що таке складений стан?
3. Які бувають підстани?
4. Для чого використовуються складні переходи?

Для моделювання поведінки на логічному рівні в мові UML можуть використовуватися відразу кілька канонічних діаграм: станів, діяльності, послідовності й кооперації, кожна з яких фіксує увага на окремому аспекті функціонування системи. На відміну від інших діаграм діаграма станів описує

процес зміни станів тільки одного класу, а точніше - одного екземпляра певного класу, тобто моделює всі можливі зміни в стані конкретного об'єкта. При цьому зміна стану об'єкта може бути викликане зовнішніми впливами з боку інших об'єктів або ззовні. Саме для опису реакції об'єкта на подібні зовнішні впливи й використовуються діаграми станів.

Головне призначення цієї діаграми - описати можливі послідовності станів і переходів, які в сукупності характеризують поведінка елемента моделі протягом його життєвого циклу. Діаграма станів представляє динамічну поведінку сутностей, на основі специфікації їх реакції на сприйняття деяких конкретних подій. Системи, які реагують на зовнішні дії від інших систем або від користувачів, іноді називають реактивними. Якщо такі дії ініціюються в довільні випадкові моменти часу, то говорять про асинхронну поведінку моделі.

Хоча діаграми станів найчастіше використовуються для опису поведінки окремих екземплярів класів (об'єктів), але вони також можуть бути застосовані для специфікації функціональності інших компонентів моделей, таких як варіанти використання, актори, підсистеми, операції й методи.

Діаграма станів по суті є графом спеціального виду, який представляє деякий автомат. Поняття автомата в контексті UML має досить специфічну семантику, засновану на теорії автоматів. Вершинами цього графа є стани й деякі інші типи елементів автомата (псевдосостояния), які зображуються відповідними графічними символами. Дуги графа служать для позначення переходів зі стану в стан. Діаграми станів можуть бути вкладені друг у друга, образуая вкладені діаграми більш детальної вистави окремих елементів моделі. Для розуміння семантики конкретної діаграми станів необхідно представляти не тільки особливості поведінки моделируемой сутності, але й знати загальні відомості по теорії автоматів.

Складений стан і подсостояние

Складений стан (composite state) - такий складний стан, який складається з інших вкладених у нього станів. Останні будуть виступати стосовно першого як подсостояния (substate). Хоча між ними має місце відношення композиції, графічно всі вершини діаграми, які відповідають вкладеним станам, зображуються усередині

символу складеного стану (мал. 6). У цьому випадку розміри графічного символу складеного стану збільшуються, так щоб умістити в себе всі подсостояния.

Рисунок 1 - Графічна вистава складеного стану із двома вкладеними в нього послідовними подсостояниями

Складений стан може містити два або більш паралельних подавтомата або трохи послідовних подсостояний. Кожний складний стан може уточнюватися тільки одним із зазначених способів. При цьому кожне з подсостояний, у свою чергу, може бути складеним станом і містити усередині себе інші вкладені подсостояния. Кількість рівнів вкладеності складених станів не фіксоване в мові UML.

Послідовні подсостояния

Послідовні подсостояния (sequential substates) використовуються для моделювання такої поведінки об'єкта, під час якого в кожний момент часу об'єкт може перебувати в одному й тільки одному подсостояний. Поведінка об'єкта в цьому випадку являє собою послідовну зміну подсостояний, починаючи від початкового й закінчуючи кінцевим подсостояниями. Хоча об'єкт продовжує перебувати в складеному стані, уведення в розгляд послідовних подсостояний дозволяє врахувати більш тонкі логічні аспекти його внутрішньої поведінки.

Для прикладу розглянемо в якості моделируемого об'єкта звичайний телефонний апарат. Він може перебувати в різних станах, одним з яких є стан дозвона до абонента. Очевидно, для того щоб подзвонити, необхідно зняти слухавку , почути тоновий сигнал, після чого набрати потрібний телефонний номер. Таким чином, стан дозвона до абонента є складовим і складається із двох послідовних подсостояний: "підняти слухавку " і "набрати телефонний номер".

Фрагмент діаграми станів для цього прикладу містить один складений стан і два послідовні подсостояний (мал. 2).

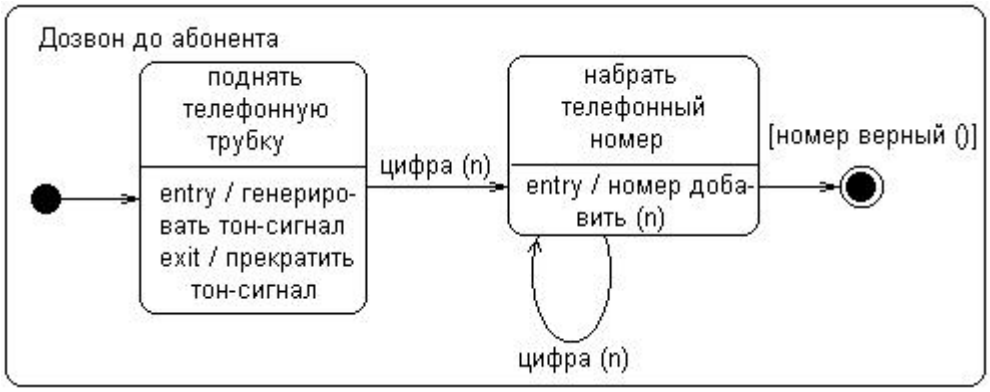


Рисунок 2 - Приклад складеного стану із двома вкладеними послідовними подсостояниями

Деяких пояснень можуть зажадати переходи. Два з них специфицируют подія- тригер набір цифри, яке має ім'я "цифра" з параметром "п". У якості параметра, як неважко припустити, виступає окрема цифра на диску телефонного апарата. Перехід з початкового під- стани нетриггерный, оскільки він не містить ніякого рядка тексту. Останній перехід у кінцеве подсостояние не має події-тригера, але має сторожову умову, що перевіряє правильність набраного номера абонента. Тільки у випадку істинності цієї умови телефонний апарат може перейти в кінцеве подсостояние, яке характеризує суперстан "дозвон до абонента" у цілому.

Складений стан може містити в якості вкладених подсостояний початковий і кінцевий стану. При цьому початкове подсостояние є вихідним, коли відбувається перехід об'єкта в даний складений стан. Якщо складений стан містить усередині себе кінцеве подсостояние, то перехід у цей вкладений кінцевий стан означає завершення знаходження об'єкта в даному вкладеному стані. Важливо пам'ятати, що для послідовних подсостояний початковий і кінцевий стану повинні бути єдиними в кожному складеному стані.

Це можна пояснити в такий спосіб. Кожна сукупність вкладених послідовних подсостояний являє собою подавтомат того автомата, якому належить розглянутий складений стан. Оскільки кожний автомат може мати по визначенню єдине початкове і єдине кінцеве стану, то для подавтомата ця умова також повинна виконуватися (мал. 2).

Паралельні подсостояния

Паралельні подсостояния (concurrent substates) дозволяють специфицировать два й більш подавтомата, які можуть виконуватися паралельно усередині складеної події. Кожний з подавтоматов займає деяку область (регіон) усередині складеного стану, яка відділяється від інших горизонтальної пунктирною лінією. Якщо на діаграмі станів є складений стан із вкладеними паралельними подсостояниями, то об'єкт може одночасно перебувати в кожному із цих подсостояний.

Однак окремі паралельні подсостояния можуть, у свою чергу, складатися з декількох послідовних подсостояний (подавтоматы 1 і 2 на мал. 3). У цьому випадку по визначенню об'єкт може перебувати тільки в одному з послідовних подсостояний подавтомата. Таким чином, для абстрактного прикладу (мал. 3) припустиме одночасне знаходження об'єкта в подсостояниях (1, 3, 4), (2, 3, 4), (1, 3, 5), (2, 3, 5). Неприпустиме знаходження об'єкта одночасно в подсостояниях (1, 2,3) або (3, 4, 5).

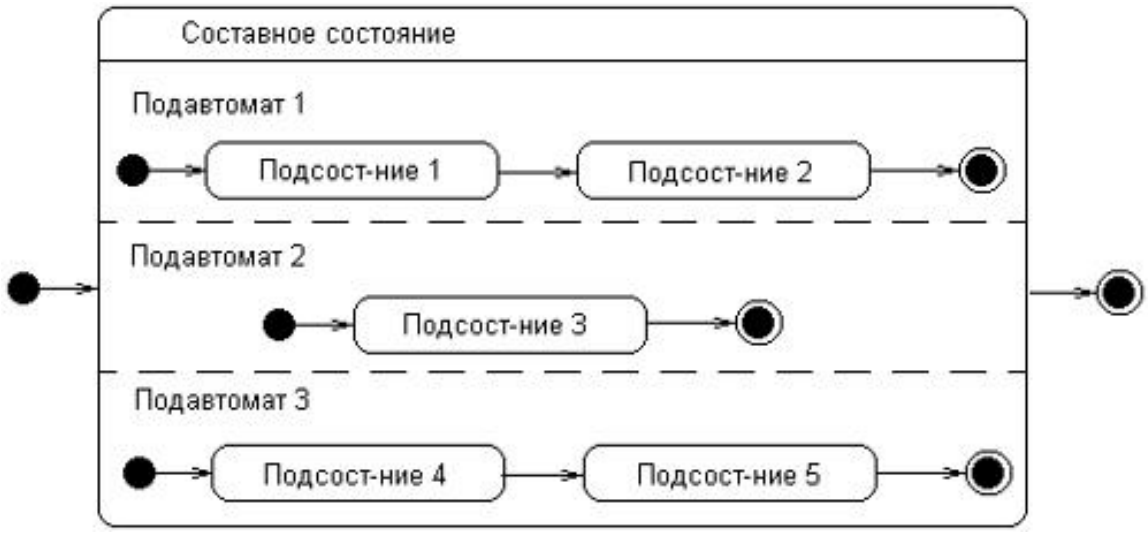


Рисунок 3 - Графічне зображення складеного стану із вкладеними паралельними подсостояниями

Оскільки кожний регіон вкладеного стану специфицирует деякий подавтомат, те для кожного із вкладених подавтоматов можуть бути визначені власні початкове й кінцеві подсостояния (мал. 8). При переході в даний складений стан кожний з подавтоматов виявляється у своєму початковому подсостояний. Далі відбувається паралельне виконання кожного із цих подавтоматов, причому вихід зі

складеного стану буде можливий лише в тому випадку, коли всі подавтоматы будуть перебувати у своїх кінцевих подсостояниях.

Якщо який-небудь із подавтоматов прийшов у свій кінцевий стан раніше інших, то він повинен очікувати, поки й інші подавтоматы не прийдуть у свої кінцеві стани.

У деяких випадках буває бажане сховати внутрішню структуру складеного стану. Наприклад, окремий подавтомат, специфицирующий складений стан, може бути настільки більшим по масштабу, що його візуалізація утруднить загальну виставу діаграми станів. У подібній ситуації допускається не розкривати на вихідній діаграмі станів даний складений стан, а вказати в правому нижньому куті спеціальний символ- піктограму (мал. 4). Надалі діаграма станів для відповідного подавтомата може бути зображена окремо від основної з необхідними коментарями.

Рисунок 4 - Складений стан зі схованою внутрішньою структурою й спеціальною піктограмою

Переходи між паралельними станами

В окремих випадках перехід може мати кілька станів- джерел і кілька цільових станів. Такий перехід одержав спеціальну назву - паралельний перехід. Уведення в розгляд паралельного переходу обумовлене необхідністю синхронізувати й/або розділити окремі подпроцессы на паралельні нитки без специфікації додаткової синхронізації в паралельних подавтоматах.

Графічно такий перехід зображується вертикальною рискою, аналогічно позначенню переходу у відомому формалізмі мереж Петри. Якщо паралельний перехід має дві або більш вхідних дуг (мал. 5, а), то його називають з'єднанням (join). Якщо ж він має дві або більш вихідних з нього дуг (мал. 5, б), то його

називають розгалуженням (fork). Текстовий рядок специфікації паралельного переходу записується поруч із рискою й ставиться до всіх вхідних (вихідним) дугам.

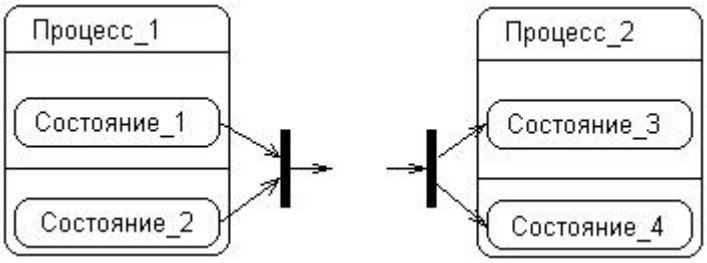


Рисунок 5 - Графічне зображення паралельного переходу з паралельних станів (а) і паралельного переходу в паралельні стани (б)

Спрацьовування паралельного переходу відбувається в такий спосіб. У першому випадку (перехід-з'єднання) перехід спрацьовує, якщо має місце подія-тригер для всіх вихідних станів цього переходу, і виконане (при його наявності) сторожова умова. При спрацьовуванні переходу- з'єднання одночасно покидаются всі вихідні стани переходу (стану 1 і 2) і відбувається перехід у цільовий стан. При цьому кожне з вихідних станів переходу повинне належати окремому подавтомату, що входить до складу автомата (процесу 1).

У другому випадку (розгалуження) відбувається розщеплення автомата на два подавтомата паралельні галузі, що утворюють, вкладених подсостояний. При цьому послу спрацьовування переходу моделируемый об'єкт одночасно буде перебувати у всіх цільових станах цього переходу (стану 3 і 4). Далі процес зміни станів буде протікати згідно з раніше розглянутими правилами для складених станів.

Переходи між складеними станами

Перехід, стрільця якого з'єднана із границею деякого складеного стану, позначає перехід у складений стан (перехід b на мал. 6). Він еквівалентний переходу в початковий стан кожного з подавтоматов (можливо, єдиному), що входять до складу даного суперстану. Перехід, що виходить зі складеного стану (переходи f і g на мал. 6), ставиться до кожного із вкладених подсостояний. Це означає, що об'єкт може покинути складений суперстан, перебуваючи в кожному з

його подсостояний. Для цього цілком достатньо виконання події й сторожової умови.

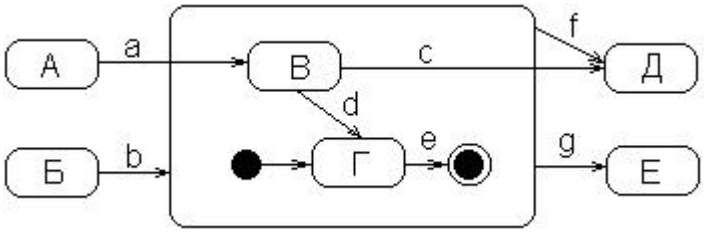


Рисунок 6 - Різні варіанти переходів в (з) складений стан

Іноді бажане реалізувати ситуацію, коли вихід з окремого вкладеного стану відповідав би виходу й зі складеного стану теж. У цьому випадку зображують перехід, який безпосередньо виходить із вкладеного подсостояния за кордон суперстану (перехід з на мал. 6). Аналогічно, допускається зображення переходів, що входять ззовні складеного стану в окремий вкладений стан (перехід а на мал. 6).

Тема №14: Діаграма діяльності

(4 години)

Мета: Закріпити матеріал отриманий на лекції та отримати практичні навики побудови діаграми діяльності.

План:

1. Діаграми діяльності:
    a. Стан дії;
    b. Стан під-діяльності;
    c. Допоміжні вершини;

Домашнє завдання:

Необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання та виконати завдання.

Контрольні питання:

1. Що таке стан дії?
2. Для чого використовується стан під-діяльності?

3. Які допоміжні вершини можуть бути присутніми на діаграмі діяльності?

Завдання:

1. Побудуйте діаграму діяльності для наступних предметних областей:
   a. СТО;
   b. Бібліотека.

Діаграми діяльності

Діаграма діяльності представляє особливу форму кінцевого автомата, у якій показуються процес обчислень і потоки робіт. У ній виділяються не звичайні стани об'єкта, а стану виконуваних обчислень - стану дій. При цьому покладається, що процес обчислень не переривається зовнішніми подіями. Словом, діаграми діяльності дуже схожі на блок-схеми алгоритмів.

Основною вершиною в діаграмі діяльності є стан дії (мал. 1), яка зображується як прямокутник із закругленими бічними сторонами.

Рисунок 1 - Стан дії

Стан дії вважається атомарним (дія не можна перервати) і виконується за один квант часу, його не можна піддати декомпозиції. Якщо потрібно представити складну дію, яку можна піддати подальшої декомпозиції (розбити на ряд більш простих дій), то використовують стан під-діяльності. Зображення стану під-діяльності містить піктограму в правому нижньому куті (мал. 2).

Рисунок 2 - Стан під-діяльності

Фактично в дану вершину вписується ім'я іншої діаграми, що має внутрішню структуру.

Переходи між вершинами - станами дій - зображуються у вигляді стрілок. Переходи виконуються по закінченню дій.

Крім того, у діаграмах діяльності використовуються *допоміжні вершини*:

- розв'язок (ромбик з однієї вхідної й декількома вихідними стрілками);

- об'єднання (ромбик з декількома вхідними й однією вихідною стрілкою);
- лінійка синхронізації - поділ (жирна горизонтальна лінія з однієї вхідної й декількома вихідними стрілками);
- лінійка синхронізації - злиття (жирна горизонтальна лінія з декількома вхідними й однією вихідною стрілкою);
- початковий стан (чорний кружок);
- кінцевий стан (не зафарбований кружок, у якому розміщений чорний кружок меншого розміру).

Вершина "розв'язок" дозволяє відобразити розгалуження обчислювального процесу стрілки, що виходять із нього, позначаються сторожовими умовами розгалуження.

Вершина "об'єднання" відзначає крапку злиття альтернативних потоків дій.

Лінійки синхронізації дозволяють показати паралельні потоки дій, відзначаючи крапки їх синхронізації при запуску (момент поділу) і при завершенні (момент злиття).

Приклад діаграми діяльності наведений на мал. 3. Ця діаграма описує діяльність покупця в Інтернет-Магазині. Тут представлено дві крапки розгалуження - для вибору способу пошуку товару й для ухвалення рішення про покупку. Присутні три лінійки синхронізації: верхня відбиває поділ на два паралельні процеси, середня відбиває й поділ, і злиття процесів, а нижня - тільки злиття процесів.

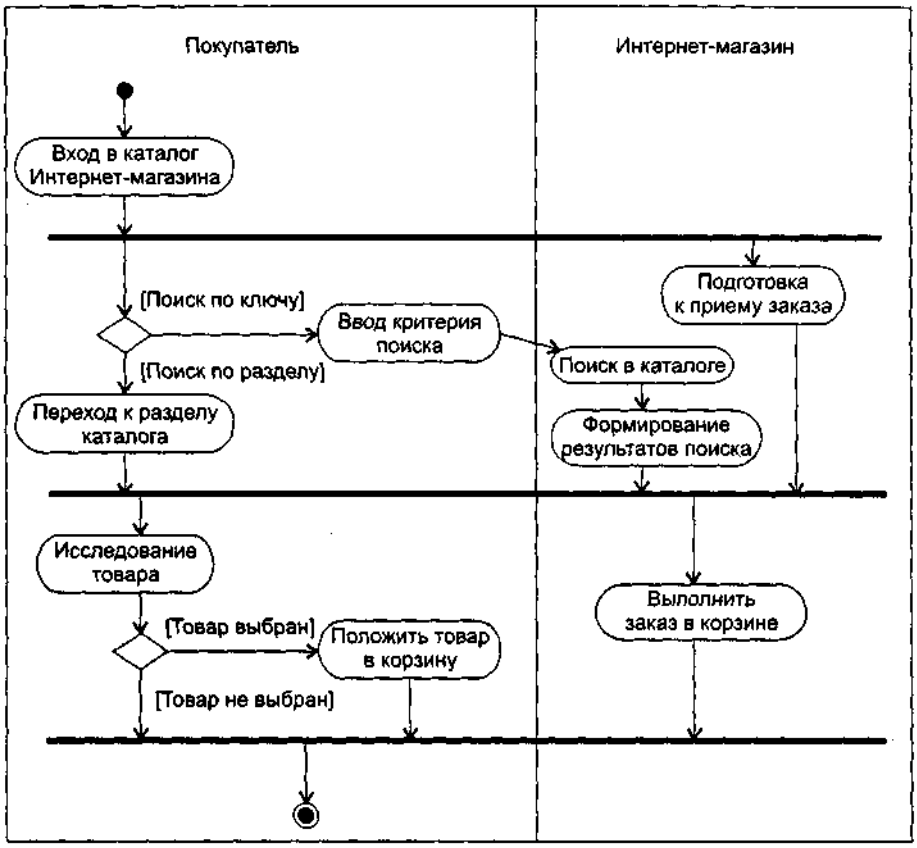


Рисунок 3 - Діаграма діяльності покупця в Інтернет-Магазині

Додатково на цій діаграмі показано дві плавальні доріжки - доріжка покупця й доріжка магазину, які розділені вертикальною лінією. Кожна доріжка має ім'я й фіксує область діяльності конкретної особи, позначаючи зону його відповідальності.

Тема №15: Основи компонентної об’єктної системи

(4 години)

Мета: Розглянути основні поняття COM. Розглянути як відбувається опис та реалізація інтерфейсу.

План:

1. Основи компонентної об'єктної моделі;
2. Організація інтерфейсу COM;
3. Ідентифікація інтерфейсу;
4. Опис інтерфейсу;

5. Реалізація інтерфейсу;

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Яке призначення COM? Які переваги дає використання COM?
2. Чим COC- Об'єкт відрізняється від звичайного об'єкта?
3. Що повинен мати клієнт для використання операції COC-Об'єкта?
4. Як ідентифікується COM- Інтерфейс?
5. Як описується COM- Інтерфейс?
6. Як реалізується COM- Інтерфейс?

Компонентна об'єктна модель (COM) - фундамент компонентно-орієнтованих засобів для всього сімейства операційних систем Windows. Розгляд цієї моделі є ілюстрацією комплексного підходу до створення й використанню компонентного програмного забезпечення.

COM визначає стандартний механізм, за допомогою якого одна частина програмного забезпечення надає свої послуги іншої частини. Загальна архітектура надання послуг у бібліотеках, додатках, системному й мережному програмному забезпеченні дозволяє COM змінити підхід до створення програм.

COM установлює поняття й правила, необхідні для визначення об'єктів і інтерфейсів; крім того, до її складу входять програми, що реалізують ключові функції.

У COM будь-яка частина ПЗ реалізує свої послуги за допомогою об'єктів COM. Кожний об'єкт COM підтримує кілька інтерфейсів. Клієнти можуть одержати доступ до послуг об'єкта COM тільки через виклики операцій його інтерфейсів - у них немає безпосереднього доступу до даних об'єкта.

Представимо об'єкт COM з інтерфейсом РаботаСФайлами. Нехай у цей інтерфейс входять операції ОткрытьФайл, ЗаписатьФайл і ЗакрытьФайл. Якщо

розроблювач захоче ввести в об'єкт COM підтримку перетворення форматів, то об'єкту буде потрібно ще один інтерфейс ПреобразованиеФорматов, можливо, з єдиною операцією ПреобразоватьФормат. Операції кожного з інтерфейсів спільно надають зв'язані один з одним послуги: або роботу з файлами, або перетворення їх форматів.

Як показано на мал. 1, об'єкт COM завжди реалізується усередині деякого сервера. Сервер може бути або бібліотекою, що динамічно підключається (DLL), подгружаемой під час роботи додатка, або окремим самостійним процесом (EXE). Відзначимо, що тут ми не будемо застосовувати графіку мови UML, а скористаємося прийнятими в COM позначеннями.

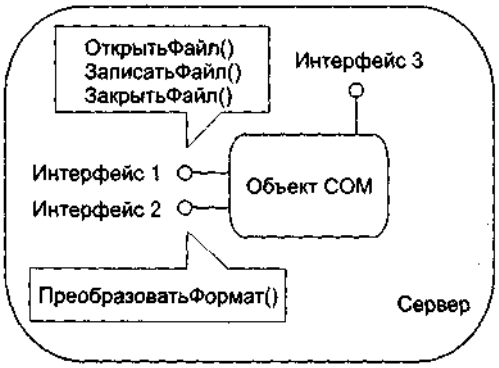


Рисунок 1 - Організація об'єкта COM

Для виклику операції інтерфейсу клієнт об'єкта COM повинен одержати покажчик на його інтерфейс. Клієнтові потрібен окремий покажчик для кожного інтерфейсу, операції якого він має намір викликати. Наприклад, як показано на мал. 24.2, клієнтові нашого об'єкта COM потрібний один покажчик інтерфейсу для виклику операцій інтерфейсу РаботаСФайлами, а іншої - для виклику операцій інтерфейсу ПреобразованиеФорматов.

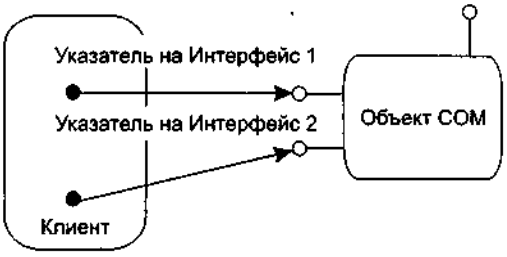


Рисунок 2 - Доступ клієнта до інтерфейсів об'єкта COM

Одержавши покажчик на потрібний інтерфейс, клієнт може використовувати послуги об'єкта, викликаючи операції цього інтерфейсу. З погляду програміста, виклик операції аналогічний виклику локальної процедури або функції. Але код, що насправді виконується при цьому, може бути частиною або бібліотеки, або окремого процесу, або операційної системи (він навіть може розташовуватися на іншому комп'ютері).

Завдяки COM клієнтам немає потреби враховувати ці відмінності - доступ до всього здійснюється одноманітно. Інакше кажучи, у COM для доступу до послуг, надаваних будь-якими типами ПО, використовується одна загальна модель.

Організація інтерфейсу COM

Кожний інтерфейс об'єкта COM - контракт між цим об'єктом і його клієнтами. Вони зобов'язуються: об'єкт - підтримувати операції інтерфейсу в точній відповідності з його визначеннями, а клієнт - коректно викликати операції. Для забезпечення контракту треба задати:

- ідентифікацію кожного інтерфейсу;
- опис операцій інтерфейсу;
- реалізацію інтерфейсу.

Ідентифікація інтерфейсу

У кожного інтерфейсу COM два імені. Простої, символьне ім'я призначене для людей, воно не унікально (допускається, щоб це ім'я було однаковим у двох інтерфейсів). Інше, складне ім'я призначене для використання програмами. Програмне ім'я унікальне, це дозволяє точно ідентифікувати інтерфейс.

Прийняте, щоб символьні імена COM- Інтерфейсів починалися з букви Й (від Interface). Наприклад, згаданий нами інтерфейс для роботи з файлами повинен називатися ІработаСФайлами, а інтерфейс перетворення їх форматів - ІпреобразованиеФорматов.

Програмне ім'я будь-якого інтерфейсу утворюється за допомогою глобально унікального ідентифікатора (globally unique identifier - GUID). GUID інтерфейсу вважається ідентифікатором інтерфейсу (interface identifier - IID). GUID - це 16-байтовая величина (128-бітове число), генерируемая автоматично.

Унікальність у часі досягається за рахунок включення в кожний GUID мітки часу, покажчика моменту створення. Унікальність у просторі забезпечується цифровими параметрами комп'ютера, який використовувався для генерації GUID.

Опис інтерфейсу

Для визначення інтерфейсів застосовують спеціальна мова - мова опису інтерфейсів (Interface Definition Language - IDL). Наприклад, Idl- Описание інтерфейсу для роботи з файлами 1Работасфайлами має вигляд

```
[ object.
uuid(E7CDODOO-1827-11CF-9946-444553540000) ]
interface Iработасфайлами: Iunknown {
import "unknown.idl"
HRESULT Открытьфайп ([in] OLECHAR ім'я [31]);
HRESULT Записатьфайл ([in] OLECHAR ім'я [31]);
HRESULT Закрытьфайл ([in] OLECHAR ім'я [31]);
}
```

Опис інтерфейсу починається зі слова object, що відзначає, що будуть використовуватися розширення, додані COM до оригінального IDL. Далі записується програмне ім'я (IID інтерфейсу), воно починається із ключового слова uuid (Universal Unique Identifier - універсально унікальний ідентифікатор). UUID є синонімом терміна GUID.

У третьому рядку записується ім'я інтерфейсу - Работасфайлами, за ним - двокрапка й ім'я іншого інтерфейсу - Iunknown. Такий запис означає, що інтерфейс Работасфайлами успадковує всі операції, певні для інтерфейсу Iunknown, тобто клієнт, у якого є покажчик на інтерфейс Iработасфайлами, може викликати й операції інтерфейсу Iunknown. Iunknown є головним інтерфейсом для COM, усі інші інтерфейси - його спадкоємці.

У четвертому рядку вказується оператор import. Оскільки даний інтерфейс успадковує від Iunknown, може знадобитися Idl- Описание для Iunknown. Аргумент оператора import указує, у якому файлі перебуває потрібний опис.

Нижче в описі інтерфейсу приводяться імена й параметри трьох операцій - Открытьфайл, Записатьфайл і Закрытьфайл. Усі вони повертають величину типу

HRESULT, що вказує коректність обробки виклику. Для кожного параметра в IDL приводиться напрямок передачі (у даному прикладі in), тип і назва.

Уважається, що такого опису досить для висновку контракту між об'єктом COM і його клієнтом.

За правилами COM забороняється будь-яка зміну інтерфейсу (після його публікації). Для реалізації змін потрібно вводити новий інтерфейс. Такий інтерфейс може бути спадкоємцем старого інтерфейсу, але відмінний від нього й має інше унікальне ім'я.

Значення правила заборони на зміну інтерфейсу важко переоцінити. Воно - застава стабільності роботи в середовищі, де безліч клієнтів взаємодіє з безліччю COM- Об'єктів і де незалежна модифікація COM- Об'єктів - звичайна справа. Саме це правило дозволяє клієнтам старих версій не постраждати при введенні нових версій COM- Об'єктів. Нова версія зобов'язано підтримувати й старий COM-Інтерфейс.

Реалізація інтерфейсу

COM задає стандартний двійковий формат, який повинен реалізувати кожний COM- Об'єкт і для кожного інтерфейсу. Стандарт гарантує, що будь-який клієнт може викликати операції будь-якого об'єкта, причому незалежно від мов програмування, на яких написані клієнт і об'єкт.

Структуру інтерфейсу ІработаСФайлами, відповідну до двійкового формату, "пояснює мал. 3.

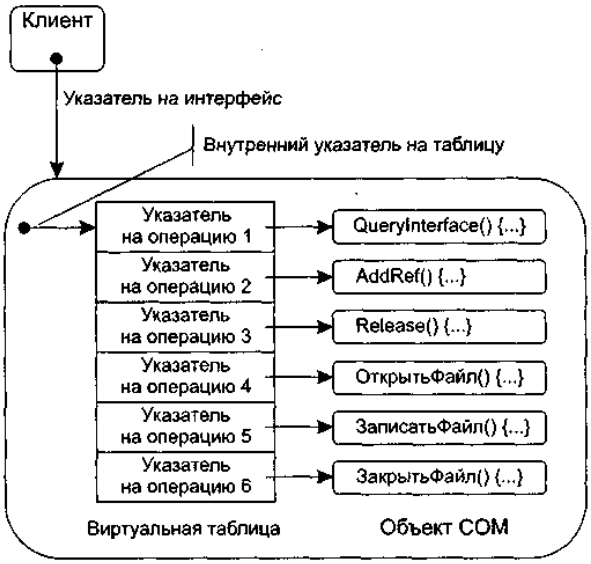


Рисунок 3 - Внутрішня структура інтерфейсу ІработаСФайлами

Зовнішній покажчик на інтерфейс (покажчик клієнта) посилається на внутрішній покажчик об'єкта COM. Внутрішній покажчик - це адреса віртуальної таблиці. Віртуальна таблиця містить покажчики на всі операції інтерфейсу.

Перші три елементи віртуальної таблиці є покажчиками на операції, успадковані від інтерфейсу Iunknown. Видне, що на власні операції інтерфейсу ІработаСФайлами вказують 4-, 5- і 6- й елементи віртуальної таблиці. Така ситуація типова для будь-якого COM- Інтерфейсу.

Обробка клієнтського виклику виконується в наступному порядку:

- за допомогою покажчика на віртуальну таблицю извлекается покажчик на необхідну операцію інтерфейсу;
- покажчик на операцію забезпечує доступ до її реалізації;
- виконання коду операції забезпечує необхідну послугу

Тема №16: Компонентні об’єктні системи

(6 години)

Мета: Ознайомитись з основами компонентної об’єктної моділ, розглянути організацію інтерфейсу COM та сервіси COM – об’єктів. Також розглянути роботу з COM – об’єктами: створення, повторне використання, маршалинг.

План:

1. Основи компонентної об’єктної моделі:
    a. Організація інтерфейсу COM;
    b. Unknown - базовий інтерфейс COM;
    c. Сервери COM-Об'єктів
2. Робота із COM-Об'єктами:
    a. Створення COM- об’єктів;
    b. Повторне використання COM – об’єктів;
    c. Маршалинг;

Домашнє завдання:

Студентам необхідно:

4. Ознайомитись з матеріалом теми;

5. Зробити конспект матеріалу теми;
6. Відповісти на контрольні питання.

Контрольні питання:

1. Яке призначення COM? Які переваги дає використання COM?
2. Чим Сом- Об'єкт відрізняється від звичайного об'єкта?
3. Що повинен мати клієнт для використання операції Сом- Об'єкта?
4. Як ідентифікується Сом- Інтерфейс?
5. Як описується Сом- Інтерфейс?
6. Як реалізується Сом- Інтерфейс?
7. Чого не можна робити із Сом- Інтерфейсом? Обґрунтуйте відповідь.
8. Поясните призначення й застосування операції Querylnterface.
9. Поясните призначення й застосування операцій Addref і Release.
10. Що таке сервер Сом- Об'єкта і які типи серверів ви знаєте?
11. У чому призначення бібліотеки COM?
12. Як створюється одиночний Сом- Об'єкт?
13. Як створюються трохи Сом- Об'єктів того самого класу?
14. Як забезпечити використання нового Сом- Класу старими клієнтами?
15. У чому полягають особливості повторного використання Сом- Об'єктів?
16. Які вимоги пред'являє агрегація до внутрішнього Сом- Об'єкту?
17. Що таке маршалинг і демаршалинг?

Компонентна об'єктна модель (COM) - фундамент компонентно-орієнтованих засобів для всього сімейства операційних систем Windows. Розгляд цієї моделі є ілюстрацією комплексного підходу до створення й використанню компонентного програмного забезпечення.

COM визначає стандартний механізм, за допомогою якого одна частина програмного забезпечення надає свої послуги іншої частини. Загальна архітектура надання послуг у бібліотеках, додатках, системному й мережному програмному забезпеченні дозволяє COM змінити підхід до створення програм.

COM установлює поняття й правила, необхідні для визначення об'єктів і інтерфейсів; крім того, до її складу входять програми, що реалізують ключові функції.

У COM будь-яка частина ПО реалізує свої послуги за допомогою об'єктів COM. Кожний об'єкт COM підтримує кілька інтерфейсів. Клієнти можуть одержати доступ до послуг об'єкта COM тільки через виклики операцій його інтерфейсів - у них немає безпосереднього доступу до даних об'єкта.

Представимо об'єкт COM з інтерфейсом Работасфайлами. Нехай у цей інтерфейс входять операції Открытьфайл, Записатьфайл і Закрытьфайл. Якщо розроблювач захоче ввести в об'єкт COM підтримку перетворення форматів, то об'єкту буде потрібно ще один інтерфейс Преобразованиеформатов, можливо, з єдиною операцією Преобразоватьформат. Операції кожного з інтерфейсів спільно надають зв'язані один з одним послуги: або роботу з файлами, або перетворення їх форматів.

Як показано на мал. 1, об'єкт COM завжди реалізується усередині деякого сервера. Сервер може бути або бібліотекою, що динамічно підключається (DLL), подгружаемой під час роботи додатка, або окремим самостійним процесом (EXE). Відзначимо, що тут ми не будемо застосовувати графіку мови UML, а скористаємося прийнятими в COM позначеннями.

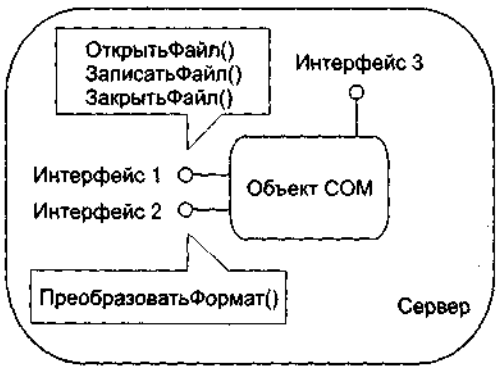


Рисунок 1 - Організація об'єкта COM

Для виклику операції інтерейсу клієнт об'єкта COM повинен одержати покажчик на його інтерфейс. Клієнтові потрібен окремий покажчик для кожного інтерфейсу, операції якого він має намір викликати. Наприклад, як показано на мал. 2, клієнтові нашого об'єкта COM потрібний один покажчик інтерфейсу для

виклику операцій інтерфейсу Работасфайлами, а іншої - для виклику операцій інтерфейсу Преобразованиеформатов.

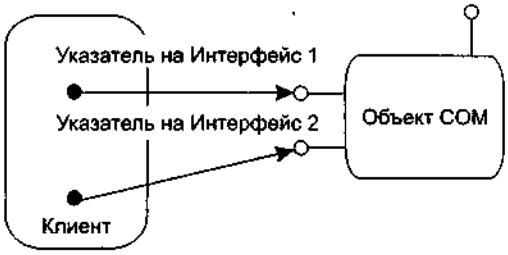


Рисунок 2 - Доступ клієнта до інтерфейсів об'єкта COM

Одержавши покажчик на потрібний інтерфейс, клієнт може використовувати послуги об'єкта, викликаючи операції цього інтерфейсу. З погляду програміста, виклик операції аналогічний виклику локальної процедури або функції. Але код, що насправді виконується при цьому, може бути частиною або бібліотеки, або окремого процесу, або операційної системи (він навіть може розташовуватися на іншому комп'ютері).

Завдяки COM клієнтам немає потреби враховувати ці відмінності - доступ до всього здійснюється одноманітно. Інакше кажучи, у COM для доступу до послуг, надаваних будь-якими типами ПО, використовується одна загальна модель.

Організація інтерфейсу COM

Кожний інтерфейс об'єкта COM - контракт між цим об'єктом і його клієнтами. Вони зобов'язуються: об'єкт - підтримувати операції інтерфейсу в точній відповідності з його визначеннями, а клієнт - коректно викликати операції. Для забезпечення контракту треба задати:

- ідентифікацію кожного інтерфейсу;
- опис операцій інтерфейсу;
- реалізацію інтерфейсу.

Ідентифікація інтерфейсу

У кожного інтерфейсу COM два імені. Простої, символьне ім'я призначене для людей, воно не унікально (допускається, щоб це ім'я було однаковим у двох інтерфейсів). Інше, складне ім'я призначене для використання програмами. Програмне ім'я унікальне, це дозволяє точно ідентифікувати інтерфейс.

Прийняте, щоб символьні імена Сом- Інтерфейсів починалися з букви Й (від Interface). Наприклад, згаданий нами інтерфейс для роботи з файлами повинен називатися Іработасфайлами, а інтерфейс перетворення їх форматів - Іпреобразованиеформатов.

Програмне ім'я будь-якого інтерфейсу утворюється за допомогою глобально унікального ідентифікатора (globally unique identifier - GUID). GUID інтерфейсу вважається ідентифікатором інтерфейсу (interface identifier - IID). GUID - це 16- байтовая величина (128-бітове число), генерируемая автоматично.

Унікальність у часі досягається за рахунок включення в кожний GUID мітки часу, покажчика моменту створення. Унікальність у просторі забезпечується цифровими параметрами комп'ютера, який використовувався для генерації GUID.

Опис інтерфейсу

Для визначення інтерфейсів застосовують спеціальна мова - мова опису інтерфейсів (Interface Definition Language - IDL). Наприклад, Idl- Описание інтерфейсу для роботи з файлами 1Работасфайлами має вигляд

```
[ object.
uuid(E7CDODOO-1827-11CF-9946-444553540000) ]
interface Іработасфайлами: Iunknown {
import "unknown.idl"
HRESULT Открытьфайп ([in] OLECHAR ім'я [31]);
HRESULT Записатьфайл ([in] OLECHAR ім'я [31]);
HRESULT Закрытьфайл ([in] OLECHAR ім'я [31]);
}
```

Опис інтерфейсу починається зі слова object, що відзначає, що будуть використовуватися розширення, додані COM до оригінального IDL. Далі записується програмне ім'я (IID інтерфейсу), воно починається із ключового слова uuid (Universal Unique Identifier - універсально унікальний ідентифікатор). UUID є синонімом терміна GUID.

У третьому рядку записується ім'я інтерфейсу - Работасфайлами, за ним - двокрапка й ім'я іншого інтерфейсу - Iunknown. Такий запис означає, що інтерфейс Работасфайлами успадковує всі операції, певні для інтерфейсу Iunknown, тобто

клієнт, у якого є покажчик на інтерфейс Іработасфайлами, може викликати й операції інтерфейсу Iunknown. Iunknown є головним інтерфейсом для COM, усі інші інтерфейси - його спадкоємці.

У четвертому рядку вказується оператор import. Оскільки даний інтерфейс успадковує від Iunknown, може знадобитися Idl- Описание для Iunknown. Аргумент оператора import указує, у якому файлі перебуває потрібний опис.

Нижче в описі інтерфейсу приводяться імена й параметри трьох операцій - Открытьфайл, Записатьфайл і Закрытьфайл. Усі вони повертають величину типу HRESULT, що вказує коректність обробки виклику. Для кожного параметра в IDL приводиться напрямок передачі (у даному прикладі in), тип і назва.

Уважається, що такого опису досить для висновку контракту між об'єктом COM і його клієнтом.

За правилами COM забороняється будь-яка зміну інтерфейсу (після його публікації). Для реалізації змін потрібно вводити новий інтерфейс. Такий інтерфейс може бути спадкоємцем старого інтерфейсу, але відмінний від нього й має інше унікальне ім'я.

Значення правила заборони на зміну інтерфейсу важко переоцінити. Воно - застава стабільності роботи в середовищі, де безліч клієнтів взаємодіє з безліччю Сом- Об'єктів і де незалежна модифікація Сом- Об'єктів - звичайна справа. Саме це правило дозволяє клієнтам старих версій не постраждати при введенні нових версій Сом- Об'єктів. Нова версія зобов'язано підтримувати й старий Сом- Інтерфейс.

Реалізація інтерфейсу

COM задає стандартний двійковий формат, який повинен реалізувати кожний Сом- Об'єкт і для кожного інтерфейсу. Стандарт гарантує, що будь-який клієнт може викликати операції будь-якого об'єкта, причому незалежно від мов програмування, на яких написані клієнт і об'єкт.

Структуру інтерфейсу Іработасфайлами, відповідну до двійкового формату, "пояснює мал. 13.3.

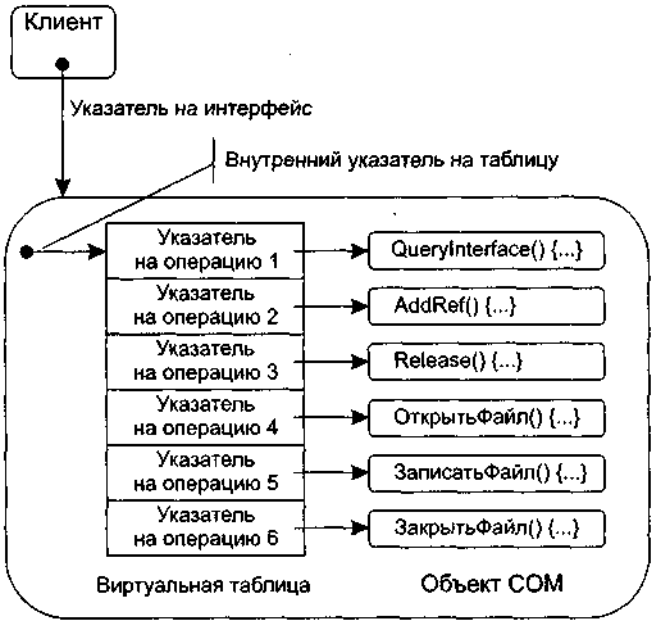


Рисунок 3 - Внутрішня структура інтерфейсу Іработасфайлами

Зовнішній покажчик на інтерфейс (покажчик клієнта) посилається на внутрішній покажчик об'єкта COM. Внутрішній покажчик - це адреса віртуальної таблиці. Віртуальна таблиця містить покажчики на всі операції інтерфейсу.

Перші три елементи віртуальної таблиці є покажчиками на операції, успадковані від інтерфейсу Iunknown. Видне, що на власні операції інтерфейсу Іработасфайлами вказують 4-, 5- і 6- й елементи віртуальної таблиці. Така ситуація типова для будь-якого Сом- Інтерфейсу.

Обробка клієнтського виклику виконується в наступному порядку:

- за допомогою покажчика на віртуальну таблицю извлекается покажчик на необхідну операцію інтерфейсу;
- покажчик на операцію забезпечує доступ до її реалізації;
- виконання коду операції забезпечує необхідну послугу.

Unknown - базовий інтерфейс COM

Інтерфейс lunknown забезпечує мінімальне "спорядження" кожного об'єкта COM. Він містить три операції й надає будь-якому об'єкту COM дві функціональні можливості:

- операція Querylnterface() дозволяє клієнтові одержати покажчик на будь-який інтерфейс об'єкта (з іншого покажчика інтерфейсу);

 операції Addref() і Release() забезпечують механізм керування часом життя об'єкта.

Свій перший покажчик на інтерфейс об'єкта клієнт одержує при створенні об'єкта COM. Порядок одержання інших покажчиків на інтерфейси (для виклику їх операцій) пояснює мал. 4, де розписано три кроки роботи. У якості параметра операції Querylnterface задається ідентифікатор необхідного інтерфейсу (IID). Якщо необхідний інтерфейс відсутній, операція повертає значення NULL.

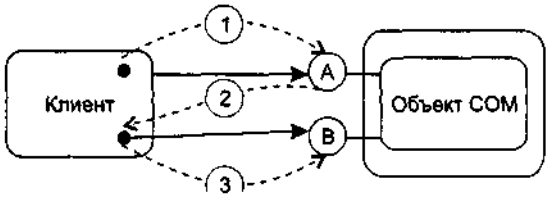


Рисунок 4 - Одержання покажчика на інтерфейс за допомогою Querylnterface: 1 - за допомогою покажчика на інтерфейс А клієнт запитує покажчик на інтерфейс В, викликаючи Querylnterface (IID_B); 2 - об'єкт повертає покажчик на інтерфейс В; 3 - тепер клієнт може викликати операції з інтерфейсу В

Має сенс відзначити й друга важлива гідність операції Querylnterface. У комбінації з вимогою незмінності Сом- Інтерфейсів вона дозволяє "убити двох зайців":

 розбудовувати компоненти;

 забезпечувати стабільність клієнтів, що використовують компоненти.

Пояснимо це твердження. За законами Сом- Етики новий Сом- Об'єкт повинен нести в собі й старий Сом- Інтерфейс, а операція Querylnterface завжди забезпечить доступ до нього.

Ну а тепер обговоримо правила життя, а точніше смерті Сом- Об'єкта. У багатоликій Сом- Середовищу тягар відповідальності за розв'язок питання про финализации повинне лежати як на клієнтові, так і на Сом- Об'єкті. Можна сказати, що фірма Microsoft (творець цієї моделі) розробила самурайський кодекс поведінки Сом- Об'єкта - він повинен сам себе знищити. Виникає питання - коли? Коли він перестане бути потрібним усім своїм клієнтам, коли витече пісок з годин його життя. Роль піскових годин відіграє лічильник посилань (СЧС) Сом- Об'єкта.

Правила финализации Сом- Об'єкта дуже прості:

- при видачі клієнтові покажчика на інтерфейс виконується СЧС+1;
- при виклику операції Addref виконується СЧС+1;
- при виклику операції Release виконується СЧС-1;
- при СЧС=0 об'єкт знищує себе.

Звичайно, клієнт повинен допомагати гідному харакірі об'єкта- самурая:

- при одержанні від іншого клієнта покажчика на інтерфейс Сом-Об'єкта він повинен викликати в цьому об'єкті операцію Addref;
- наприкінці роботи з об'єктом він зобов'язано викликати його операцію Release.

Сервери COM- Об'єктів

Кожний COM- Об'єкт існує усередині конкретного сервера. Цей сервер містить програмний код реалізації операцій, а також дані активного Сом- Об'єкта. Один сервер може забезпечувати кілька об'єктів і навіть трохи Сом- Класів. Як показано на мал. 13.5, використовуються три типи серверів:

- Сервер "у процесі" (in-process) - об'єкти перебувають у бібліотеці, що динамічно підключається, і, отже, виконуються в тому ж процесі, що й клієнт;
- Локальний сервер (out-process) - об'єкти перебувають в окремому процесі, що виконується на тому ж комп'ютері, що й клієнт;
- Вилучений сервер - об'єкти перебувають в DLL або в окремому процесі, які розташовані на вилученому від клієнта комп'ютері.

З погляду логіки, клієнтові байдуже, у сервері якого типу перебуває Сом-Об'єкт - створення об'єкта, одержання покажчика на його інтерфейси, виклик його операцій і финализация виконуються завжди однаково. Хоча тимчасові витрати на організацію взаємодії в кожному із трьох випадків, звичайно, відрізняються друг від друга.

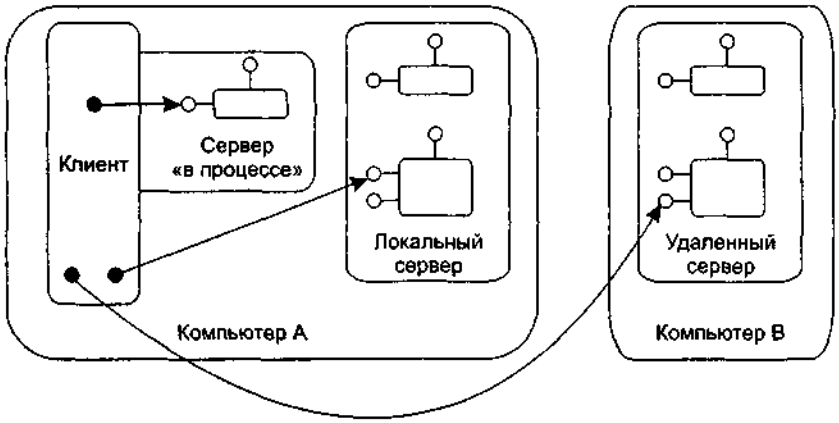


Рисунок 5 - Різні сервери Сом- Об'єктів

Переваги COM

У якості коротких висновків відзначимо основні переваги COM.

1. COM забезпечує зручний спосіб фіксації послуг, надаваних різними фрагментами ПО.

2. Загальний підхід до створення всіх типів програмних послуг у COM спрощує проблеми розробки.

3. COM байдужний мову програмування, на якій пишуться Сом- Об'єкти й клієнти.

4. COM забезпечує ефективне управління зміною програм - заміну поточної версії компонента на нову версію з додатковими можливостями.

Робота із COM- Об'єктами

При роботі із Сом- Об'єктами доводиться їх створювати, повторно використовувати, розміщати в інших процесах, описувати в бібліотеці операційної системи. Розглянемо кожний із цих питань.

Створення COM- Об'єктів

Створення COM- Об'єкта базується на використанні функцій бібліотеки COM. Бібліотека COM:

- містить функції, що пропонують базові послуги об'єктам і їх клієнтам;
- надає клієнтам можливість запуску серверів Сом- Об'єктів.

Доступ до послуг бібліотеки COM виконується за допомогою викликів звичайних функцій. Найчастіше імена функцій бібліотеки COM починаються із префікса "З". Наприклад, у бібліотеці є функція Cocreateinstance.

Для створення Сом- Об'єкта клієнт викликає функцію бібліотеки COM Cocreateinstance. У якості параметрів цієї функції посилають ідентифікатор класу об'єкта CLSID і IID інтерфейсу, підтримуваного об'єктом. За допомогою CLSID бібліотека шукає сервер класу (це робить диспетчер керування сервісами SCM - Service Control Manager). Пошук проводиться в системному реєстрі (Registry), що відображає CLSID на адресу коду, що виконується, сервера. У системному реєстрі повинні бути зареєстровані класи всіх Сом- Об'єктів.

Закінчивши пошук, бібліотека COM запускає сервер класу. У результаті створюється неинициализированный Сом- Об'єкт, тобто об'єкт, дані якого не визначені. Описаний процес ілюструє мал. 6.

Як правило, після одержання покажчика на створений Сом- Об'єкт клієнт пропонує об'єкту самоинициализироваться, тобто завантажити себе конкретними значеннями даних. Цю процедуру забезпечують стандартні Сом- Інтерфейси Ipersistfile, Ipersiststorage і Ipersiststream.

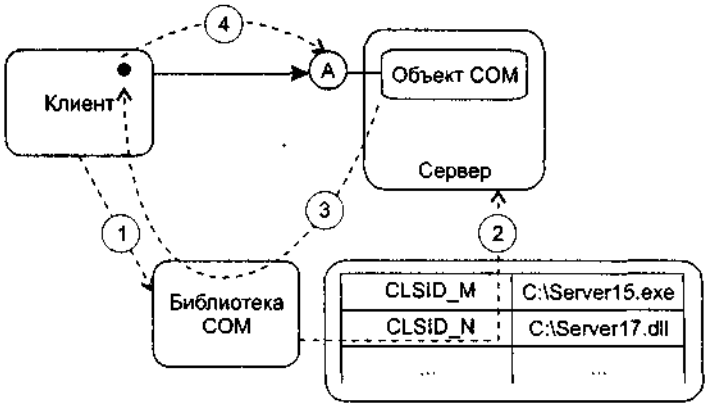


Рисунок 6 - Створення одиночного Сом- Об'єкта: 1 - клієнт викликає Cocreatelnstance (CLSID M, IID A); 2 - бібліотека COM знаходить сервер і запускає його; 3 - бібліотека COM повертає покажчик на інтерфейс A; 4 - тепер клієнт може викликати операції Сом- Об'єкта

Параметри функції Cocreateinstance, використовуваної клієнтом, дозволяють також задати тип сервера, який потрібно запустити (наприклад, "у процесі" або локальний).

У більш загальному випадку клієнт може створити трохи COM- Об'єктів того самого класу. Для цього клієнт використовує фабрику класу (class factory) - Сом- Об'єкт, здатний генерувати об'єкти одного конкретного класу.

Фабрика класу підтримує інтерфейс Iclassfactory, що включає дві операції. Операція Createlnstance створює COM- Об'єкт - екземпляр конкретного класу, має параметр - ідентифікатор інтерфейсу, покажчик на який треба повернути клієнтові. Операція Lockserver дозволяє зберігати сервер фабрики завантаженим на згадку.

Клієнт викликає фабрику за допомогою функції бібліотеки COM Cogetclassobject:

Cogetclassobject (<CLSID створюваного обьекта>, < IID інтерфейсу Iclassfactory>)

У якості третього параметра функції можна задати тип сервера, що запускається.

Бібліотека COM запускає фабрику класу й повертає покажчик на інтерфейс Iclassfactory цієї фабрики. Подальший порядок роботи за допомогою фабрики ілюструє мал. 7.

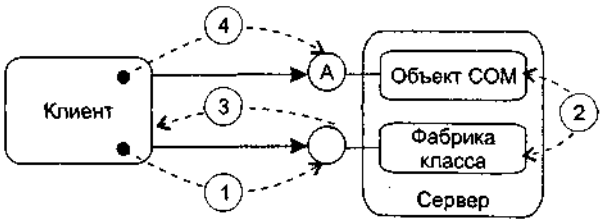


Рисунок 7 - Створення Сом- Об'єкта за допомогою фабрики класу: 1 - клієнт викликає Iclassfactory :: Createlnstance (IID A); 2 - фабрика класу створює Сом- Об'єкт і одержує покажчик на його інтерфейс; 3 - фабрика класу повертає покажчик на інтерфейс А Сом-Об'єкта; 4 - тепер клієнт може викликати операції Сом- Об'єкта

Клієнт викликає операцію Iclassfactory::Createinstance фабрики, у якості параметра якої задає ідентифікатор необхідного інтерфейсу об'єкта (IID). У відповідь фабрика класу створює Сом- Об'єкт і повертає покажчик на заданий інтерфейс. Тепер клієнт застосовує повернутий покажчик для виклику операцій Сом- Об'єкта.

Дуже часто виникає наступна ситуація - існуючий Сом- Клас замінили інш, що підтримують як старі, так і додаткові інтерфейси, що й мають інший CLSID. З'являється завдання - забезпечити використання нового Сом- Класу старими клієнтами. Звичайним розв'язком є запис до системного реєстру відповідності між

старим і новим CLSID. Запис виконується за допомогою бібліотечної функції Cotreatasclass:

Cotreatasclass (<старий CLSID>, <новий CLSID>).

Повторне використання COM- Об'єктів

Відомо, що основним засобом повторного використання існуючого коду є спадкування реалізації (новий клас успадковує реалізацію операцій існуючого класу). COM не підтримує цей засіб. Причина - у типовий COM- Середовищу базові об'єкти й об'єкти- спадкоємці створюються, випускаються й обновляються незалежно. У цих умовах зміни базових об'єктів можуть викликати непередбачені наслідки в об'єктах- спадкоємцях їх реалізації. COM пропонує інші засоби повторного використання - включення й агрегування.

Застосовуються наступні терміни:

- внутрішній об'єкт - це базовий об'єкт;
- зовнішній об'єкт - це об'єкт, що повторно використовує послуги внутрішнього об'єкта.

При включенні (делегуванні) зовнішній об'єкт є звичайним клієнтом внутрішнього об'єкта. Як показано на мал. 8, коли клієнт викликає операцію зовнішнього об'єкта, ця операція, у свою чергу, викликає операцію внутрішнього об'єкта.

Рисунок 8 - Повторне використання Сом- Об'єкта за допомогою включення

При цьому внутрішній об'єкт нічого не зауважує.

Гідність включення - простота. Недолік - низька ефективність при довгому ланцюжку ", щоделегують" об'єктів.

Недолік включення усуває агрегування. Воно дозволяє зовнішньому об'єкту обманювати клієнтів - представляти в якості власних інтерфейси, реалізовані внутрішнім об'єктом. Як показано на мал. 9, коли клієнт запитує в зовнішнього об'єкта покажчик на такий інтерфейс, йому вертається покажчик на внутрішній,

агрегированный інтерфейс. Клієнт про агрегування нічого не знає, зате внутрішній об'єкт обов'язково повинен знати про те, що він агрегирован.

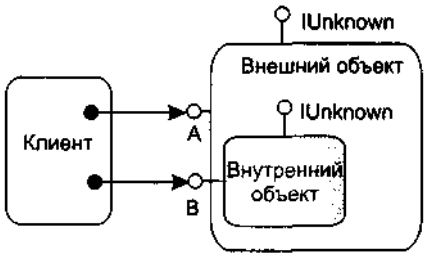


Рисунок 9 - Повторне використання Сом- Об'єкта за допомогою агрегування

Навіщо потрібно таке знання? У чому причина? Відповідь полягає в необхідності особливої реалізації внутрішнього об'єкта. Вона повинна забезпечити правильний підрахунок посилань і коректну роботу операції Querylnterface.

Представимо два практичні завдання:

- запит клієнтом у внутрішнього об'єкта (за допомогою операції Querylnterface) покажчика на інтерфейс зовнішнього об'єкта;
- зміна клієнтом лічильника посилань внутрішнього об'єкта (за допомогою операції Addref) і інформування про цей зовнішнього об'єкта.

Ясно, що при автономному й незалежному внутрішньому об'єкті їх розв'язати не можна. У противному ж випадку розв'язок елементарний - внутрішній об'єкт повинен відмовитися від власного інтерфейсу Iunknown і застосовувати тільки операції Iunknown зовнішнього об'єкта. Іншими словами, адреса власного Iunknown повинен бути заміщений адресою Iunknown агрегирующего об'єкта. Це вказується при створенні внутрішнього об'єкта (за рахунок додавання адреси як параметра операції Cocreatelnstance або операції Iclassfactory::Createinstance).

Маршалинг

Клієнт може містити пряме посилання на COM- Об'єкт тільки в одному випадку - коли COM- Об'єкт розміщений у сервері "у процесі". У випадку локального або вилученого сервера, як показано на мал. 13.10, він посилається на посередника.

Посередник - COM- Об'єкт, розміщений у клієнтському процесі, що й надає клієнтові ті ж інтерфейси, що й запитуваний об'єкт. Запит клієнтом операції через таке посилання приводить до виконання коду посередника.

Посередник ухвалює параметри, передані клієнтом, і впаковує їх для подальшого пересилання. Ця процедура називається маршалингом. Потім посередник (за допомогою засобу комунікації) надсилає запит у процес, який насправді реалізує COM- Об'єкт.

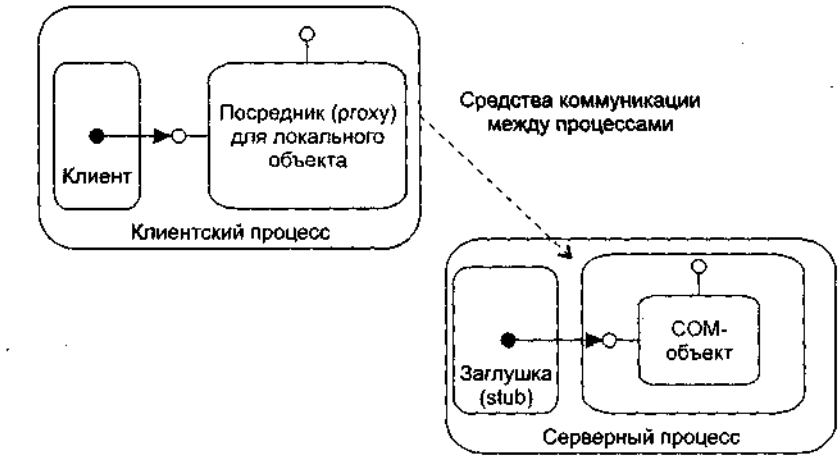


Рисунок 10 - Організація маршалинга й демаршалинга

Після прибуття в процес локального сервера запит передається заглушці. Заглушка розпаковує параметри запиту й викликає операцію COM- Об'єкта. Ця процедура називається демаршалингом. Після завершення COM- Операції результати вертаються у зворотному напрямку.

Код посередника й заглушки автоматично генерується компілятором MIDL (Microsoft IDL) по Idl- Описанию інтерфейсу.

Тема №17: Діаграма розміщення

(4 години)

Мета: Розглянути основні компоненти діаграми розміщення – вузли, а також з’єднання вузлів .

План:

1. Діаграма розміщення;
2. Вузол;
3. З’єднання;

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Які вершини й ребра утворюють діаграму розміщення?
2. Чим відрізняється вузол від компонента?
3. Де можна використовувати й де не можна використовувати екземпляри компонентів?
4. Як застосовують діаграми розміщення?

Фізична вистава програмної системи не може бути повним, якщо відсутня інформація про те, на якій платформі й на яких обчислювальних засобах вона реалізована. Звичайно, якщо розробляється проста програма, яка може виконуватися локально на комп'ютері користувача, не задіючи ніяких периферійних обладнань і ресурсів, то в цьому випадку немає необхідності в розробці додаткових діаграм. Однак при розробці корпоративних додатків ситуація представляється зовсім по-іншому.

По-перше, складні програмні системи можуть реалізовуватися в мережному варіанті на різних обчислювальних платформах і технологіях доступу до розподілених баз даних. Наявність локальної корпоративної мережі вимагає розв'язку цілого комплексу додаткових завдань по раціональному розміщенню компонентів по вузлах цієї мережі, що визначає загальну продуктивність програмної системи.

По-друге, інтеграція програмної системи з Інтернетом визначає необхідність розв'язку додаткових питань при проектуванні системи, таких як забезпечення безпеки, криптозащищенности й стійкості доступу до інформації для

корпоративних клієнтів. Ці аспекти в чималому ступені залежать від реалізації проекту у формі фізично існуючих вузлів системи, таких як сервери, робочі станції, брандмауери, канали зв'язку й сховища даних.

Нарешті, технології доступу й маніпулювання даними в рамках загальної схеми " клієнт-сервер" також вимагають розміщення більших баз даних у різних сегментах корпоративної мережі, їх резервного копіювання, архівування, кэширования для забезпечення необхідної продуктивності системи в цілому. Ці аспекти також вимагають візуальної вистави з метою специфікації програмних і технологічних особливостей реалізації розподілених архитектур.

Як було відзначено в главі 10, першої з діаграм фізичної вистави є діаграма компонентів. Другою формою фізичної вистави програмної системи є діаграма розгортання (синонім - діаграма розміщення). Вона застосовується для вистави загальної конфігурації й топології розподіленої програмної системи й містить розподіл компонентів по окремих вузлах системи. Крім того, діаграма розгортання показує наявність фізичних з'єднань - маршрутів передачі інформації між апаратними обладнаннями, задіяними в реалізації системи.

Діаграма розгортання призначена для візуалізації елементів і компонентів програми, що існують лише на етапі її виконання (runtime). При цьому представляються тільки компоненти- екземпляри програми, що є здійсненними файлами або динамічними бібліотеками. Ті компоненти, які не використовуються на етапі виконання, на діаграмі розгортання не показуються. Так, компоненти з вихідними текстами програм можуть бути присутнім тільки на діаграмі компонентів. На діаграмі розгортання вони не вказуються.

Діаграма розгортання містить графічні зображення процесорів, обладнань, процесів і зв'язків між ними. На відміну від діаграм логічної вистави, діаграма розгортання є єдиною для системи в цілому, оскільки повинна цілком відбивати особливості її реалізації. Ця діаграма, по суті, завершує процес ООАП для конкретної програмної системи і її розробка, як правило, є останнім етапом специфікації моделі.

Отже, перелічимо мети, переслідувані при розробці діаграми розгортання:

Визначити розподіл компонентів системи по її фізичних вузлах.

Показати фізичні зв'язки між усіма вузлами реалізації системи на етапі її виконання.

Виявити вузькі місця системи й реконфигурировать її топологію для досягнення необхідної продуктивності.

Для забезпечення цих вимог діаграма розгортання розробляється спільно системними аналітиками, мережними інженерами й системотехніками. Далі розглянемо окремі елементи, з яких полягають діаграми розгортання.

Вузол

Вузол (node) являє собою деякий фізично існуючий елемент системи, що володіє деяким обчислювальним ресурсом. У якості обчислювального ресурсу вузла може розглядатися наявність щонайменше деякого обсягу електронної або магнітооптичної пам'яті й/або процесора. В останній версії мови UML поняття вузла розширене й може містити в собі не тільки обчислювальні обладнання (процесори), але й інші механічні або електронні обладнання, такі як датчики, принтери, модеми, цифрові камери, сканери й маніпулятори.

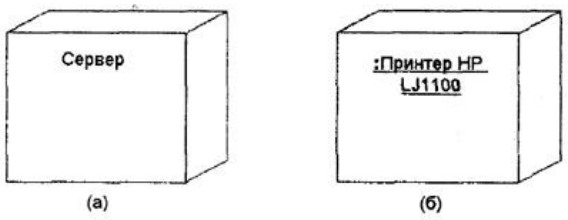


Рисунок 1 - Графічне зображення вузла на діаграмі розгортання

У першому випадку ім'я вузла записується без підкреслення й починається із заголовної букви. У другому ім'я вузла- екземпляра записується у вигляді <ім'я вузла ':' ім'я типу вузла>. Ім'я типу вузла вказує на деякий різновид вузлів, що присутствующих у моделі системи.

Наприклад, апаратна частина системи може складатися з декількох персональних комп'ютерів, кожний з яких відповідає окремому вузлу- екземпляру в моделі. Однак усі ці вузли- екземпляри ставляться до одному типу вузлів, а саме вузлу з іменем типу "Персональний комп'ютер". Так, на представленому вище малюнку (мал. 1, а) вузол з іменем "Сервер" ставиться до загального типу й ніяк не конкретизується. Другий же вузол (мал. 1, б) є анонімним вузлом- екземпляром конкретної моделі принтера.

Так само, як і на діаграмі компонентів, зображення вузлів можуть розширюватися, щоб включити деяку додаткову інформацію про специфікацію вузла. Якщо додаткова інформація ставиться до імені вузла, то вона записується під цим іменем у формі позначеного значення (мал. 14.2).

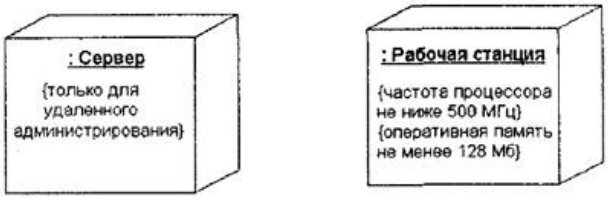


Рисунок 2 - Графічне зображення вузла- екземпляра з додатковою інформацією у формі позначеного значення

Якщо необхідно явно вказати компоненти, які розміщаються на окремому вузлі, то це можна зробити двома способами. Перший з них дозволяє розділити графічний символ вузла на дві секції горизонтальною лінією. У верхній секції записують ім'я вузла, а в нижній секції - розміщені на цьому вузлі компонента (мал. 3, а).

Другий спосіб дозволяє показувати на діаграмі розгортання вузли із вкладеними зображеннями компонентів (мал. 3, б). Важливо пам'ятати, що в якості таких вкладених компонентів можуть виступати компоненти, що тільки виконуються.

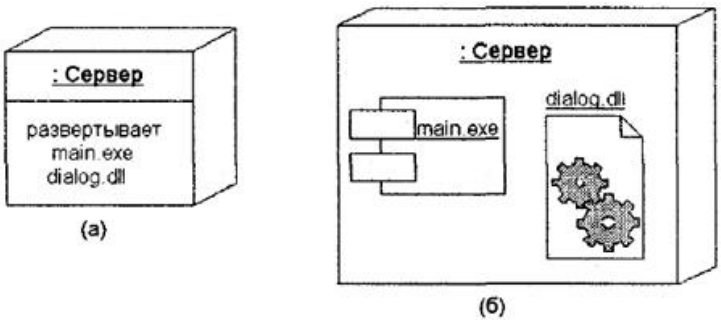


Рисунок 3 - Варіанти графічного зображення вузлів- екземплярів з розташовуваними на них компонентами

У якості доповнення до імені вузла можуть використовуватися різні стереотипи, які явно специфицируют призначення цього вузла. Хоча в мові UML стереотипи для вузлів не визначені, у літературі зустрічаються наступні їхні варіанти: "процесор", "датчик", "модем", "мережа", "консоль" і ін., які самостійно можуть бути визначені розроблювачем. Більше того, на діаграмах розгортання

допускаються спеціальні позначення для різних фізичних обладнань, графічне зображення яких проясняє призначення або виконувані обладнанням функції.

З'єднання

Крім властиво зображень вузлів на діаграмі розгортання вказуються відносини між ними. У якості відносин виступають фізичні з'єднання між вузлами й залежності між вузлами й компонентами, зображення яких теж можуть бути присутнім на діаграмах розгортання.

З'єднання є різновидом асоціації й зображуються відрізками ліній без стрілок. Наявність такої лінії вказує на необхідність організації фізичного каналу для обміну інформацією між відповідними вузлами. Характер з'єднання може бути додатково специфікований приміткою, позначеною значенням або обмеженням. Так, на представленому нижче фрагменті діаграми розгортання (мал. 4) явно визначені не тільки вимоги до швидкості передачі даних у локальній мережі за допомогою позначеного значення, але й рекомендації з технології фізичної реалізації з'єднань у формі примітки.

Рисунок 4 - Фрагмент діаграми розгортання із з'єднаннями меходу вузлами

Крім з'єднань на діаграмі розгортання можуть бути присутнім відносини залежності між вузлом і розгорнутими на ньому компонентами. Подібний спосіб є альтернативою вкладеному зображенню компонентів усередині символу вузла, що не завжди зручно, оскільки робить цей символ зайво об'ємним. Тому при великій кількості розгорнутих на,вузлі компонентів відповідну інформацію можна представити у формі відносини залежності (мал. 5).

Діаграми розгортання можуть мати більш складну структуру, що включає вкладені компоненти, інтерфейси й інші апаратні обладнання. На зображеній нижче діаграмі розгортання (мал. 6) представлений фрагмент фізичної вистави

системи вилученого обслуговування клієнтів банку. Вузлами цієї системи є вилучений термінал (вузол-тип) і сервер банку (вузол-екземпляр).

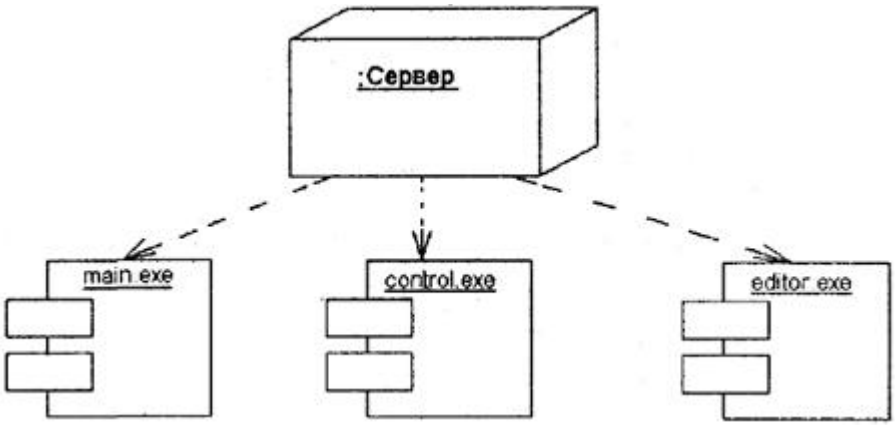


Рисунок 5 - Діаграма розгортання з відношенням залежності між вузлом і розгорнутими на ньому компонентами

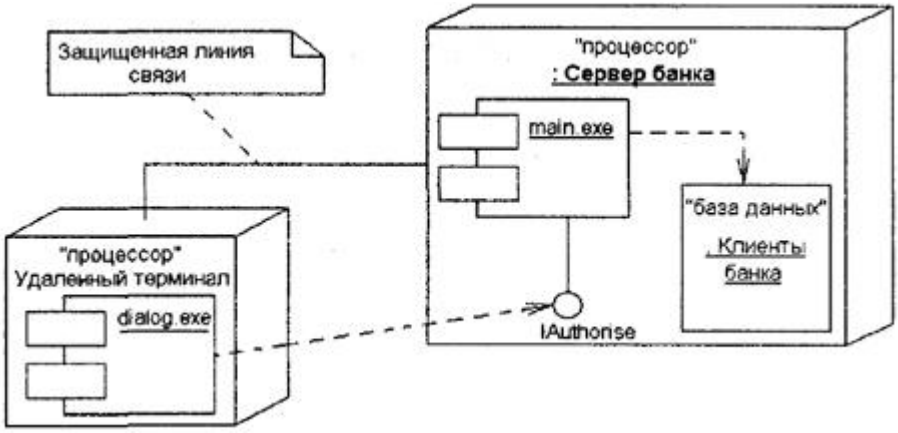


Рисунок 6 - Діаграма розгортання для системи вилученого обслуговування клієнтів банку

Тема №18: Інтеграція. Тестування інтеграції

(2 години)

Мета: Розглянути інтеграційне тестування компонентно-базового ПЗ. Основні визначення інтеграційного тестування.

План:

1. Тестування інтеграції;
2. Спадне тестування інтеграції;
3. Висхідне тестування інтеграції;

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Коли й навіщо виконується тестування інтеграції? Який етап конструювання воно перевіряє?
2. У чому ціль тестування інтеграції?
3. Які категорії помилок інтерфейсу ви знаєте?
4. У чому суть спадного тестування інтеграції?
5. Поясните кроки процесу спадної інтеграції.
6. Поясните гідності й недоліки спадної інтеграції.
7. Які категорії заглушок ви знаєте?
8. У чому суть висхідного тестування інтеграції?
9. Поясните кроки процесу висхідної інтеграції.
10. Поясните гідності й недоліки висхідної інтеграції.
11. Які категорії драйверів ви знаєте?

Тестування інтеграції підтримує складання цільної програмної системи.

Ціль складання й тестування інтеграції: побрати модулі, протестовані як елементи, і побудувати програмну структуру, необхідну проектом.

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

- втрата даних при проходженні через інтерфейс;
- відсутність у модулі необхідного посилання;
- несприятливий вплив одного модуля на іншій;
- подфункции при об'єднанні не утворюють необхідну головну функцію;
- окремі (припустимі) неточності при інтеграції виходять за припустимий рівень;
- проблеми при роботі із глобальними структурами даних.

Існує два варіанти тестування інтеграції, що підтримують процес: спадне тестування й висхідне тестування. Розглянемо кожний з них.

Спадне тестування інтеграції

У даному підході модулі поєднуються рухом зверху вниз по керуючій ієрархії, починаючи від головного керуючого модуля. Підлеглі модулі додаються в структуру або в результаті пошуку в глибину, або в результаті пошуку завширшки .

Розглянемо приклад (мал. 1). Інтеграція пошуком у глибину буде підключати всі модулі, що перебувають на головному керуючому шляху структури (по вертикалі). Вибір головного керуючого шляху почасти довільний і залежить від характеристик, обумовлених додатком. Наприклад, при виборі лівого шляху насамперед будуть підключені модулі Ml, M2, M5. Наступним підключається модуль M8 або Мб (якщо це необхідно для правильного функціонування M2). Потім будується центральний або правий керуючий шлях.

При інтеграції пошуком завширшки структура послідовно проходиться по рівнях- горизонталям. На кожному рівні підключаються модулі, безпосередньо підлеглі керуючому модулю - начальникові. У цьому випадку насамперед підключаються модулі M2, M3, M4. На наступному рівні - модулі M5, Мб і т.д..

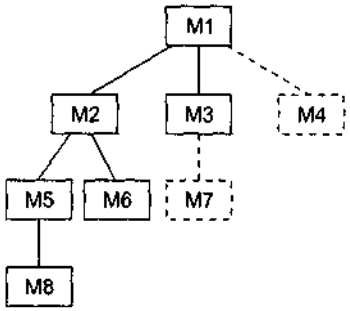


Рисунок 1 - Спадна інтеграція системи

Опишемо можливі кроки процесу спадної інтеграції.

1. Головний керуючий модуль (перебуває на вершині ієрархії) використовується як тестовий драйвер. Усі безпосередньо підлеглі йому модулі тимчасово заміщаються заглушками.

2. Одна із заглушок заміняється реальним модулем. Модуль вибирається пошуком завширшки або в глибину.

3. Після підключення кожного модуля (і установки на ньому заглушок) проводиться набір тестів, перевіряючих отриману структуру.

4. Якщо в модулі- драйвері вже немає заглушок, проводиться зміна модуля- драйвера (пошуком завширшки або в глибину).

5. Виконується повернення на крок 2 (доти , поки не буде побудована ціла структура).

Гідність спадної інтеграції: помилки в головн частині, що управляє, системи виявляються в першу чергу.

Недолік: труднощі в ситуаціях, коли для повного тестування на верхніх рівнях потрібні результати обробки з нижніх рівнів ієрархії.

Існують 3 можливості боротьби із цим недоліком:

1) відкладати деякі тести до заміщення заглушок модулями;

2) розробляти заглушки, що частково виконують функції модулів;

3) підключати модулі рухом знизу нагору.

Перша можливість викликає складності в оцінці результатів тестування.

Для реалізації другої можливості вибирається одна з наступних категорій заглушок:

- заглушка A - відображає трассируемое повідомлення;
- заглушка B - відображає минаючий параметр;
- заглушка З - повертає величину з таблиці;
- заглушка D - виконує табличний пошук по ключу (вхідному параметру) і повертає пов'язаний з ним вихідний параметр.

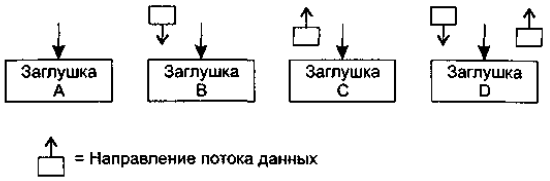


Рисунок 2 - Категорії заглушок

Очевидно, що заглушка A найбільш проста, а заглушка D найбільш складна в реалізації.

Цей підхід працездатний, але може привести до істотних витрат, тому що заглушки стають усе більш складними.

Третю можливість обговоримо окремо.

Висхідне тестування інтеграції

При висхідному тестуванні інтеграції складання й тестування системи починаються з модулів-атомів, розташовуваних на нижніх рівнях ієрархії. Модулі підключаються рухом знизу нагору. Підлеглі модулі завжди доступні, і немає необхідності в заглушках.

Розглянемо кроки методики висхідної інтеграції.

1. Модулі нижнього рівня поєднуються в кластери (групи, блоки), що виконують певну програмну подфункцию.

2. Для координації введень-висновків тестового варіанта пишеться драйвер, керуючий тестуванням кластерів.

3. Тестується кластер.

4. Драйвери віддаляються, а кластери поєднуються в структуру рухом нагору. Приклад висхідної інтеграції системи наведений на мал. 3

Модулі поєднуються в кластери 1,2,3. Кожний кластер тестується драйвером. Модулі в кластерах 1 і 2 підлеглі модулю Ma, тому драйвери D1 і D2 віддаляються й кластери підключають прямо до Ma. Аналогічно драйвер D3 віддаляється перед підключенням кластера 3 до модуля Mb. В останню чергу до модуля Mc підключаються модулі Ma й Mb.

Розглянемо різні типи драйверів:

- драйвер A - викликає підлеглий модуль;
- драйвер B - посилає елемент даних (параметр) із внутрішньої таблиці;
- драйвер З - відображає параметр із підлеглого модуля;
- драйвер D - є комбінацією драйверів У и С.

Очевидно, що драйвер A найбільш простий, а драйвер D найбільш складний у реалізації. Різні типи драйверів представлені на мал. 4.

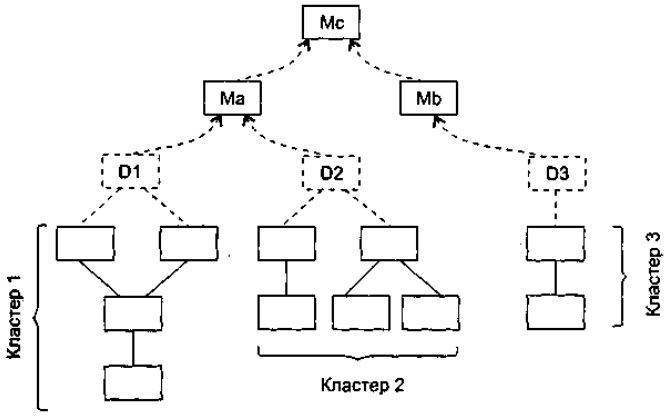


Рисунок 3 - Висхідне тестування інтеграції

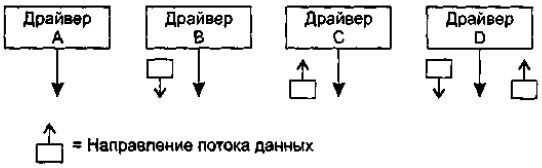


Рисунок 4 - Різні типи драйверів

У міру просування інтеграції нагору необхідність у виділенні драйверів зменшується. Як правило, у дворівневій структурі драйвери не потрібні.

Порівняння спадного й висхідного тестування інтеграції

Спадне тестування:

1) основний недолік- необхідність заглушок і пов'язані з ними труднощі тестування;

2) основна гідність - можливість раннього тестування головних керуючих функцій.

Висхідне тестування:

1) основний недолік - система не існує як об'єкт доти , поки не буде доданий останній модуль;

2) основна гідність - спрощується розробка тестових варіантів, відсутні заглушки.

Можливий комбінований підхід. У ньому для верхніх рівнів ієрархії застосовують спадну стратегію, а для нижніх рівнів - висхідну стратегію тестування.

При проведенні тестування інтеграції дуже важливо виявити критичні модулі. Ознаки критичного модуля:

1) реалізує кілька вимог до програмної системи;

2) має високий рівень керування (перебуває досить високо в програмній структурі);

3) має високу складність або схильність до помилок (як індикатор може використовуватися цикломатическая складність - її верхня розумна межа становить 10);

4) має певні вимоги до продуктивності обробки.

Критичні модулі повинні тестуватися якомога раніше . Крім того, до них повинне застосовуватися регресійне тестування (повторення вже виконаних тестів у повному або частковому обсязі

Тема 19: Основи шаблонів проєктування

(4 години)

Мета: Розглянути основні структурні паттерни проектування, ситуації в яких стає необхідність їх використувавати та розглянути приклади вирішення таких ситуацій..

План:

1. Породжуючі паттерни;
2. Структурні паттерни;
3. Паттерни поведінки;

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Що таке шаблон проектування?
2. Які бувають структурні паттерни проектування?
3. Для чого необхідні паттерни поведінки?
4. Приведіть приклад породжуючи паттернів.

Шаблони проектування програмного забезпечення (англ. software design patterns) — ефективні способи вирішення задач проектування програмного забезпечення. Шаблон не є закінченим зразком, який можна безпосередньо транслювати в програмний код. Об'єктно-орієнтований шаблон найчастіше є зразком вирішення проблеми і відображає відношення між класами та об'єктами, без вказівки на те, як буде зрештою реалізоване це відношення.

Так уже прийнято, що усі дизайн патерни поділені на три великі групи, а саме: породжуючі, структурні та поведінкові.

породжуючі патерни. Основним завданням таких патернів є спростити створення об’єктів, які необхідні аплікації.

Інколи ви працюєте із певним набором об’єктів через групу інтерфейсів. А тоді хочете створювати об’єкти тільки із іншого набору, щоб пристосувати ваш код до інших умов. Звичайно група інтерфейсів, через які ви оперуєте, залишається та ж сама. Спростити створення відповідного набору допоможе Абстрактна Фабрика. А інколи структура деякого об’єкта дуже складна і залежить від багатьох чинників. Щоб спростити створення такого об’єкту зазвичай використовують Будівельника.

А щоб зручно вибрати одну реалізацію та інстанціювати її, відштовхуючись від простої умови, можна використати Фабричний Метод. Нерідко постає завдання отримати копію уже існуючого об’єкта, або отримати можливість швидко генерувати багато подібних екземплярів. У такому випадку Прототип якраз згодиться.

Вибагливе множення об’єктів не єдине завдання, яке вам слід буде виконувати у роботі, вам часто буде потрібно робити все точно навпаки – мати один-єдиний екземпляр об’єкта і ні за яких обставин не доступатися до йому подібних. Функціональність єдиного екземпляра забезпечуються Синглтоном.

Основним завданням структурних патернів є формування найбільш підходящої структури та взаємодії між класами для виконання певних завдань. Якщо потрібно, щоб один об’єкт міг бути зрозумілим під іншим інтерфейсом, використовується Адаптер.

Якщо ви хочете розділити абстракцію та імплементацію так, що на одному боці ви матимете абстракцію, а на іншому декілька реалізацій, причому всі

доступні до модифікацій, то слід задуматися над поєднанням таких незалежних абстракції та реалізації за допомогою патерну Міст.

Якщо елемент містить собі подібні елементи, а вони в свою чергу також можуть містити елементи, то найлегше таку структуру реалізувати за допомогою Компонувальника.

Для швидкої та динамічної можливості розширення існуючої функціональності, без зміни її носителя, можна скористатися Декоратором.

Якщо ваша система використовує багато об'єктів, що мають спільні дані, то такі дані можна винести та зробити загальнодоступними для економії пам'яті за допомогою патерну Легковаговик.

Якщо відсутня можливість працювати із об'єктом напряму, використайте Проксі, що дозволить донести ваші команди до пункту призначення.

Ще однією групою патернів є такі, що акцентують свою увагу на поведінці. Вони або інкапсулюють поведінку, або дозволяють її розподілити.

Щоб забезпечити почергову передачу роботи від одного класу до іншого і так дальше, аж до поки робота не буде виконана, використовують *Ланцюжок Відповідальностей*.

Інколи краще запакувати інформацію про дії, які слід виконати, в один об'єкт *Команди* і переслати на опрацювання, або ж просто виконати в потрібному місці.

Багато явищ можна описати за допомогою якоїсь спеціальної мови, наприклад погодні умови можуть бути записані значками, зрозумілими тільки метеорологам, але якщо вам подана граматика цієї мови і пояснення значків, цілком можливо, що ви зможете *Інтерпретувати* метеорологічне речення і зрозуміти його суть.

Колекції об'єктів можуть бути «хитрими» і містити багато підколекцій та поокремих об'єктів. Щоб спростити життя користувачу такої колекції та щоб не викривати логіки колекції, придумали *Ітератор*, який допомагає легко і грамотно обійти усі об'єкти всередині.

Спрощення координації роботи між деякою кількістю об’єктів може бути досягнуте виділенням посередника або медіатора.*Медіатором* може бути ваш бригадир на будівництві або ваш менеджер.

Можливість повернутися до попереднього стану системи має велике значення. Така функціональність може бути досягнута*Хранителем*.

Зверху завжди видніше, що коїться внизу. *Спостерігач* допоможе централізувати огляд роботи декількох класів та генерувати відповідні події.

Стан системи та умови переходу між ними можуть бути винесені в окремі класи для легшого контролю над цією системою. Все це досягається за допомогою дизайн патерну *Стан*.

Піти на право чи на ліво, а чи взагалі кудись йти? Відповідь на це питання може залежати від певних параметрів і є нічим іншим як певною *Стратегією*.

Втомились від одноманітної роботи, яка завжди шаблонна, окрім деталей, які час від часу міняються? Віддайте цю робому *Шаблонному Методу*.

Коли потрібно виконати деякі дії над об’єктом, причому вони кожного разу різні, такі дії можуть бути винесені в окремі класи-відвідувачі. Опісля ваш об’єкт може приймати *Відвідувачів* для виконання конкретних дій.

Тема 20: Тестування «Чорної скриньки». Тестування «Білої скриньки»

(4 години)

Мета: Розглянути особливості тестування методом «чорного ящика». Спосіб розбивки по класам еквівалентності. Спосіб аналізу граничних значень. Розглянути особливості тестування методом «білої скриньки». Спосіб тестування базового шляху. Потоковий граф. Розібрати кроки тестування базового шляху.

План:

1. Особливості тестування "чорного ящика";
2. Спосіб розбивки по еквівалентності;
3. Спосіб аналізу граничних значень;
4. Особливості тестування «білої скриньки»;

5. Спосіб тестування базового шляху;
6. Цикломатична складність;
7. Кроки способу тестування базового шляху.

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання.

Контрольні питання:

1. Які особливості тестування методом "чорного ящика"?
2. Які категорії помилок виявляє тестування методом "чорного ящика"?
3. Які гідності має тестування методом "чорного ящика"?
4. Поясните суть способу розбивки по еквівалентності.
5. Що таке клас еквівалентності?
6. Що може задавати умова введення?
7. Які правила формування класів еквівалентності ви знаєте?
8. Як вибирається тестовий варіант при тестуванні по способу розбивки по еквівалентності?
9. Поясните суть способу аналізу граничних значень.
10. Чим спосіб аналізу граничних значень відрізняється від розбивки по еквівалентності?
11. Поясните правила аналізу граничних значень.
12. Що таке дерево розбивок? Які його особливості?
13. У чому полягає суть тестування "білого ящика"?
14. Які особливості тестування "білого ящика"?
15. Які недоліки має тестування "білого ящика"?
16. Які гідності має тестування "білого ящика"?
17. Дайте характеристику способу тестування базового шляху.
18. Які особливості має потоковий граф?
19. Поясните поняття незалежного шляху.
20. Поясните поняття цикломатической складності.

21. Що така базова безліч?
22. Які властивості має базова безліч?
23. Які способи обчислення цикломатической складності ви знаєте?
24. Поясните кроки способу тестування базового шляху.
25. Поясните гідності, недоліки й область застосування способу тестування базового шляху.

Особливості тестування "чорного ящика"

Тестування "чорного ящика" (функціональне тестування) дозволяє одержати комбінації вхідних даних, що забезпечують повну перевірку всіх функціональних вимог до програми. Програмний виріб тут розглядається як "чорний ящик", чия поведінка можна визначити тільки дослідженням його входів і відповідних виходів. При такому підході бажане 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 ВІД).

Як показано на мал. 1, 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 ВІД.

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

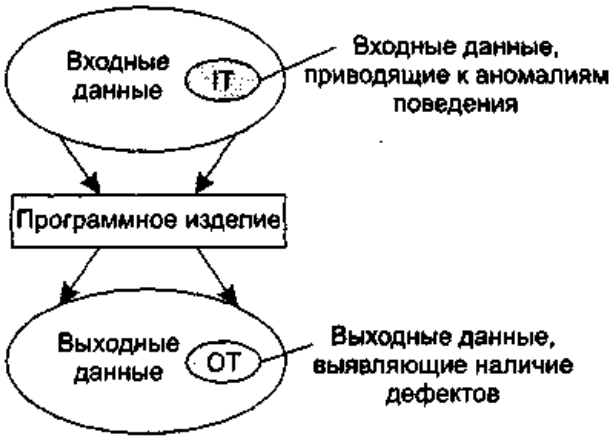


Рисунок 1 - Тестування "чорного ящика"

Принцип "чорного ящика" не альтернативний принципу "білого ящика". Скоріше це підхід, що доповнює, 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 й 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" зневажають керуючою структурою програми. Тут увага концентрується на інформаційній області визначення програмної системи.

Техніка "чорного ящика" орієнтована на розв'язок наступних завдань:

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

Спосіб розбивки по еквівалентності

Розбивка по еквівалентності - самий популярний спосіб тестування "чорного ящика".

У цьому способі вхідна область даних програми ділиться на класи еквівалентності. Для кожного класу еквівалентності розробляється один тестовий варіант.

Клас еквівалентності - набір даних із загальними властивостями. Обробляючи різні елементи класу, програма повинна поводитися однаково. Інакше кажучи, при обробці будь-якого набору із класу еквівалентності в програмі задіється той самий набір операторів (і зв'язків між ними).

На мал. 2 кожний клас еквівалентності показаний еліпсом. Тут виділені вхідні класи еквівалентності припустимих і неприпустимих вихідних даних, а також класи результатів.

Класи еквівалентності можуть бути визначені по специфікації на програму.

Рисунок 2 - Розбивка по еквівалентності

Наприклад, якщо специфікація задає в якості припустимих вхідних величин 5- розрядні цілі числа в діапазоні 15 000...70 000, то клас еквівалентності припустимих ИД (вихідних даних) включає величини від 15 000 до 70 000, а два класи еквівалентності неприпустимих ИД становлять:

- числа менші, чому 15 000;
- числа більші, ніж 70 000.

Клас еквівалентності включає безліч значень даних, припустимих або неприпустимих за умовами введення.

Умова введення може задавати:

1) певне значення;

2) діапазон значень;

3) безліч конкретних величин;

4) булево умова.

Сформулюємо правила формування класів еквівалентності.

1. Якщо умова введення задає діапазон п...т, то визначаються один припустимий і два неприпустимі класи еквівалентності:

- V_Class={n.. .т} - припустимий клас еквівалентності;
- Inv_31ass1={x|для будь-якого х: х < п} - перший неприпустимий клас еквівалентності;
- Inv_31ass2={у|для кожного в: в > т} - другий неприпустимий клас еквівалентності.

2. Якщо умова введення задає конкретне значення а, то визначається один припустимий і два неприпустимі класи еквівалентності:

- V_Class={a};
- Inv_Class1 ={x|для будь-якого х: х < а};
- Inv_31ass2={у|для кожного в: в > а}.

3. Якщо умова введення задає безліч значень {a, b, з}, то визначаються один припустимий і один неприпустимий клас еквівалентності:

- V_Class={a, b, з};
- Inv_31ass={x|для будь-якого х: (x a)&(x b)&(x c)}.

4. Якщо умова введення задає булево значення, наприклад true, то визначаються один припустимий і один неприпустимий клас еквівалентності:

- V_Class={true};
- Inv_Class={false}.

Після побудови класів еквівалентності розробляються тестові варіанти. Тестовий варіант вибирається так, щоб перевірити відразу найбільша кількість властивостей класу еквівалентності.

Спосіб аналізу граничних значень

Як правило, більша частина помилок відбувається на границях області введення, а не в центрі. Аналіз граничних значень полягає в одержанні тестових

варіантів, які аналізують граничні значення. Даний спосіб тестування доповнює спосіб розбивки по еквівалентності.

Основні відмінності аналізу граничних значень від розбивки по еквівалентності:

1) тестові варіанти створюються для перевірки тільки ребер класів еквівалентності;

2) при створенні тестових варіантів ураховують не тільки умови введення, але й область висновку.

Сформулюємо правила аналізу граничних значень.

1. Якщо умова введення задає діапазон п...т, то тестові варіанти повинні бути побудовані:

- для значень п и т;
- для значень ледве левее п і ледве правее т на числовій осі.

Наприклад, якщо заданий вхідний діапазон -1,0...+1,0, то створюються тести для значень - 1,0, +1,0, - 1,001, +1,001.

2. Якщо умова введення задає дискретна безліч значень, то створюються тестові варіанти:

- для перевірки мінімального й максимального зі значень;
- для значень ледве менше мінімуму й ледве більше максимуму.

Так, якщо вхідний файл може містити від 1 до 255 записів, то створюються тести для ПРО, 1, 255, 256 записів.

3. Правила 1 і 2 застосовуються до умов області висновку.

Розглянемо приклад, коли в програмі потрібно виводити таблицю значень. Кількість рядків і стовпців у таблиці міняється. Задається тестовий варіант для мінімального висновку (за обсягом таблиці), а також тестовий варіант для максимального висновку (за обсягом таблиці).

4. Якщо внутрішні структури даних програми мають запропоновані границі, то розробляються тестові варіанти, що перевіряють ці структури на їхніх границях.

5. Якщо вхідні або вихідні дані програми є впорядкованими безлічами (наприклад, послідовним файлом, лінійним списком, таблицею), то треба тестувати обробку першого й останнього елементів цих безлічей.

Більшість розроблювачів використовують цей спосіб інтуїтивно. При застосуванні описаних правил тестування границь буде більш повним, у зв'язку із чим зросте ймовірність виявлення помилок.

Розглянемо застосування способів розбивки по еквівалентності й аналізу граничних значень на конкретному прикладі. Покладемо, що потрібно протестувати програму бінарного пошуку. Нам відома специфікація цієї програми. Пошук виконується в масиві елементів M, вертається індекс I елемента масиву, значення якого відповідає ключу пошуку Key.

Предусловия:

1) масив повинен бути впорядкований;

2) масив повинен мати не менш одного елемента;

3) нижня границя масиву (індекс) повинна бути менше або рівна його верхній границі.

Постусловия:

1) якщо елемент знайдений, то прапор Result=True, значення Й - номер елемента;

2) якщо елемент не знайдений, то прапор Result=False, значення Й не визначене.

Для формування класів еквівалентності (і їх ребер) треба зробити розбивку області ИД - побудувати дерево розбивок. Листи дерева розбивок дадуть нам шукані класи еквівалентності. Визначимо стратегію розбивки. На першому рівні будемо аналізувати выполнимость предусловий, на другому рівні - выполнимость постусловий. На третьому рівні можна аналізувати спеціальні вимоги, отримані із практики розроблювача. У нашому прикладі ми знаємо, що вхідний масив повинен бути впорядкований. Обробка впорядкованих наборів з парної й непарної кількості елементів може виконуватися по-різному. Крім того, прийнято виділяти спеціальний випадок одноелементного масиву. Отже, на рівні спеціальних вимог можливі наступні еквівалентні розбивки:

1) масив з одного елемента;

2) масив з парної кількості елементів;

3) масив з непарної кількості елементів, більшого одиниці.

Нарешті на останньому, 4- м рівні критерієм розбивки може бути аналіз ребер класів еквівалентності. Очевидно, можливі наступні варіанти:

1) робота з першим елементом масиву;

2) робота з останнім елементом масиву;

3) робота із проміжним (ні з першим, ні з останнім) елементом масиву.

Структура дерева розбивок наведена на мал. 3.

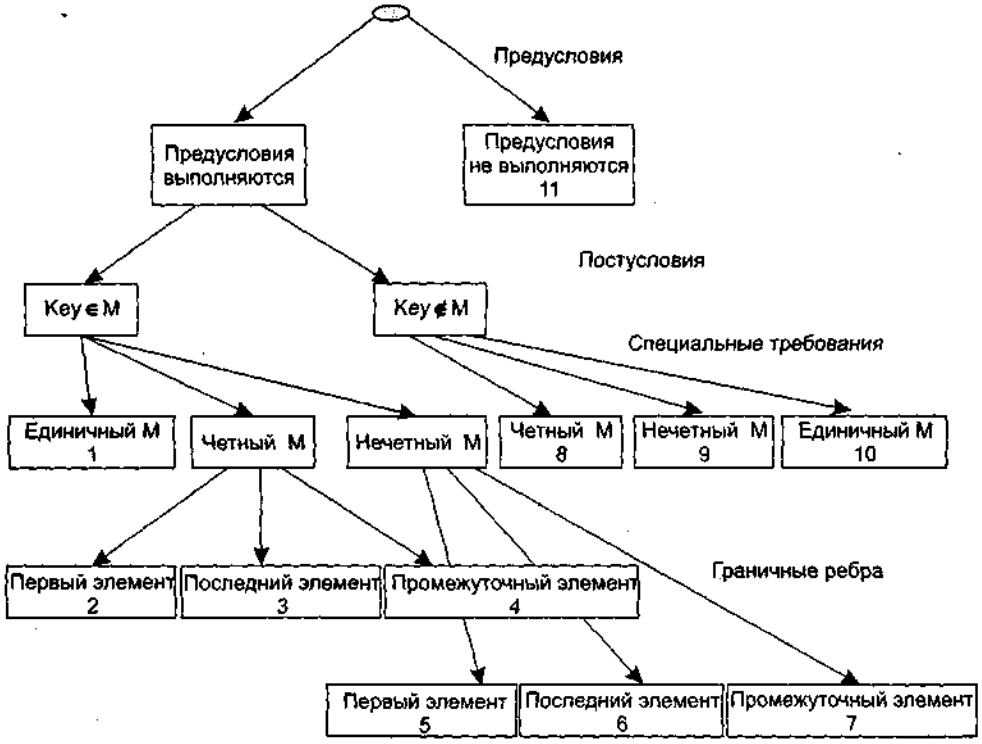


Рисунок 3 - Дерево розбивок області вихідних даних бінарного пошуку

Це дерево має 11 листів. Кожний аркуш задає окремий тестовий варіант. Покажемо тестові варіанти, засновані на проведених розбивках.

Тестовий варіант 1 (одиничний масив, елемент знайдений) ТВ1:

ИД: M=15; Key=15.

ОЖ.РЕЗ.: Resutt=True; I=1.

Тестовий варіант 2 (парний масив, знайдений 1- й елемент) ТВ2:

ИД: M=15, 20, 25,30,35,40; Key=15.

ОЖ.РЕЗ.: Result=True; I=1.

І так далі…

Звичайне тестування "білого ящика" засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреної, якщо проведене вичерпне тестування маршрутів (шляхів) її графа керування.

У цьому випадку формуються тестові варіанти, у яких:

- гарантується перевірка всіх незалежних маршрутів програми;
- проходяться галузі True, False для всіх логічних розв'язків;
- виконуються всі цикли (у межах їх границь і діапазонів);
- аналізується правильність внутрішніх структур даних.

Недоліки тестування "білого ящика":

1. Кількість незалежних маршрутів може бути дуже велике. Наприклад, якщо цикл у програмі виконується k раз, а усередині циклу є п розгалужень, то кількість маршрутів обчислюється по формулі

$$m = \sum_{i=1}^{k} n^i$$

При п = 5 і k = 20 кількість маршрутів т = 1014. Приймемо, що на розробку, виконання й оцінку тесту по одному маршруту витрачається 1 мс. Тоді при роботі 24 години на добу 365 днів у році на тестування піде 3170 років.

2. Вичерпне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.

3. У програмі можуть бути пропущені деякі маршрути.

4. Не можна виявити помилки, появу яких залежить від оброблюваних даних (це помилки, обумовлені вираженнями типу if abs (a-b) < eps..., if(a+b+c)/3=a...).

Гідності 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 керування або даних у програмі часто бувають некоректні. У результаті типовим може стати маршрут, модель обчислень по якому пророблена слабко.

3. При записі алгоритму ПО у вигляді тексту мовою програмування можливе внесення типових помилок трансляції (синтаксичних і семантичних).

4. Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

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

Тестування базового шляху

Тестування базового шляху - це спосіб, який заснований на принципі "білого ящика". Автор цього способу - Том МАККЕЙБ (1976) .

Спосіб тестування базового шляху дає можливість:

- одержати оцінку комплексної складності програми;
- використовувати цю оцінку для визначення необхідної кількості тестових варіантів.

Тестові варіанти розробляються для перевірки базової безлічі шляхів (маршрутів) у програмі. Вони гарантують однократне виконання кожного оператора програми при тестуванні.

Потоковий граф

Для вистави програми використовується потоковий граф. Перелічимо його особливості.

1. Граф будується відображенням керуючої структури програми. У ході відображення закриваючі дужки умовних операторів і операторів циклів (end if; end loop) розглядаються як окремі (фіктивні) оператори.

2. Вузли (вершини) потокового графа відповідають лінійним ділянкам програми, включають один або трохи операторів програми.

3. Дуги потокового графа відображають потік керування в програмі (передачі керування між операторами). Дуга - це орієнтоване ребро.

4. Розрізняють операторные й предикатні вузли. З операторного вузла виходить одна дуга, а із предикатного - дві дуги.

4. Предикатні вузли відповідають простим умовам у програмі. Складена умова програми відображається в кілька предикатних вузлів. Складовим називають умова, у якому використовується одна або трохи булевых операцій (OR, AND).

5. Наприклад, фрагмент програми

if a OR b

then x

else в

end if;

замість прямого відображення в потоковий граф виду, показаного на мал. 30.1, відображається в перетворений потоковий граф (мал. 30.2).

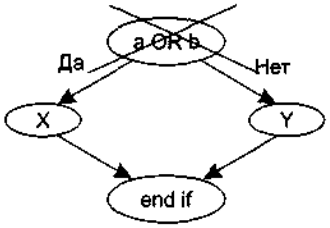


Рисунок 30.1 - Пряме відображення в потоковий граф

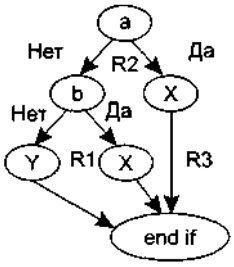


Рисунок 2 - Перетворений потоковий граф

6. Замкнені області, утворені дугами й вузлами, називають регіонами.

7. Навколишня граф середовище розглядається як додатковий регіон. Наприклад, показаний тут граф має три регіони - Rl, R2, R3.

Приклад 1. Розглянемо процедуру стиску:

процедура стиск

1 виконувати поки немає EOF

1 читати запис;

2 якщо запис порожній

3 те вилучити запис:

4 інакше якщо поле a >= поля b

5 те вилучити b;

6 інакше вилучити a;

7a кінець якщо;

7a кінець якщо;

7b кінець виконувати;

8 кінець стиск;

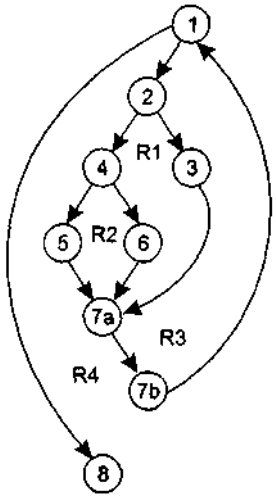


Рисунок 3 - Перетворений потоковий граф процедури стиску

Вона відображається в потоковий граф, представлений на мал. 30.3. Бачимо, що цей потоковий граф має чотири регіони.

Цикломатическая складність

Цикломатическая складність - метрика ПЗ, яка забезпечує кількісну оцінку логічної складності програми. У способі тестування базового шляху Цикломатическая складність визначає:

- кількість незалежних шляхів у базовій безлічі програми;
- верхню оцінку кількості тестів, яка гарантує однократне виконання всіх операторів.

Незалежним називається будь-який шлях, який уводить новий оператор обробки або нова умова. У термінах потокового графа незалежний шлях повинен містити дугу, що не входить у раніше певні шляхи.

Перелічимо незалежні шляхи для потокового графа із прикладу 1:

Шлях 1: 1-8.

Шлях 2: 1-2-3-7a-7b-1-8.

Шлях 3: 1-2-4-5-7a-7b-1-8.

Шлях 4: 1-2-4-6-7a-7b-1-8.

Помітимо, що кожний новий шлях включає нову дугу.

Усі незалежні шляхи графа утворюють базову безліч.

Властивості базової безлічі:

1) тести, що забезпечують його перевірку, гарантують:

- однократне виконання кожного оператора;
- виконання кожної умови по True- Галузі й по False- Галузі;

2) потужність базової безлічі рівна цикломатической складності потокового графа.

Значення 2- го властивості важко переоцінити - воно дає апріорну оцінку кількості незалежних шляхів, яке має сенс шукати в графові.

Цикломатическая складність обчислюється одним із трьох способів:

1) цикломатическая складність дорівнює кількості регіонів потокового графа;

2) цикломатическая складність визначається по формулі

V(G)-E-N+2,

де E - кількість дуг, N - кількість вузлів потокового графа;

3) цикломатическая складність формується по вираженню V(G) =p+ 1, де p - кількість предикатних вузлів у потоковому графові G.

Обчислимо цикломатическую складність графа із прикладу 1 кожним із трьох способів:

1) потоковий граф має 4 регіону;

2) V(G) = 11 дуг - 9 вузлів + 2 = 4;

3) V(G) = 3 предикатних вузла +1=4.

Таким чином, цикломатическая складність потокового графа із прикладу 1 рівна чотирьом.

Кроки способу тестування базового шляху

Для ілюстрації кроків даного способу використовуємо конкретну програму - процедуру обчислення середнього значення:

```
        процедура сред;
1               і := 1;
1               введено := 0;
1               колич := 0;
1               сум := 0;
        вып пока 2 -вел( і ) <> stop и введено <=500 - 3
4                       введено:= введено + 1;
        если 5 -вел( і ) >= мин и вел( і ) <= макс - 6
7                               то колич := колич + 1;
7                               сум := сум + вел( і );
8                       конец если;
8                       і := і + 1;
9       конец вып;
10      если колич > 0
11              то сред := сум / колич;
12      иначе сред := stop;
13      конец если;
13      конец сред;
```

Помітимо, що процедура містить складені умови (у заголовку циклу й умовному операторові). Елементи складених умов для наочності поміщені в рамки.

Крок 1. На основі тексту програми формується потоковий граф:

- нумеруються оператори тексту (номера операторів показані в тексті процедури);
- проводиться відображення пронумерованого тексту програми у вузли й вершини потокового графа (мал. 4).

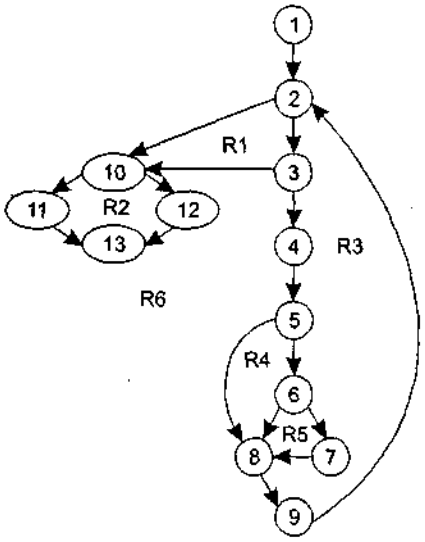


Рисунок 4 - Потоковий граф процедури обчислення середнього значення

Крок 2. Визначається цикломатическая складність потокового графа - по кожній із трьох формул:

1) V(G) = 6 регіонів;

2) V(G) = 17 дуг - 13 вузлів + 2 = 6;

3) V(G) = 5 предикатних вузлів + 1 = 6.

Крок 3. Визначається базова безліч незалежних лінійних шляхів:

Шлях 1: 1-2-10-11-13; /вів=stop, колич>0.

Шлях 2: 1-2-10-12-13;/вів=stop, колич=0.

Шлях 3: 1-2-3-10-11-13; /спроба обробки 501- й величини.

Шлях 4: 1-2-3-4-5-8-9-2-... /вів<хв.

Шлях 5: 1-2-3-4-5-6-8-9-2-... /вів>Макс.

Шлях 6: 1-2-3-4-5-6-7-8-9-2-... /режим нормальної обробки.

Крок 4. Підготовляються тестові варіанти, що ініціюють виконання кожного шляху.

Кожний тестовий варіант формується в наступному виді:

Вихідні дані (ИД):

Очікувані результати (ОЖ.РЕЗ.):

Вихідні дані повинні вибиратися так, щоб предикатні вершини забезпечували потрібні перемикання - запуск тільки тих операторів, які перераховані в конкретному шляху, причому в необхідному порядку.

Визначимо тестові варіанти, що задовольняють виявленій безлічі незалежних шляхів.

Тестовий варіант для шляху 1 ТВ1:

ИД: вів(k) = припустиме значення, де k < i; вів(i) = stop, де 2 < i < 500.

ОЖ.РЕЗ.: коректне усереднення ґрунтується на k величинах і правильному підрахунку. Тестовий варіант для шляху 2 ТВ2:

ИД: вів(1)=stop.

ОЖ.РЕЗ.: середовищ=stop, інші величини мають початкові значення.

Тестовий варіант для шляху 3 ТВ3:

ИД: спроба обробки 501- й величини, перші 500 величин повинні бути правильними.

ОЖ.РЕЗ.: коректне усереднення ґрунтується на k величинах і правильному підрахунку.

Тестовий варіант для шляху 4 ТВ4:

ИД: вів(i)=припустиме значення, де й ≤ 500; вів(k) < хв, де k < i.

ОЖ.РЕЗ.: коректне усереднення ґрунтується на k величинах і правильному підрахунку.

І так далі…

Реальні результати кожного тестового варіанта рівняються з очікуваними результатами. Після виконання всіх тестових варіантів гарантується, що всі оператори програми виконані щонайменше один раз.

Важливо відзначити, що деякі незалежні шляхи не можуть перевірятися изолированно. Такі шляхи повинні перевірятися при тестуванні іншого шляху (як частина іншого тестового варіанта).

Тема №21: Організація процесу тестування програмного забезпечення

(2 години)

Мета: Розглянути організацію процесу тестування програмного забезпечення. Розглянути процес тестування елементів.

План:

1. Організація процесу тестування програмного забезпечення;
    a. Методика тестування програмних систем;
    b. Тестування елементів.

Домашнє завдання:

Студентам необхідно:

1. Ознайомитись з матеріалом теми;
2. Зробити конспект матеріалу теми;
3. Відповісти на контрольні питання та виконати завдання.

Контрольні питання:

1. Поясните суть методики тестування програмної системи.
2. Коли й навіщо виконується тестування елементів? Який етап конструювання воно перевіряє?
3. Поясните суть тестування елементів.
4. Перелічите найбільш загальні помилки обчислень.
5. Перелічите джерела помилок порівняння й неправильних потоків керування.
6. На які ситуації орієнтоване тестування шляхів обробки помилок?
7. Що таке драйвер тестування?
8. Що таке заглушка?

Класичний процес тестування забезпечує перевірку результатів, отриманих на кожному етапі розробки. Як правило, він починається з тестування в малому, коли перевіряються програмні модулі, триває при перевірці об'єднання модулів у систему й завершується тестуванням у великому, при якому перевіряються відповідність програмного продукту вимогам замовника і його взаємодія з іншими компонентами комп'ютерної системи. Дана глава послідовно описує зміст кожного

кроку тестування. Тут же розглядається організація налагодження ПО, яка проводиться для усунення виявлених при тестуванні помилок.

Методика тестування програмних систем

Процес тестування поєднує різні способи тестування в спланову послідовність кроків, які приводять до успішної побудови програмної системи (ПС). Методика тестування ПС може бути представлена у вигляді спіралі, що розвертається (рис. 1).

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 спирали проводиться тестування правильності, що перевіряє коректність етапу аналізу вимог до ПС. На заключному витку спирали проводиться системне тестування, що виявляє дефекти етапу системного аналізу ПС.

Охарактеризуємо кожний крок процесу тестування.

1. Тестування елементів. Ціль - індивідуальна перевірка кожного модуля. Використовуються способи тестування "білого ящика".

Рисунок 1 - Спіраль процесу тестування ПС

2. Тестування інтеграції. Ціль - тестування складання модулів у програмну систему. В основному застосовують способи тестування "чорного ящика".

3. Тестування правильності. Ціль - перевірити реалізацію в програмній системі всіх функціональних і поведінкових вимог, а також вимоги ефективності. Використовуються винятково способи тестування "чорного ящика".

4. Системне тестування. Ціль - перевірка правильності об'єднання й взаємодії всіх елементів комп'ютерної системи, реалізації всіх системних функцій.

Організація процесу тестування у вигляді еволюційної спіралі, що розвертається, забезпечує максимальну ефективність пошуку помилок. Однак виникає питання - коли закінчувати тестування?

Відповідь практика звичайно заснована на статистичному критерії: "Можна з 95%-ний упевненістю сказати, що провели достатнє тестування, якщо ймовірність безвідмовної роботи ЦП із програмним виробом протягом 1000 годин становить щонайменше 0,995".

Науковий підхід при відповіді на це питання полягає в застосуванні математичної моделі відмов. Наприклад, для логарифмічної моделі Пуассона формула розрахунків поточної інтенсивності відмов має вигляд:

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

де - поточна інтенсивність програмних відмов (кількість відмов в одиницю часу); - початкова інтенсивність відмов (на початку тестування); p - експонентне зменшення інтенсивності відмов за рахунок, що виявляються помилок, що й усуваються; t -час тестування.

За допомогою рівняння (16.1) можна передбачити зниження помилок у ході тестування, а також час, що вимагається для досягнення припустиме низкою інтенсивності відмов.

Тестування елементів

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

Тестуванню зазнають:

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

Інтерфейс модуля тестується для перевірки правильності введення-висновку тестової інформації. Якщо немає впевненості в правильному введенні-висновку даних, нема рації проводити інші тести.

Дослідження внутрішніх структур даних гарантує цілісність даних, що зберігаються.

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

Найбільш загальними помилками обчислень є:

1) неправильний або незрозумілий пріоритет арифметичних операцій;

2) змішана форма операцій;

3) некоректна ініціалізація;

4) непогодженість у виставі точності;

5) некоректна символічна вистава виражень.

Джерелами помилок порівняння й неправильних потоків керування є:

1) порівняння різних типів даних;

2) некоректні логічні операції й пріоритетність;

3) очікування еквівалентності в умовах, коли помилки точності роблять еквівалентність неможливої;

4) некоректне порівняння змінних;

5) неправильне припинення циклу;

6) відмова у виході при відхиленні ітерації;

7) неправильна зміна змінних циклу.

Звичайно при проектуванні модуля передбачать деякі помилкові умови. Для захисту від помилкових умов у модуль уводять шляхи обробки помилок. Такі шляхи теж повинні тестуватися. Тестування шляхів обробки помилок можна орієнтувати на наступні ситуації:

1) повідомлення про помилку незрозуміло;

2) текст повідомлення не відповідає, виявленій помилці;

3) втручання системних засобів реєстрації аварії відбулося до обробки помилки в модулі;

4) обробка виняткової умови некоректна;

5) опис помилки не дозволяє визначити її причину.

І, нарешті, перейдемо до граничного тестування. Модулі часто відмовляють на "границях". Це означає, що помилки часто відбуваються:

1) при обробці n- го елемента n- елементного масиву;

2) при виконанні m- й ітерації циклу з т проходами;

3) з появою мінімального (максимального) значення.

Тестові варіанти, орієнтовані на дані ситуації, мають високу ймовірність виявлення помилок.

Тестування елементів звичайно розглядається як доповнення до етапу кодування. Воно починається після розробки тексту модуля. Тому що модуль не є автономною системою, то для реалізації тестування потрібні додаткові засоби, представлені на мал. 2.

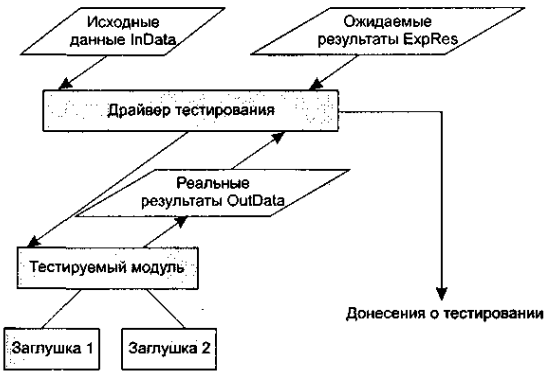


Рисунок 2 - Програмне середовище для тестування модуля

Додатковими засобами є драйвер тестування й заглушки. Драйвер - керуюча програма, яка ухвалює вихідні дані (Indata) і очікувані результати (Expres) тестових варіантів, запускає в роботу тестируемый модуль, одержує з модуля реальні результати (Outdata) і формує повідомлення про тестування. Алгоритм роботи тестового драйвера наведений на мал. 3.

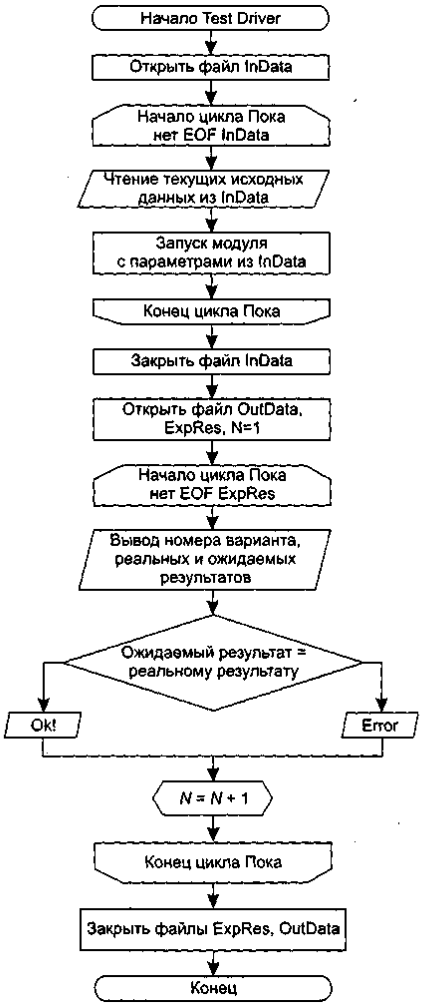


Рисунок 3 - Алгоритм роботи драйвера тестування

Заглушки заміщають модулі, які викликаються тестируемым модулем. Заглушка, або "фіктивна підпрограма", реалізує інтерфейс підлеглого модуля, може виконувати мінімальну обробку даних, імітує приймання й повернення даних.

Створення драйвера й заглушок має на увазі додаткові витрати, тому що вони не поставляються з кінцевим програмним продуктом.

Якщо ці засоби прості, то додаткові витрати невеликі. На жаль, багато модулів не можуть бути адекватно протестовані за допомогою простих додаткових засобів. У цих випадках повне тестування може бути відкладене до кроку тестування інтеграції (де драйвери або заглушки також використовуються).

Тестування елемента просто здійснити, якщо модуль має високу связность. При реалізації модулем тільки однієї функції кількість тестових варіантів зменшується, а помилки легко передвіщаються й виявляються.

Рекомендована література

1 Основна література

1. Крег Ларман «Книга Застосування UML 2.0 і шаблонів проєктування. 3-тє вид», Диалектика, 736 с. ISBN - 978-5-907144-36-1
2. Martin Fowler «UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition Addison-Wesley Professional, 2016 p., 208 c.
3. Martin Fowler «UML Distilled: A Brief Guide to the Standard Object Modeling Language, 2 Edition Addison-Wesley Professional, 2013 p., 192 c.
4. http://www.znannya.org – електноний ресурс.

2 Додаткова література

1. Шенталер Ф. "Бізнес-процеси. Мови моделювання, методи, інструменти", Альпіна Паблішер, 2019 р., 264 с.