МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДНІПРОВСЬКИЙ ДЕРЖАВНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

«Архітектура та проектування програмного забезпечення»

для здобувачів вищої освіти першого (бакалаврського) рівня
за освітньо-професійною програмою
«Інженерія програмного забезпечення»
із спеціальності 121 – «Інженерія програмного забезпечення»
галузі знань 12 – «Інформаційні технології»

Затверджено редакційно-видавничою
секцією науково-методичної ради ДДТУ
протокол №___ від _____ 2019р.

Кам’янське
2019

Конспект лекцій з дисципліни «Архітектура та проектування програмного забезпечення» для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Інженерія програмного забезпечення» із спеціальності 121 – «Інженерія програмного забезпечення» / Укл. В.В.Завгородній, К.М.Ялова.– Кам'янське: ДДТУ, 2019.– 144с.

Відповідальний за випуск:

д. т. н., професор Шумейко О.О.

Рецензент:

д. т. н., професор Самохвалов С.Є.

Затверджено на засіданні кафедри „Програмне забезпечення систем"

Протокол № 1 від «30» серпня 2019 р.

Коротка анотація видання. У конспекті лекцій з дисципліни " Архітектура та проектування програмного забезпечення " наведено основні підходи та правили до розробки ПЗ; технології створення ПЗ; аналіз, характеристика та структура ПЗ; проектування архітектури ПЗ; стратегії и методи проектування ПЗ; стандарти та інструментальні засоби при виборі архітектури ПЗ; архітектурні шаблони і стилі; нотації та засоби підтримки проектування.

ЗМІСТ

ВСТУП

Сьогодні програмна інженерія - молода та швидко розвивається область, яка орієнтована на вирішення складних завдань, що пов'язані з організацією, змістом і управлінням процесами розробки програмних систем.

Програмні системи - це унікальні продукти, що не мають фізичної оболонки. Створення таких систем є однією з найскладніших завдань і вимагає великого інтелектуального потенціалу. У результаті для процесу розробки створювалися абсолютно нові підходи та технології.

Сучасне суспільство вже немислимо без інформаційних систем, які пронизують майже всі галузі діяльності і постійно розвиваються і удосконалюються. Поліпшення якості інформаційних систем нерозривно пов'язане з процесом їх виробництва. Кожен маленький елемент відіграє дуже велике значення.

Процес розробки програмного забезпечення орієнтований на життєвий цикл. Процес розробки програмного забезпечення пов'язаний з областю управління проектами, тому що будь-який програмний продукт є унікальним результатом. Від організації цього процесу залежать основні характеристики виконання програмного проекту - терміни виконання, запланований бюджет та якість продукту, що випускається. Важливу роль в цьому відіграє архітектура програмної системи, досвід і кваліфікація учасників команди розробки, а також правильне документування всіх процесів розробки програмного забезпечення.

В даному курсі будуть розглянуті основні підходи та правили до розробки ПЗ; технології створення ПЗ; аналіз, характеристика та структура ПЗ; проектування архітектури ПЗ; стратегії и методи проектування ПЗ; стандарти та інструментальні засоби при виборі архітектури ПЗ; архітектурні шаблони і стилі; нотації та засоби підтримки проектування.

Тема 1. Основні підходи до розробки програмного забезпечення

1.1 Основні етапи розвитку технології розробки

Технологією програмування - сукупність методів і засобів, що використовуються в процесі розробки ПЗ. Як будь-яка інша технологія, ***технологія програмування*** - це набір технологічних інструкцій, що включають: 1. Вказівка послідовності виконання технологічних операцій; перерахування умов, при яких виконується та чи інша операція. 2. Описи самих операцій, де для кожної операції визначені вихідні дані, результати, а також інструкції, нормативи, стандарти, критерії, методи оцінки і т. п. (рис. 1.1).

Рисунок 1.1 — Структура опису технологічної операції

Розрізняють технології, використовувані на конкретних етапах розробки або для вирішення окремих завдань цих етапів (основі лежить обмежено застосовний метод, що дозволяє вирішити конкретну задачу), і технології, що охоплюють кілька етапів або весь процес розробки (базовий метод або підхід, що визначає сукупність методів, використовуваних на різних етапах розробки, або проектованої системи, точніше моделі, використовуваної на конкретному етапі розробки). Розглянемо їх в історичному контексті, виділяючи основні етапи розвитку програмування як науки.

1.2 «Стихійне» програмування

Цей етап охоплює період від моменту появи перших обчислювальних

машин до середини 60-х рр. XX ст. У цей час практично відсутні технології розробки ПЗ та програмування фактично було мистецтвом. Перші програми мали найпростішу структуру. Вони складалися з власне програми на машинній мові і оброблюваних нею даних. Складність програм у машинних кодах обмежувалася здатністю програміста одночасно відстежувати послідовність виконуваних операцій та місцезнаходження даних при програмуванні. Створення мов програмування високого рівня, таких як FORTRAN і ALGOL, істотно спростило програмування обчислень, знизивши рівень деталізації операцій. Це дозволило збільшити складність програм. Революційним була поява в мовах засобів, що дозволяють оперувати підпрограмами. Підпрограми можна було зберігати і використовувати в інших програмах. У результаті були створені величезні бібліотеки розрахункових та службових підпрограм, які в міру потреби викликалися з розроблюваної програми. Слабким місцем такої архітектури було те, що при збільшенні кількості підпрограм зростала ймовірність спотворення частини глобальних даних якою-небудь підпрограмою. Щоб скоротити кількість таких помилок, було запропоновано в підпрограма розміщувати локальні дані. На початку 60-х рр. XX ст. вибухнула «криза програмування». Вона висловлювалася в тому, що фірми, які взялися за розробку складного програмного забезпечення такого, як операційні системи, зривали всі терміни завершення проектів. Проект застарівав раніше, ніж був готовий до впровадження, збільшувалася його вартість, і в результаті багато проектів так ніколи і не були завершені. Об'єктивно все це було викликано недосконалістю технології програмування. Насамперед, стихійно використовувалася розробка «знизу-вгору» - підхід, при якому спочатку проектували і реалізовували порівняно прості підпрограми, з яких потімнамагалися побудувати складну програму. Зрештою процес тестування і налагодження програм займав більше 80% часу розробки, якщо взагалі коли-небудь закінчувався. На порядку денному найсерйознішим чином стояло питання розробки технології створення складних програмних продуктів, що знижує ймовірність помилок проектування. Аналіз причин виникнення більшості помилок дозволив сформулювати новий підхід до програмування, який був названий «структурним».

1.3 Структурний підхід до програмування (60-70-і рр. XX ст.)

Структурний підхід до програмування являє собою сукупність рекомендованих технологічних прийомів, що охоплюють виконання всіх етапів розробки ПЗ. В основі структурного підходу лежить декомпозиція (розбиття на частини) складних систем з метою подальшої реалізації у вигляді окремих невеликих підпрограм. З появою інших принципів декомпозиції даний спосіб отримав назву «процедурної декомпозиції». Проектування здійснювалося «зверху-вниз» і передбачало реалізацію спільної ідеї, забезпечуючи опрацювання інтерфейсів підпрограм. Одночасно вводилися обмеження на конструкції алгоритмів, рекомендувалися формальні моделі та опису, а також спеціальний метод проектування алгоритмів - метод деталізації. Підтримка принципів структурного програмування була закладена в основу так званих процедурних мов програмування. Серед найбільш відомих мов цієї групи варто назвати PL/1, ALGOL-68, Pascal, C. Одночасно зі структурним програмуванням з'явилася величезна кількість мов, що базуються на інших концепціях, але більшість з них не витримало конкуренції. Якісь мови були просто забуті, ідеї інших були в подальшому використані в наступних версіях. Подальше зростання складності і розмірів розроблюваного ПЗ зажадав розвитку структурування даних. Як наслідок цього в мовах з'являється можливість визначення користувача типів даних. Одночасно посилилося прагнення розмежувати доступ до глобальних даними програми, щоб зменшити кількість помилок, що виникають при роботі з глобальними даними. У результаті з’явилася і почала розвиватися технологія модульного програмування**.** Модульне програмування передбачає виділення груп підпрограм, що використовують одні й ті ж глобальні дані в окремо компільовані модулі (бібліотеки підпрограм). Зв'язки між модулями при використанні даної технології здійснюються через спеціальний інтерфейс, у той час як доступ до реалізації модуля заборонений. Цю технологію підтримують сучасні версії мов Pascal і C, мови Ада і Modula. Використання модульного програмування істотно спростило розробку ПЗ кількома програмістами. Тепер кожен з них міг розробляти свої модулі незалежно, забезпечуючи взаємодію модулів через спеціально

обумовлені між модульні інтерфейси. Крім того, модулі надалі без змін можна було використовувати в інших розробках, що підвищило продуктивність праці програмістів. Для розробки програмного забезпечення великого обсягу було запропоновано використовувати об'єктний підхід.

1.4 Об'єктний підхід до програмування (з середини 1980-х рр.. до нашого часу)

Об'єктно-орієнтоване програмування визначається як технологія створення складного ПЗ, заснована на уявленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного класу, а класи утворюють ієрархію зі спадкуванням властивостей. Об'єктна структура програми вперше була використана в мові імітаційного моделювання складних систем Simula, що з'явилося ще в 60-х рр. XXст. Природний для мов моделювання спосіб представлення програми отримав розвиток в іншому спеціалізованому мові моделювання - мові Smalltalk (70-і рр. XX ст.), А потім був використаний в нових версіях універсальних мов програмування, таких як Pascal, C, Modula, Java. Основною перевагою об'єктно-орієнтованого програмування є «більш природна» декомпозиція програмного забезпечення, яка істотно полегшує його розробку. Це призводить до більш повної локалізації даних та інтегруванню їх з підпрограмами обробки, що дозволяє вести практично незалежну розробку окремих частин (об'єктів) програми. Крім цього, об'єктний підхід пропонує нові способи організації програм, засновані на механізмах спадкування, поліморфізму, композиції, наповнення. Ці механізми дозволяють конструювати складні об'єкти з порівняно простих. У результаті істотно збільшується показник повторного використання кодів і з'являється можливість створення бібліотек класів для різних застосувань. Бурхливий розвиток технологій програмування, заснованих на об'єктних підході, дозволило вирішити багато проблем. Так, були створені середовища, що підтримують візуальне програмування, наприклад, Delphi, C++ Builder, Visual C++. При використанні візуального середовища у програміста з'являється можливість проектувати деяку частину, наприклад, інтерфейси майбутнього продукту, із застосуванням візуальних засобів

додавання та налаштування спеціальних бібліотечних компонентів. При використанні цих мов програмування зберігається об'єктивна залежність модулів ПЗ, так як модулі повинні взаємодіяти між собою, звертаючись до ресурсів один одного. Зв'язки модулів не можна розірвати, але можна спробувати стандартизувати їх взаємодія, на чому і заснований компонентний підхід до програмування.

1.5 Компонентний підхід і CASE-технології (з середини 1990-х рр. до нашого часу)

Компонентний підхід передбачає побудову ПЗ з окремих компонентів - фізично окремо існуючих частин ПЗ, які взаємодіють між собою через стандартизовані виконавчі інтерфейси. На відміну від звичайних об'єктів об'єкти-компоненти можна зібрати в динамічне викликані бібліотеки або виконувані файли, поширювати в двійковому вигляді (без вихідних текстів) і використовувати в будь-якій мові програмування, що підтримує відповідну технологію. Компонентний підхід лежить в основі технологій, розроблених на базі COM (Component Object Model - Компонентна модель об'єктів), і технології створення розподілених додатків CORBA (Common Object Request Broker Architecture - загальна архітектура з посередником обробки запитів об'єктів). Ці технології використовують подібні принципи і розрізняються лише особливостями їх реалізації.Технологія COM фірми Microsoft є розвитком технології OLE (Object Linked and Embedding - зв'язування і впровадження об'єктів), яка використовувалася в ранніх версіях Windows для створення складених документів. Об'єкт завжди функціонує у складі сервера - динамічної бібліотеки або виконуваного файлу, що забезпечують функціонування об'єкта. Розрізняють три типи серверів: 1. Внутрішній сервер: реалізується динамічними бібліотеками, які підключаються до додатку-клієнта і працюють в одному з ними адресному просторі. Це найбільш ефективний сервер, крім того, він не вимагає спеціальних засобів. 2. Локальний сервер: створюється окремим процесом (exe), який працює на одному комп'ютері з клієнтом. 3. Віддалений сервер: створюється процесом, який

працює на іншому комп'ютері.

1.6 Розробка, орієнтована на архітектуру і CASE-технології (з початку XXI ст. до нашого часу)

До кінця XX ст. всілякі проектні моделі вживали виключно для документування проміжних і заключних етапів розробки ПЗ, для чого застосовувалися різні графічні нотації і технології, які згодом були використані для створення стандарту об'єктного моделювання. У листопаді 1997 р. після тривалого процесу об'єднання різних методик група OMG (Object Management Group) прийняла вийшов внаслідок уніфікована мова моделювання (Unified Model Language - UML) в якості стандарту. У 2001 р. члени OMG почали роботу над новою версією UML, додаючи в неї відсутні елементи і усуваючи недоліки, виявлені в UML1. Версія UML2 була прийнята в 2004 р. З офіційною специфікацією UML можна ознайомиться на веб-сайті OMG за адресую www.omg.org. Ідея створення мови UML включала в себе реалізацію можливості використання UML як мови програмування. Цей момент викликав масу проблем при здійсненні, так як мова візуального моделювання за визначенням не міг містити в собі всієї виразності об'єктно-орієнтованих мов в плані проектування (програмування) динаміки та реалізації алгоритмів. Потрібна була мова, яка на більш високому (абстрактному) рівні змогла би забезпечити розробку на UML. Такою мовою була створена об'єктна мова обмежень OCL (Object Constraint Language). Коли розробка програмних систем починається від проектування її структури до подальшого кодування і всі зміни у функціях розроблюваної системи реалізуються починаючи з перепроектування архітектури, то така технологія називається орієнтованої на архітектуру (Model Driven Architecture - MDA). Компанія Borland починаючи з сьомої версії свого середовища розробки (Delphi) вже використовує набір компонентів, який реалізує підхід, орієнтований на архітектуру, але в цій версії присутня ще маса недоліків і недоробок.

1.7 Контрольні питання

1. Що таке технологія програмування?
2. Які основні етапи та принципи «стихійного» програмування?
3. Які основні етапи та принципи структурного підходу до програмування?
4. Які основні етапи та принципи компонентного підходу?

Тема 2. Застосування візуального моделювання з використаних мови UML в процесі розробки програмного забезпечення

2.1 Цілі та історія створення UML

Уніфікована мова моделювання UML (Unified Modeling Language) - уніфікована мова об'єктно-орієнтованого моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. Створення UML фактично почалося в кінці 1994 р., коли Граді Буч і Джеймс Рамбо почали роботу з об'єднання їхніх методів Booch [Буч-1999] і OMT (Object Modeling Technique) під егідою компанії Rational Software. До кінця 1995 р. вони створили першу специфікацію об'єднаного методу, названого ними Unified Method, версія 0.8. Тоді ж у 1995 р. до них приєднався творець методу OOSE (Object-Oriented Software Engineering) Івар Якобсон. Таким чином, UML є прямим об'єднанням і уніфікацією методів Буча, Рамбо і Якобсона, однак доповнює їх новими можливостями. UML знаходиться в процесі стандартизації, проведеному консорціумом OMG (Object Management Group), в даний час він прийнятий в якості стандартного мови моделювання і отримав широку підтримку. UML прийнятий на озброєння практично всіма найбільшими компаніями - виробниками програмного забезпечення (Microsoft, IBM, Hewlett- Packard, Oracle, Sybase).

2.2 Засоби UML

UML містить стандартний набір діаграм і нотацій найрізноманітніших видів. Стандарт UML версії 1.1, прийнятий OMG в 1997 р., пропонує наступний набір діаграм для моделювання:

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

S **діаграми взаємодії (interaction diagrams), які діляться на: діаграми послідовності (sequence diagrams) і кооперативні діаграми (collaboration diagrams) - для моделювання процесу обміну повідомленнями між об'єктами;**

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

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

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

S **діаграми компонентів (ccomponent diagrams)** - для моделювання ієрархії компонентів (підсистем) системи;

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

2.2.1 Діаграми варіантів використання

Варіант використання являє собою послідовність дій (транзакцій), виконуваних системою у відповідь на подію, що ініціюється деяким зовнішнім об'єктом (дійовою особою). Варіант використання описує типове взаємодія між користувачем і системою. У простому випадку варіант використання визначається в процесі обговорення з користувачем тих функцій, які він хотів би реалізувати.

Дійова особа (actor) **-** це роль, яку користувач грає по відношенню до системи. Дійові особи представляють собою ролі, а не конкретних людей або назви робіт. Незважаючи на те, що на діаграмах варіантів використання вони зображають у вигляді стилізованих людських фігурок, дійова особа може також бути зовнішньою системою,

якій необхідна деяка інформація від даної системи. Показувати на діаграмі дійових осіб слід тільки в тому випадку, коли їм дійсно необхідні деякі варіанти використання.

Дійові особи діляться на три основні типи - *користувачі системи, інші системи, які взаємодіють з даною, і час*. Час стає дійовою особою, якщо від нього залежить запуск яких подій в системі. На рис. 2.1 показаний приклад такої діаграми для банкомату (Automated Teller Machine, ATM). На даній діаграмі людські фігурки позначають *дійових осіб,* овали - *варіанти використання,* а лінії і стрілки - *різні зв'язки між діючими особами та варіантами використання*.

Рисунок 2.1 - Приклад діаграми варіантів використання

У мові UML на діаграмах варіантів використання підтримується кілька типів зв'язків між елементами діаграми. Це зв'язку комунікації (communication), включення (include), розширення (extend) і узагальнення (generalization).

Зв'язок комунікації - це зв'язок між варіантом використання і дійовою особою. Мовою UML зв'язку комунікації показують за допомогою односпрямованої асоціації (суцільної лінії зі стрілкою). Напрямок стрілки дозволяє зрозуміти, хто ініціює комунікацію. Мовою UML зв'язку включення та розширення показують у вигляді залежностей з відповідними стереотипами, як показано на рис. 2.2.

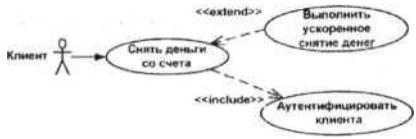

Рисунок 2.2 - Зв'язки використання та розширення

За допомогою зв'язку узагальнення показують, що у кількох діючих осіб існують спільні риси. Наприклад, клієнти можуть бути двох типів: корпоративні та індивідуальні. Цей зв'язок можна моделювати за допомогою нотації, показаної на рис. 2.3. ***Кожен варіант використання -*** це потенційне вимога до системи, і поки воно не виявлено, неможливо запланувати його реалізацію.

Рисунок 2.3 - Узагальнення діючої особи

2.2.2 Діаграми взаємодії

Діаграми взаємодії (interaction diagrams) описують поведінку взаємодіючих груп об'єктів, охоплює поведінку об'єктів в рамках тільки одного варіанту використання. На такий діаграмі відображається ряд об'єктів і ті повідомлення, якими вони обмінюються між собою. ***Повідомлення (message) -*** це засіб, за допомогою якого об'єкт-відправник запитує у об'єкта одержувача виконання однієї з його операцій. ***Інформаційне (informative) повідомлення -*** це повідомлення, що постачає об'єкт-одержувач деякою інформацією для оновлення його стану. ***Повідомлення-запит (interrogative)*** - це повідомлення, що запитує видачу деякою інформацією про об'єкт-одержувачі. ***Імперативне***
(imperative) повідомлення - це повідомлення, що подає запит у об'єкта- одержувача виконання деяких дій. Існує два види діаграм взаємодії: діаграми послідовності (sequence diagrams) і кооперативні діаграми (collaboration diagrams).

2.2.2.1 Діаграми послідовності

Діаграми послідовності (sequence diagrams) відображають потік подій, що відбуваються в рамках варіанту використання. Наприклад, варіант використання «Зняти гроші» передбачає кілька можливих послідовностей, такі як зняття грошей, спроба зняти гроші, не маючи їх достатньої кількості на рахунку, спроба зняти гроші по неправильному ідентифікаційному номеру і деякі інші. (рис. 2.4). На діаграмі послідовності об'єкт зображується у вигляді прямокутника, від якого вниз проведена пунктирна вертикальна лінія. Ця лінія називається лінією життя (lifeline) об'єкта.

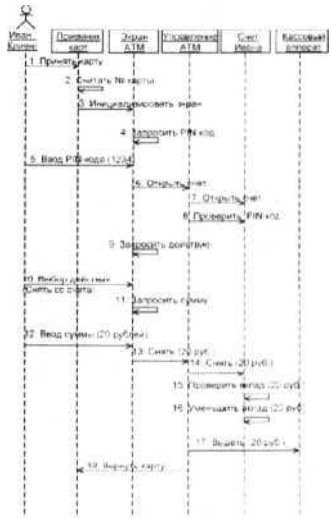

Рисунок 2.4 - Діаграма послідовності для зняття клієнтом грошей з рахунку

Кожне повідомлення представляється у вигляді стрілки між *лініями життя* двох об'єктів. Повідомлення з'являються в тому порядку, як вони показані на сторінці зверху вниз. Повідомлення буде позначено як мінімум ім'ям повідомлення і деяку керуючу інформацію і можна показати *само-делегування* (self-delegation) - повідомлення, яке об'єкт посилає самому собі, при цьому стрілка повідомлення вказує на ту ж саму лінію життя.

2.1.2.2 Кооперативні діаграми

Кооперативні діаграми (collaborations) відображають потік подій через конкретний сценарій варіанту використання. Діаграми послідовності впорядковані за часом, а кооперативні діаграми більше уваги загострюють на зв'язках між об'єктами. На рис. 2.5 приведена кооперативна діаграма, що описує, як клієнт знімає гроші з рахунку. Тут представлена вся та інформація, яка була і на діаграмі послідовності, але кооперативна діаграма по-іншому описує потік подій. З неї легше зрозуміти зв'язки між об'єктами, однак, важче усвідомити послідовність подій.

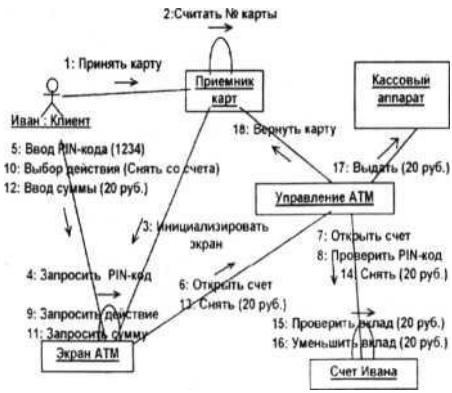

Рисунок 2.5 - Кооперативна діаграма, що описує процес зняття клієнтом грошей зі свого рахунку

2.2.3 Діаграми класів

Діаграма класів визначає типи класів системи і різного роду статичні зв'язки, які існують між ними. На діаграмах класів зображуються також атрибути класів, операції класів та обмеження, які накладаються на зв'язку між класами. Діаграма класів для варіанту використання «Зняти гроші» (рис. 2.6).

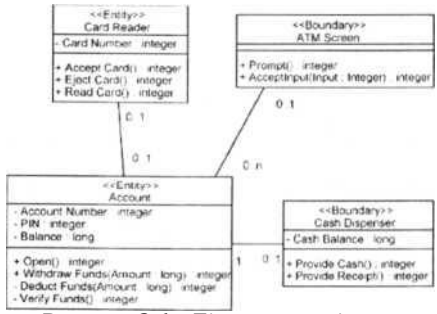

Рисунок 2.6 - Діаграма класів для варіанту використання «Зняти гроші»

На цій діаграмі класів показані зв'язки між класами, що реалізовують варіант використання «Зняти гроші». У *цьому процесі задіяні чотири класи:* Card Reader (пристрій для читання карток), Account (рахунок), ATM Screen (екран АТМ) і Cash Dispenser (касовий апарат). Кожен клас на діаграмі виглядає у вигляді прямокутника, розділеного натри частини. У *першій міститься ім'я класу, в другій - його атрибути.* В останній частині містяться *операції класу,* що відображають його поведінку (дії, що виконуються класом).

Стереотипи - це механізм, що дозволяє розділяти класи на категорії. У мові UML визначено три основних стереотипу класів: Boundary (кордон), Entity (сутність) і Control (управління).

Гоаничними класами (boundary classes) називаються такі класи, які розташовані на кордоні системи і всієї навколишнього середовища. Це екранні форми, звіти, інтерфейси з апаратурою (такий як принтери або сканери) та інтерфейси

з іншими системами.

Класи-сутності (entity classes) містять збережену інформацію. Вони мають найбільше значення для користувача, і тому в їхніх назвах часто використовують терміни з предметної області. Зазвичай для кожного класу- сутності створюють таблицю в базі даних.

Керуючі класи (control classes) відповідають за координацію дій інших класів. Зазвичай у кожного варіанту використання є один керуючий клас, контролюючий послідовність подій цього варіанту використання.

Механізм пакетів застосуємо до будь-яких елементів моделі, а не тільки до класів. Якщо для групування класів не використовувати деякі евристики, то вона стає довільною. Одна з них, яка в основному використовується в UML, - *це залежність*. Залежність між двома пакетами існує в тому випадку, якщо між будь-якими двома класами в пакетах існує будь-яка залежність. Таким чином, *діаграма пакетів* (рис. 2.7) являє собою діаграму, що містить пакети класів і залежності між ними. Строго кажучи, пакети і залежності є елементами діаграми класів, тобто *діаграма пакетів* - це форма діаграми класів.

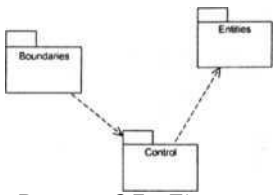

Рисунок 2.7 - Діаграма пакетів

Атрибут - це елемент інформації, пов'язаний з класом. Наприклад, у класу Company (компанія) можуть бути атрибути Name (Назва), Address (Адреса) і NumberOfEmployees (Число службовців). Так як атрибути містяться всередині класу, вони приховані від інших класів. У зв'язку з цим може знадобитися вказати, які класи мають право читати і змінювати атрибути. Це властивість називається *видимістю атрибута* (attribute visibility). *У атрибута можна*

*визначити чотири можливих значення цього параметра***.** Нехай у нас є клас Employee з атрибутом Address і клас Company (рис. 2.8):

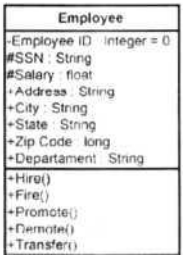

Рисунок 2.8 - Видимість атрибутів

■У *Public (загальний, відкритий)***.** Це значення видимості припускає, що атрибут буде видно усіма іншими класами. Будь клас може переглянути або змінити значення атрибута. У такому випадку клас Company може змінити значення атрибута Address класу Employee. Відповідно до нотацією UML загальному атрибуту передує знак «+».

У *Private (закритий, секретний)***.** Відповідний атрибут не видний ніяким іншим класом. Клас Employee буде знати значення атрибута Address і зможе змінювати його, але клас Company не зможе його ні побачити, ні редагувати. Якщо це знадобиться, він повинен попросити клас Employee переглянути або змінити значення цього атрибута, що зазвичай робиться за допомогою спільних операцій. Закритий атрибут позначається знаком «-» відповідно до нотацією UML.

У *Protected (захищений)***.** Такий атрибут доступний тільки самому класу і його нащадкам. Припустимо, що у нас є два різних типи співробітників - з погодинною оплатою і на окладі. Таким чином, ми отримуємо два інших класу HourlyEmp і SalariedEmp, є нащадками класу Employee. Захищений атрибут Address можна переглянути або змінити з класів Employee, HourlyEmp і SalariedEmp, але не з класу Company. Нотація UML для захищеного атрибуту - це знак «#».

S Package or Implementation (пакетний). Припускає, що даний атрибут є

загальним, але тільки в межах його пакета. Припустимо, що атрибут Address має пакетну видимість. У такому випадку він може бути змінений з класу Company, тільки якщо цей клас знаходиться в тому ж пакеті. Цей тип видимості не позначається ніяким спеціальним значком.

Операції реалізують пов'язане з класом поведінку. Операція включає три частини - *ім'я, параметри і тип значення. Параметри* - це аргументи, одержувані операцією «на вході». Тип значення, що повертається відноситься до результату дії операції. На діаграмі класів можна показувати як імена операцій, так і імена операцій разом з їх параметрами і типом, що повертається. Щоб зменшити завантаженість діаграми, корисно буває на деяких з них показувати лише імена операцій, а на інших їх повну сигнатуру.

Зв'язок являє собою семантичну взаємозв'язок між класами. Вона дає класу можливість дізнаватися про атрибути, операціях і зв'язках іншого класу. Щоб один клас міг послати повідомлення іншому на діаграмі послідовності або кооперативної діаграмі, між ними повинна існувати зв'язок. *Існують чотири типи зв'язків,* які можуть бути встановлені між класами: асоціації, залежності, агрегації і узагальнення.

Асоціація (association) - це семантична зв'язок між класами. їх малюють на діаграмі класів у вигляді звичайної лінії (рис. 2.9). Мовою UML двонаправлені асоціації малюють у вигляді простої лінії без стрілок або зі стрілками з обох її сторін. На односпрямованої асоціації зображують тільки одну стрілку, що показує її напрямок.

Рисунок 2.9- Асоціація

Залежності. Зв'язки залежності (dependency) також відображають зв'язок між класами, але вони завжди однонаправлені і показують, що один клас залежить від визначень, зроблених в іншому (рис. 2.10). Залежності зображують у вигляді стрілки, проведеної пунктирною лінією.

Рисунок 2.10- Залежність

Агрегації (aggregations) являють собою більш тісну форму асоціації. Агрегація - це зв'язок між цілим і його частиною (рис. 2.11). Наприклад, у вас може бути клас Автомобіль, а також класи Двигун, Покришки й класи для інших частин автомобіля. У результаті об'єкт класу Автомобіль буде складатися з об'єкта класу Двигун, чотирьох об'єктів покришок і т. д. Агрегації візуалізують у вигляді лінії з ромбиком у класу, який є цілим:

Рисунок 2.11 - Агрегація

Узагальнення. За допомогою узагальнень (generalization) показують зв'язку спадкування між двома класами (рис. 2.12). Більшість об'єктно- орієнтованих мов безпосередньо підтримують концепцію спадкоємства. Вона дозволяє одному класу успадковувати всі атрибути, операції та зв'язку іншого. Мовою UML зв'язку спадкування називають узагальненнями і зображують у вигляді стрілок від класу-нащадку до класу-предку:

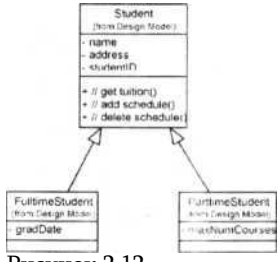

Рисунок 2.12 - Узагальнення

Множинність (multiplicity) показує, скільки примірників одного класу взаємодіють за допомогою цієї зв'язку з одним екземпляром іншого класу в даний момент часу. Наприклад, при розробці системи реєстрації курсів в університеті можна

визначити класи Course (курс) і Student (студент). Між ними встановлено зв'язок: у курсів можуть бути студенти, а у студентів - курси. Питання, на яке повинен відповісти параметр множинності : «Скільки курсів студент може відвідувати в даний момент? Скільки студентів може за раз відвідувати один курс? Так як множинність дає відповідь на обидва ці питання, її індикатори встановлюються на обох кінцях лінії зв'язку. У прикладі реєстрації курсів ми вирішили, що один студент може відвідувати від нуля до чотирьох курсів, а один курс можуть слухати від 10 до 20 студентів. На діаграмі класів це можна зобразити, як показано на рис. 2.13.

Рисунок 2.13 - Множинність

У мові UML прийняті наступні нотації для позначення множинності:

Множинність	Значення
0..*	Нуль або більше
1..*	Один або більше
0..1	Нуль або один
1.. 1 (скорочений запис: 1)	Рівно один

Імена зв'язків. Зв'язки можна уточнити за допомогою імен зв'язків або рольових імен. Ім'я зв'язку - це зазвичай дієслово або дієслівна фраза, що описує, навіщо вона потрібна. Наприклад, між класом Person (людина) і класом Company (компанія) може існувати асоціація (рис. 2.14). Можна задати в зв'язку з цим питання, чи є об'єкт класу Person клієнтом компанії, її співробітником або власником? Щоб визначити це, асоціацію можна назвати «employs» (наймає):

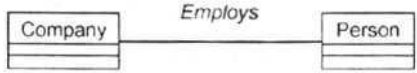

Рисунок 2.14 - Ім'я зв'язку

Імена у зв'язків визначати не обов'язково. Зазвичай це роблять, якщо причина створення зв'язку не очевидна. Ім'я показують близько лінії відповідної зв'язку.

Ролі. Рольові імена застосовують у зв'язках асоціації або агрегації замість імен для опису того, навіщо ці зв'язки потрібні. Повертаючись до прикладу з класами Person і Company, можна сказати, що клас Person грає роль співробітника класу Company. Рольові імена - це зазвичай іменники або засновані на них фрази, їх показують на діаграмі поруч з класом, граючим відповідну роль (рис. 2.15). Як правило, користуються або рольових ім'ям, або ім'ям зв'язку, але не обома відразу. Як і імена зв'язків, рольові імена не обов'язкові, їх дають, тільки якщо мета зв'язку не очевидна. Приклад ролей наводиться нижче:

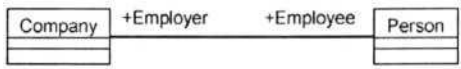

Рисунок 2.15 - Рольові імена

2.2.4 Діаграми станів

Діаграми станів визначають всі можливі стани, в яких може перебувати конкретний об'єкт, а також процес зміни станів об'єкта в результаті настання деяких подій. Існує багато форм діаграм станів, незначно відрізняються один від одного семантикою. Найбільш поширена форма, використовувана в об'єктно-орієнтованих методах, вперше застосовувалася в методі OMT і згодом була адаптована Граді Бучем. На рис. 2.16 наводиться приклад діаграми станів для банківського рахунку. З даної діаграми видно, в яких станах може існувати рахунок. Можна також бачити процес

переходу рахунки з одного стану в інший. Наприклад, якщо клієнт вимагає закрити відкритий рахунок, він переходить в стан «Закрито». Вимога клієнта називається подією (event), саме такі події і викликають перехід з одного стану в інший. Якщо клієнт знімає гроші з відкритого рахунку, він може перейти в стан «Перевищення кредиту». Це відбувається, тільки якщо баланс за цим рахунком менше нуля, що відображено умовою [негативний баланс] на нашій діаграмі. Укладена в квадратних дужках умова (guard condition) визначає, коли може або не може відбутися перехід з одного стану в інший.

На діаграмі є два спеціальних стану - *початкове (start) і кінцеве (stop)***.** Початковий стан виділено чорною точкою, воно відповідає стану об'єкта, коли він тільки що був створений. Кінцеве стан позначається чорною точкою в білому кружку, воно відповідає стану об'єкта безпосередньо перед його знищенням. На діаграмі станів може бути одне і тільки одне початкова стан. Водночас, може бути стільки кінцевих станів, скільки вам потрібно, або їх може не бути взагалі. Коли об'єкт знаходиться в якомусь конкретному стані, можуть виконуватися різні процеси. У нашому прикладі при перевищенні кредиту клієнту надсилається відповідне повідомлення. Процеси, що відбуваються, коли об'єкт знаходиться в певному стані, називаються діями (actions).

Рисунок 2.16 - Діаграма станів для класу Account

Зі станом можна пов'язувати дані п'яти типів: діяльність, вхідна дія, вихідна дія, подія і історія стану. Розглянемо кожен з них в контексті діаграми станів для класу Account

системи АТМ.

Діяльністю (activity) називається поведінка, що реалізовується об'єктом, поки він знаходиться в даному стані. Наприклад, коли рахунок знаходиться в стані «Закрито», відбувається повернення кредитної картки користувачеві. Діяльність - це переривається поведінку. Воно може виконуватися до свого завершення, поки об'єкт знаходиться в даному стані, або може бути перервано переходом об'єкта в інший стан. Діяльність зображують всередині самого стану, їй має передувати слово do (робити) і двокрапка.

Вхідною дією (entry action) називається поведінка, яка виконується, коли об'єкт переходить в даний стан. У прикладі рахунку в банку, коли він переходить в стан «Перевищено рахунок», виконується дія «Тимчасово заморозити рахунок», незалежно від того, звідки об'єкт перейшов у цей стан. Таким чином, дана дія здійснюється не після того, як об'єкт перейшов в цей стан, а, швидше, як частина цього переходу. На відміну від діяльності, вхідна дія розглядається як небезперервна.

Вихідна дія (exit action) подібно вхідного. Однак, воно здійснюється як складова частина процесу виходу з даного стану. У нашому прикладі при виході об'єкта Account зі стану «Перевищено рахунок», незалежно від того, куди він переходить, виконується дія «Розморозити рахунок». Воно є частиною процесу такого переходу. Як і вхідний, вихідний дія є небезперервна. Вихідна дія зображують всередині стану, йому передує слово exit (вихід) і двокрапка.

Переходом (Transition) називається переміщення з одного стану в інший. Сукупність переходів діаграми показує, як об'єкт може переміщатися між своїми станами. На діаграмі всі переходи зображують у вигляді стрілки, що починається на первісному стані і закінчується наступним. Переходи можуть бути рефлексивними. Об'єкт може перейти в той же стан, в якому він зараз перебуває. Рефлексивні переходи зображують у вигляді стрілки, що починається і завершується на одному і тому ж стані. У переході існує кілька специфікацій. Вони включають події, аргументи, огороджувальні умови, дії та їх посилають події. Розглянемо кожне з них у контексті

прикладу АТМ.

Подія (event) - це те, що викликає перехід з одного стану в інший. У нашому прикладі подія «Клієнт вимагає закрити» викликає перехід рахунки з відкритого в закритий стан. Подія розміщують на діаграмі уздовж лінії переходу. На діаграмі для відображення події можна використовувати як ім'я операції, так і звичайну фразу. У нашому прикладі події описані звичайними фразами. Якщо ви хочете використовувати операції, то подія «Клієнт вимагає закрити» можна було б назвати RequestCloser ().

Огороджувальні умови (guard conditions) визначають, коли перехід може, а коли не може здійснитися. У нашому прикладі подія «Зробити внесок» переведе рахунок зі стану «Перевищення рахунки» в стан «Відкрито», але тільки якщо баланс буде більше нуля. В іншому випадку перехід не здійсниться.

Дією (action), як уже говорилося, є безперервна поведінка, що здійснюється як частина переходу. Вхідні і вихідні дії показують всередині станів, оскільки вони визначають, що відбувається, коли об'єкт входить або виходить з нього. Більшу частину дій, однак, зображують уздовж лінії переходу, так як вони не повинні здійснюватися при вході або виході зі стану. Подія або дія можуть бути поведінкою всередині об'єкта, а можуть являти собою повідомлення, що посилається іншому об'єкту. Якщо подія або дія надсилається іншому об'єкту, перед ним на діаграмі поміщають знак «^Л».

Діаграми станів не треба створювати для кожного класу, вони застосовуються лише у складних випадках.

2.2.5 Діаграми діяльності

На відміну від більшості інших засобів UML, діаграми діяльностей н|, мають явно вираженого джерела в попередніх роботах Буча, Рамбо і Якобсона, і запозичують ідеї з декількох різних методів, зокрема, методу моделювання станів SDL і мереж Петрі. Ці

діаграми особливо корисні в описі поведінки, що включає велику кількість паралельних процесів [Фаулер-1999]. Подібно більшості інших засобів, що моделюють поведінку, діаграми діяльності володіють певними перевагами і недоліками, тому їх краще всього використовувати у поєднанні з іншими засобами.

Найбільшою перевагою діаграм діяльності є підтримка паралелізму. Завдяки цьому вони є потужним засобом моделювання потоків робіт і, по суті, паралельного програмування. *Найбільший їх недолік* полягає в тому, що зв'язки між діями і об'єктами проглядаються не дуже чітко.

Ці зв'язки можна спробувати визначити, використовуючи для діяльності мітки з іменами об'єктів, але цей спосіб не володіє такою ж простою безпосередністю, як у діаграм взаємодії. *Діаграми діяльності переважне використовувати в таких ситуаціях:*

^ Аналіз варіанту використання. На цій стадії нас не цікавить зв'язок між діями і об'єктами, а потрібно тільки зрозуміти, які дії повинні мати місце і які залежності в поведінці системи. Зв'язування методів і об'єктів виконується пізніше за допомогою діаграм взаємодії.

^ Аналіз потоків робіт (workflow) в різних варіантах використання. Коли варіанти використання взаємодіють один з одним, діаграми діяльності є потужним засобом представлення та аналізу їх поведінки.

2.2.6 Діаграми компонентів

Діаграми компонентів показують, як виглядає модель на фізичному рівні. На них зображені компоненти програмного забезпечення та зв'язку між ними. При цьому на такий діаграмі виділяють два типи компонентів: виконувані компоненти і бібліотеки коду. Кожен клас моделі (або підсистема) перетворюється на компонент вихідного коду. Після створення вони відразу

додаються до діаграми компонентів. Між окремими компонентами зображують залежності, відповідні залежностям на етапі компіляції або виконання програми. На рис. 2.17 зображена одна з діаграм компонентів для системи ATM.

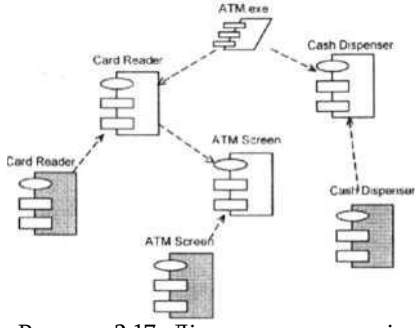

Рисунок 2.17- Діаграма компонентів для клієнта ATM

На цій діаграмі показані компоненти клієнта системи ATM. У даному випадку система розробляється на мові C + +. У кожного класу є свій власний заголовний файл і файл з розширенням. CPP, так що кожен клас перетвориться в свої власні компоненти на діаграмі. Наприклад, клас ATM Screen перетвориться в компонент ATM Screen діаграми. Він перетворюється також і в другій компонент ATM Screen. Разом ці два компоненти представляють тіло і заголовок класу ATM Screen. Виділений темним компонент називається специфікацією пакету (package specification) і відповідає файлу тіла класу ATM Screen мовою C + + (файл з розширенням. CPP). Невиділений компонент також називається специфікацією пакету, але відповідає заголовні файли класу мови C + + (файл з розширенням. H). Компонент ATM. exe є специфікацією завдання і представляє потік обробки інформації (thread of processing). У даному випадку потік обробки є виконуваною програмою, омпоненти з'єднані штриховий лінією, що відповідає залежностям між ними. Наприклад, клас Card Reader залежить від класу ATM Screen. Це означає, що, для того, щоб клас Card Reader міг бути скомпільований, клас ATM Screen повинен вже існувати. Після компіляції всіх класів може бути створений виконуваний файл ATMClient.exe. Приклад ATM містить

два потоки обробки і, таким чином, виходять два виконуваних файлу. Один з них - це клієнт ATM, він містить компоненти Cash Dispenser, Card Reader і ATM Screen. Другий файл - це сервер ATM, що включає в себе компонент Account Діаграма компонентів для сервера ATM показана на рис. 2.18. Діаграми компонентів застосовуються тими учасниками проекту, хто відповідає за компіляцію системи. З неї видно, в якому порядку треба компілювати компоненти, а також які виконувані компоненти будуть створені системою. На такий діаграмі показано відповідність класів реалізованим компонентам. Вона потрібна там, де починається генерація коду.

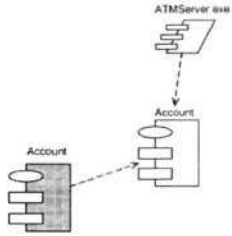

Рисунок 2.18 - Діаграма компонентів для сервера ATM

2.2.7 Діаграма розміщення

Діаграма розміщення (deployment diagram) відображає фізичні взаємозв'язки між програмними і апаратними компонентами системи. Вона є хорошим засобом для того, щоб показати маршрути переміщення об'єктів і компонентів в розподіленої системі. Кожен вузол на діаграмі розміщення являє собою певний тип обчислювального пристрою - у більшості випадків, частина апаратури. Ця апаратура може бути простим пристроєм або датчиком, а може
бути і мейнфреймом. Діаграма розміщення показує фізичне розташування мережі та місцезнаходження у ній різних компонентів. У нашому прикладі система ATM складається з великої кількості підсистем, виконуваних на окремих фізичних пристроях, або вузлах (node). Діаграма розміщення для системи ATM показана на рис.

2.19. З даної діаграми можна дізнатися про фізичне розміщення системи.

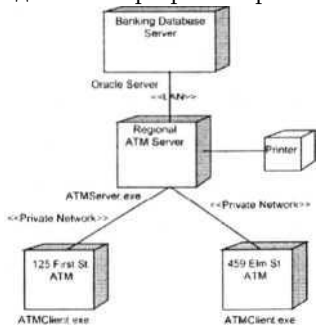

Рисунок 2.19 - Діаграма розміщення для системи ATM

Діаграма розміщення використовується менеджером проекту, користувачами, архітектором системи і експлуатаційним персоналом, щоб зрозуміти фізичне розміщення системи і розташування її окремих підсистем.

2.3 Контрольні питання

1. Що таке уніфікована мова моделювання (UML)?
2. Які видів діаграм відомі?
3. Що таке варіант використання?
4. Що таке дійова особа (actor)?
5. Що описують діаграми взаємодії, послідовності, класів, станів, діяльності, компонентів, розміщення та кооперативні діаграми?

Тема 3. Технології створення програмного забезпечення

3.1 Визначення технології створення програмного забезпечення

ТС ПЗ - упорядкована сукупність взаємозв'язаних технологічних процесів у рамках ЖЦ ПЗ. *Технологічний процес* - сукупність взаємопов'язаних технологічних операцій. *Технологічна операція - основна одиниця роботи, яка виконується певною роллю, яка*[4]. 1. Увазі чітко визначену відповідальність ролі. 2. Дає чітко певний результат (набір робітників продуктів), що базується на певних вихідних даних (іншому наборі робочих продуктів). 3. Представляє собою одиницю роботи з жорстко визначеними межами, які встановлюються при плануванні проекту. *Робочий продукт* - інформаційна або матеріальна сутність, яка створюється, модифікується або використовується в деякій технологічної операції (модель, документ, код, тест тощо), визначає область відповідальності ролі і є об'єктом управління конфігурацією. *Роль* - визначення поведінки і обов'язків окремої особи або групи осіб у середовищі організації розробника ПЗ, що здійснюють діяльність у рамках деякого технологічного процесу і відповідальних за певні робочі продукти. *Керівництво* - практичне керівництво з виконання однієї або сукупності технологічних операцій. Керівництва включають методичні матеріали, інструкції, нормативи, стандарти і критерії оцінки якості робочих продуктів. *Інструментальне засіб (САБЕ-засіб)* - програмне засіб, що забезпечує автоматизовану підтримку діяльності, що виконується в рамках технологічних операцій.

3.2 Загальні вимоги, запропоновані до ТС ПЗ

Основною вимогою, що пред'являються до сучасних ТС ПЗ, є їх відповідність стандартам і нормативним документам, пов'язаним з процесами
ЖЦ ПЗ і оцінкою технологічної зрілості організацій-розробників (180 12207, 180 9000, СММ та ін.). *Згідно з цими нормативами ТС ПЗ повинна підтримувати такі*

процеси: управління вимогами; аналіз і проектування ПЗ; розробка ПЗ; експлуатація; супровід; документування; управління конфігурацією і змінами; тестування; управління проектом.

Повнота підтримки процесів ЖЦ ПЗ повинна підтримуватися комплексом інструментальних засобів (CA8E-засобів). *Відповідність стандартам* означає також, зокрема, використання загальноприйнятих, стандартних нотацій і угод. Для того щоб проект міг виконуватися різними колективами розробників, необхідне використання стандартних методів моделювання та стандартних нотацій, які повинні бути оформлені у вигляді нормативів до початку процесу проектування. Недотримання проектних стандартів ставить розробників у залежність від фірми-виробника даного засобу, робить скрутним формальний контроль коректності проектних рішень і знижує можливості залучення додаткових колективів розробників, зміни виконавців і відчуження проекту, оскільки число фахівців, знайомих з даним методом (нотацією), може бути обмеженим.

3.3 Впровадження ТС ПЗ

Процес впровадження ТС ПЗ складається з наступних етапів.

1. Визначення потреб в ТС ПЗ, характеристики об'єкта впровадження та проектів створення ПЗ.
2. Визначення вимог, що пред'являються до ТС ПЗ (аналіз характеристик об'єкта впровадження та проектів, обгрунтування вимог до ТС ПО, визначення пріоритетів вимог).
3. Оцінка варіантів ТС ПЗ. Попередня експертна оцінка полягає в аналізі доступних ТС ПЗ на предмет відповідності вимогам, незадовільні варіанти (з точки зору реалізації найбільш пріоритетних вимог) відкидаються, формується список претендентів.
4. Вибір ТС ПЗ. Виробляється порівняльний аналіз технологій і остаточний вибір

ТС ПЗ за допомогою експертної оцінки.

5. Адаптація ТС ПЗ до умов застосування. Виробляється формування конкретної робочої конфігурації ТС ПЗ, адаптованої до умов об'єкта впровадження.

У процесі впровадження ТС ПЗ збирається статистика і оцінюється ефективність її впровадження з точки зору ряду критеріїв (мінімум трудомісткості супроводу ПЗ, мінімум витрат на супровід ПЗ та ін.) При зміні умов об'єкта впровадження та за результатами аналізу ефективності впровадження ТС ПЗ приймається рішення: а) про внесення змін до робочої конфігурацію ТС ПЗ; б) про перехід на нову ТС ПЗ. У разі переходу повторюються пп. 3, 4, 5.

Процес успішного впровадження ТС ПЗ не обмежується тільки її використанням. Насправді він охоплює планування і реалізацію безлічі технічних, організаційних, структурних процесів, змін в загальній культурі організації і заснований на чіткому розумінні можливостей ТС ПЗ.

Чинники, що ускладнюють визначення можливого ефекту від використання ТС ПЗ:

^ широке розмаїття якості і можливостей ТС ПЗ;

^відносно невеликий час використання ТС ПЗ в різних організаціях і брак досвіду їх застосування;

^ різноманітність практики впровадження ТС ПЗ в різних організаціях;

^ відсутність детальних метрик і даних для вже виконаних і поточних проектів;

^широкий діапазон предметних областей проектів;

^ різна ступінь інтеграції ТС ПЗ в різних проектах.

Успішне впровадження ТС ПЗ повинно забезпечити:

^ високий рівень технологічної підтримки процесів розробки і супроводу ПЗ;

^ позитивний вплив на продуктивність, якість продукції, дотримання стандартів, документування;

^ прийнятний рівень віддачі від інвестицій в ТС ПЗ.

3.4 Визначення потреб у ТС ПЗ

Мета даного етапу (рис. 3.3) - досягнення розуміння потреб організації в ТС ПЗ. Він повинен привести до виділення тих областей діяльності організації, в яких застосування ТС ПЗ може принести реальну користь. Результатом даного етапу є документ, що визначає стратегію впровадження ТС ПЗ.

Рисунок 3.3 - Визначення

потреб в ТС ПЗ **3.**

4.1 Аналіз можливостей організації

Першою дією даного етапу є аналіз можливостей організації щодо її технологічної бази, персоналу і використовуваного ПЗ. Такий аналіз може бути формальним чи неформальним. Формальні підходи визначаються моделлю CMM, а також стандартами 150 9001:1994, ІБО 9003-3:1991 та 150 9004-2:1991. Головне в цих підходах - аналіз різних аспектів відбуваються в організації процесів. Для отримання інформації щодо стану та потреб організації можуть використовуватися неформальні оцінки та анкетування. Перелік запитань, які можуть допомогти у неформальній

оцінці поточної практики використання ПЗ, ТС ПЗ і персоналу, наведено нижче. Оцінка готовності організації до впровадження ТС ПЗ повинна бути об'єктивною і ретельно вивіреною, оскільки в разі відсутності такої готовності всі зусилля з впровадження потерплять крах.

3.4.2 Визначення організаційних потреб та огляд ринку технологій

Організаційні потреби йдуть безпосередньо з проблем організації і цілей, які вона прагне досягти. Проблеми і цілі можуть бути пов'язані з управлінням, процесами, виробництвом продукції, економікою, персоналом чи технологією. Питання, що стосуються визначення цілей, потреб і очікуваних результатів, наведено нижче. Визначення потреб повинно виконуватися в поєднанні з *оглядом ринку технологій*, оскільки інформація про технології, доступних на ринку в даний момент, може вплинути на потреби. Цілі організації відіграють головну роль у визначенні її конкретних потреб і очікуваних результатів. Для *їх розуміння необхідно відповісти на наступні питання:*

^ намір організації використовувати технологію для допомоги в досягненні певних цілей або очікувань (наприклад, певного рівня CMM або сертифікації відповідно до 150 9001);

^ сприйняття ТС ПЗ як чинника, що сприяє досягненню стратегічних цілей організації;

^ наявність у організації власної програми вдосконалення процесу розробки ПЗ;

^ сприйняття ініціативи впровадження ТС ПЗ як частини більш широкомасштабного проекту по створенню середовища розробки ПЗ.

Результатом даної дії є формулювання потреб з їх пріоритетами, яка використовується на етапі оцінки і вибору в якості «користувапьницьких потреб».

3.4.3 Визначення критеріїв успішного впровадження

Обумовлені критерії повинні дозволяти кількісно оцінювати ступінь задоволення кожної з потреб, пов'язаних з впровадженням. Крім того, за кожним критерієм має бути встановлено його конкретне оптимальне значення. На окремих етапах впровадження ці критерії повинні аналізуватися для того, щоб оцінити поточну ступінь задоволення потреб. Як правило, більшість організацій здійснює впровадження ТС ПЗ для підвищення продуктивності процесів розробки і супроводу ПЗ, а також якості результатів розробки. Однак ряд організацій не займається і не займався раніше збором кількісних даних по вказаних параметрах. Відсутність таких даних ускладнює кількісну оцінку впливу, що чиниться впровадженням ТС ПЗ. Для таких організацій рекомендується розробка відповідних метрик. Якщо базові метричні дані відсутні, організація часто може витягти корисну інформацію зі своїх проектних архівів. Крім продуктивності та якості, корисну інформацію про стан впровадження ТС ПЗ також можуть дати і інші характеристики організаційних процесів і персоналу. Наприклад, оцінка ступеня успішності впровадження може включати відсоток проектів, що використовують технологію, рейтингові оцінки рівня кваліфікації фахівців, пов'язані з використанням ТС ПЗ, і результати опитувань персоналу з приводу ставлення до використання ТС ПЗ. ^ Наведемо інші проектні характеристики, які можуть бути оцінені кількісно:

- ^ узгодженість проектних результатів;
- ^ точність вартісних і планових оцінок;
- ^ мінливість зовнішніх вимог;
- ^ дотримання стандартів організації;
- ступінь повторного використання існуючих компонентів ПЗ;
- ^ обсяг і види необхідного навчання;
- типи і моменти виявлення проектних помилок.

3.4.4 Розробка стратегії впровадження ТС ПЗ

Стратегія повинна забезпечувати задоволення визначених раніше потреб і критеріїв.

Дана стратегія визначає: організаційні потреби;

^ базові метрики, необхідні для подальшого порівняння результатів;

^ критерії успішного впровадження, пов'язані із задоволенням організаційних потреб, включаючи очікувані результати послідовних етапів процесу впровадження;

^ підрозділи організації, в яких має виконуватися впровадження ТС ПЗ;

^ вплив, який чиниться на інші підрозділи організації;

^ стратегії і плани оцінки і вибору, пілотного проектування і переходу до повномасштабного впровадження;

^ основні фактори ризику;

^ орієнтовний рівень і джерела фінансування процесу впровадження ТС ПЗ;

^ ключовий персонал та інші ресурси.

Необхідно відзначити, що впровадження нової ТС ПЗ може включати важливі і важкі зміни в культурі організації. Велика увага повинна приділятися ролям різних груп, залучених у процес таких змін. *Найбільш істотними є наступні ролі:*

^ *спонсор* (звичайно з числа менеджерів вищого рівня). Дана роль є критичною для підтримки проекту і забезпечення необхідного фінансування. Спонсор повинен володіти чітким розумінням необхідності серйозних зусиль, пов'язаних з впровадженням ТС ПЗ, і бути готовий до тривалого періоду очікування відчутних результатів;

^ *виконавець* - зазвичай особа (або група осіб), яка усвідомлює потенційні можливості нової ТС ПЗ, яке користується авторитетом серед технічного персоналу і здатне очолити процес впровадження нової ТС ПЗ;

^ *цільова група* - звичайно включає менеджерів і технічний персонал, які будуть залучені до безпосереднього використання ТС ПЗ, а також фахівців, які будуть залучені побічно, таких, як фахівці з документування, персонал підтримки мережі і замовники.

Повинні бути визначені потреби кожної такої групи та план їх ефективного задоволення. У загальному випадку впровадження ТС ПЗ має управлятися і фінансуватися таким же чином, як і будь-який проект розробки ПЗ. Стратегія впровадження може бути переглянута у випадку появи додаткової інформації.

3.5 Критерії оцінки та вибору ТС ПЗ

Критерії формують базис для процесів оцінки і вибору і можуть приймати різні форми:

^ числові заходи в широкому діапазоні значень, наприклад, обсяг необхідних ресурсів;

^ числові заходи в обмеженому діапазоні значень, наприклад простота освоєння, виражена в балах від 1 до 5;

^ двійкові міри (істина / неправда, так / ні), наприклад здатність генерації документації в заданому форматі;

^ заходи, які можуть приймати одне значення або більше з кінцевих множин значень, наприклад платформи, для яких підтримується ТС ПЗ.

Фактори вибору ТС ПЗ:

^ характеристики об'єкта впровадження, що визначають вимоги, що пред'являються до ТС ПЗ;

^ параметри доступних ТС ПЗ;

^ ресурси проекту (фінансові, кадрові та технічні).

Вихідні дані для вибору та оцінки застосовності - набір параметрів (техніко-

економічних характеристик) ТС ПЗ (рис. 3.4).

Рисунок 3.4 - Сукупність параметрів

ТЕХ ТС ПЗ

3.6 Виконання пілотного проекту

Пілотний проект являє собою первісне реальне використання ТС ПЗ в призначеній для цього середовищі і зазвичай має на увазі більш широкий масштаб використання ТС ПЗ по відношенню до того, який був досягнутий під

час оцінки. Пілотний проект повинен володіти багатьма характеристиками реальних проектів, для яких призначений даний засіб. *Він переслідує такі цілі:*

^ підтвердити достовірність результатів оцінки та вибору;

^ встановити, чи дійсно ТС ПО годиться для використання в даній організації, і якщо так, то визначити найбільш підходящу область її застосування;

^ зібрати інформацію, необхідну для розробки плану практичного впровадження;

^ придбати власний досвід використання ТС ПЗ.

Пілотний проект дозволяє отримати важливу інформацію, необхідну для оцінки ТС ПЗ і його підтримки з боку постачальника після того, як засіб встановлено. Важливою функцією пілотного проекту є прийняття рішення щодо придбання чи відмови від використання ТС ПЗ. Провал пілотного проекту дозволяє уникнути більш значних і дорогих невдач надалі, оскільки пілотний проект зазвичай вимагає придбання відносно невеликої кількості ліцензій та навчання вузького кола фахівців. Первісне використання нової ТС ПЗ в пілотному проекті має ретельно плануватися і контролюватися. Пілотний проект включає наступні кроки (рис. 3.5).

Рисунок 3.5 - Кроки пілотного проекту

3.6.1 Характеристики пілотного проекту

Пілотний проект повинен мати такими характеристиками:

Типовість предметної області. Щоб полегшити остаточне визначення області застосування ТС ПЗ, предметна область пілотного проекту повинна бути типовою для звичайної діяльності організації.

Масштабованість. Результати, отримані в пілотному проекті, повинні показати масштабованість ТС ПЗ. Мета - отримати чітке уявлення про масштаби проектів, для яких дана ТС ПЗ застосовна.

Показність. Пілотний проект не повинен бути незвичайним або унікальним для організації. ТС ПЗ повинна використовуватися для вирішення завдань, що відносяться до предметної області, добре розуміється всією організацією.

Критичність. Пілотний проект повинен мати істотну значущість, щоб опинитися в центрі уваги, але не повинен бути критичним для успішної діяльності організації в цілому.

Авторитетність. Група фахівців, що беруть участь у проекті, повинна володіти високим авторитетом, при цьому результати проекту будуть всерйоз сприйняті іншими співробітниками організації.

Готовність проектної групи. Проектна група повинна володіти готовністю до нововведень, технічної зрілістю і прийнятним рівнем досвіду і знань у даній ТС ПЗ і предметної області.

3.6.2 Планування пілотного проекту

Планування пілотного проекту має по можливості вписуватися в звичайний процес планування проектів в організації. План повинен містити
наступну інформацію: цілі, завдання і критерії оцінки; персонал; процедури та угоди;

навчання; графік і ресурси. *Цілі, завдання та критерії оцінки*

Очікувані результати пілотного проекту повинні бути чітко визначені.

3.6.3 Особливості пілотного проекту

Дуже важливо провести аналіз пілотного проекту з тим, щоб визначити його елементи, які є критичними для успіху, і ступінь відображення цими елементами організації в цілому. *Відзначимо найважливіші характеристики пілотного проекту, що не є представницькими для організації в цілому:*

^ процеси в пілотному проекті в чому-небудь відрізняються від процесів у всій організації;

кваліфікація групи пілотного проекту не відображає кваліфікацію інших фахівців організації;

^ ресурси, виділені на виконання проекту, можуть відрізнятися від тих, які виділяються для звичайних проектів;

^ предметна область або масштаб проекту можуть відрізнятися від інших проектів.

3.7 Вигода від використання ТС ПЗ

Результати пілотного проекту слід порівняти з можливостями організації в цілому. *Варіанти рішення про впровадження. Можливим рішенням має бути одне з наступних:*

^ Впровадити ТС ПЗ. У цьому випадку рекомендований масштаб впровадження повинен бути визначений в термінах структурних підрозділів та предметної області.

^ Виконати додатковий пілотний проект. Такий варіант має розглядатися тільки в тому випадку, якщо залишилися конкретні невирішені питання щодо впровадження ТС ПЗ в організації. Новий пілотний проект повинен бути таким, щоб відповісти на ці питання.

^ Відмовитися від ТС ПЗ. У цьому випадку причини відмови від конкретної ТС ПЗ повинні бути визначені в термінах потреб організації або критеріїв, які залишилися незадоволеними.

^ Відмовитися від використання ТС ПЗ взагалі. Пілотний проект може показати, що організація або не готова до впровадження ТС ПЗ, або автоматизація даного аспекту процесу створення і супроводу ПЗ не дає ніякого ефекту для організації.

3.8 Контрольні питання

1. Що таке робочий продукт?
2. Які основні визначення потреб у ТС ПЗ?
3. Що таке технологія створення ПЗ?
4. Що таке інструментальне засіб?
5. Які критерії оцінки та вибору ТС ПЗ?
6. Поняття пілотного проекту.
7. Які основні характеристики та особливості пілотного проекту?

Тема 4. Основні відомості про CASE-засіб Rational Rose

4.1 Введення в Rational Rose

Rational Rose - сімейство об'єктно-орієнтованих CASE-засобів фірми Rational Software Corporation - призначено для автоматизації процесів аналізу і проектування ПЗ, а також для генерації кодів на різних мовах і випуску проектної документації. Rational Rose використовує метод об'єктно- орієнтованого аналізу і проектування, заснований на мові UML. Поточна версія Rational Rose реалізує генерацію кодів програм для C++, Visual C + +, Visual Basic, Java, PowerBuilder, CORBA Interface Definition Language (IDL), генерацію описів баз даних для ANSI SQL, Oracle, MS SQL Server, IBM DB2, Sybase, а також дозволяє розробляти проектну документацію у вигляді діаграм і специфікацій. Крім того, Rational Rose містить засоби реверсного інжинірингу програм і баз даних, що забезпечують повторне використання програмних компонентів в нових проектах.

Структура і функції. В основі роботи Rational Rose лежить побудова діаграм і специфікацій UML, що визначають архітектуру системи, її статичні і динамічні аспекти. У складі Rational Rose можна виділити шість основних структурних компонентів: репозиторій, графічний інтерфейс користувача, засоби перегляду проекту (браузер), засоби контролю проекту, засоби збору статистики і генератор документів. До них додаються генератор кодів (індивідуальний для кожної мови) і аналізатор для C++, що забезпечує реверсний інжиніринг. *Репозиторій* являє собою базу даних проекту. Браузер забезпечує "навігацію" за проектом, в тому числі переміщення по ієрархіям класів і підсистем, переключення від одного виду діаграм до іншого і т. д. Засоби контролю та збору статистики дають можливість знаходити і усувати помилки у міру розвитку проекту, а не після завершення його опису. Генератор звітів формує тексти вихідних документів на основі міститься в репозиторії інформації. Засоби автоматичної генерації кодів програм мовою C++, використовуючи інформацію, що міститься в діаграмах класів і компонентів,

формують файли заголовків і файли описів класів та об'єктів. Створюваний таким чином скелет програми може бути уточнений шляхом прямого програмування мовою C++. Аналізатор кодів C++ реалізований у вигляді окремого програмного модуля. Його призначення - створювати модулі проектів Rational Rose на основі інформації, що міститься в визначених користувачем вихідних текстах на C++. У процесі роботи аналізатор здійснює контроль правильності вихідних текстів і діагностику помилок. Модель, отримана в результаті його роботи, може цілком або фрагментарно використовуватися в різних проектах. Аналізатор має широкі можливості настройки по входу і виходу. Таким чином, Rational Rose / C++ забезпечує можливість повторного використання програмних компонентів. *В результаті розробки проекту за допомогою CASE-засоби Rational Rose формуються наступні документи:*

- Діаграмі UML, в сукупності представляються собою модель розроблюваної програмної системи;
- Специфікації класів, об'єктів, атрибутів і операцій;
- Заготовки текстів програм.

Взаємодія з іншими засобами і організація групової роботи. Для підтримки командної роботи над проектом на кожній стадії життєвого циклу ПЗ є інтегрований набір продуктів Rational Suite. *Rational Suite існує в наступних варіантах:*

- Rational Suite AnalystStudio - призначений для визначення та управління повним набором вимог до розроблюваної системі;
- Rational Suite DevelopmentStudio - призначений для проектування і реалізації ПЗ;
- Rational Suite TestStudio - являє собою набір продуктів, призначених для автоматичного тестування додатків;
- Rational Suite Enterprise - забезпечує підтримку повного життєвого циклу ПЗ і призначений як для менеджерів проекту, так і окремих розробників, що

виконують кілька функціональних ролей в команді розробників.

До складу Rational Suite, крім Rational Rose, входять наступні компоненти:

^ Rational Requisite Pro - засіб управління вимогами, призначене для організації спільної роботи групи розробників. Воно дозволяє команді розробників створювати, структурувати, встановлювати пріоритети, відстежувати, контролювати зміни вимог, що виникають на будь-якому етапі розробки компонентів додатка;

^ Rational ClearCase - засіб управління конфігурацією ПЗ;

^ Rational SoDA - засіб автоматичної генерації проектної документації;

^ Rational ClearQuest - засіб для управління змінами та відстеження дефектів в проекті на основі засобів e-mail і Web;

^ Rational TeamTest - засіб автоматичного виявлення помилок під час виконання програми і генерації сценаріїв для проведення регресійного тестування;

^ Rational Robot - засіб для створення, модифікації і автоматичного запуску тестів;

^ Rational Purify - засіб для локалізації важко виявлених помилок часу виконання програми;

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

^ Rational Quantify - засіб кількісного визначення вузьких місць, що впливають на загальну ефективність роботи програми;

^ Rational Suite Performance Studio - засіб навантажувального тестування додатків «клієнт-сервер» і Web-додатків.

Середа функціонування. Rational Rose функціонує на різних платформах: IBM PC (Windows 95/98/NT), Sun SPARCstations (UNIX, Solaris, SunOS), Hewlett-Packard (HP UX), IBM RS/6000 (AIX).

4.2 Робота в середовищі Rational Rose

Елементи екрана. На рис. 4.1 показані різні частини інтерфейсу Rose. П'ять основних елементів інтерфейсу Rose - це браузер, вікно документації, панелі інструментів, вікно діаграми і журнал (log). їх призначення полягає в наступному:

^ браузер (browser) - використовується для швидкої навігації по моделі;

^ вікно документації (documentation window) - застосовується для роботи з текстовим описом елементів моделі;

^ панелі інструментів (toolbars) - застосовуються для швидкого доступу до найбільш поширених командам;

^ вікно діаграми (diagram window) - використовується для перегляду і редагування однієї або декількох діаграм UML;

^ журнал (log) - застосовується для перегляду помилок і звітів про результати виконання різних команд.

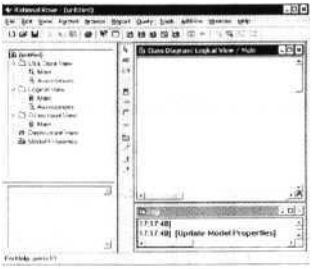

Рисунок 4.1 - Інтерфейс Rational Rose

Браузер **-** це ієрархічна структура, що дозволяє здійснювати навігацію по моделі. Все, що додається в модель - дійові особи, варіанти використання, класи, компоненти - буде показано у вікні браузера. Браузер підтримує чотири вистави (view): подання варіантів використання, компонентів, розміщення і логічне уявлення. Браузер організований у деревовидному стилі. Кожен елемент моделі може містити інші елементи, що знаходяться нижче його в ієрархії. Знак «-» близько елемента означає, що його гілка повністю розкрита. Знак «+» - що його гілка згорнута.

Вікно документації. З його допомогою можна документувати елементи моделі Rose. Наприклад, можна зробити короткий опис кожної дійової особи. При документуванні класу все, що буде написано у вікні документації, з'явиться потім у вигляді коментаря в сгенерированном коді, що позбавляє від необхідності згодом вносити ці коментарі вручну. Документація буде виводитися також у звітах, що створюються в середовищі Rose.

Панелі інструментів Rose забезпечують швидкий доступ до найбільш поширених командам. У цьому середовищі існує два типи панелей інструментів: стандартна панель і панель діаграми. Стандартна панель видна завжди, її кнопки відповідають командам, які можуть використовуватися для роботи з будь діаграмою. Панель діаграми своя для кожного типу діаграм UML.

Вікно діаграми. У вікні діаграми видно, як виглядає одна або кілька діаграм UML моделі. При внесенні в елементи діаграми змін Rose автоматично оновить браузер. Аналогічно, при внесенні змін до елемент за допомогою браузера Rose автоматично оновить відповідні діаграми. Це допомагає підтримувати модель в несуперечливою стані.

Журнал. Принаймні роботи над вашою моделлю певна інформація буде спрямовуватися у вікно журналу. Наприклад, туди поміщаються повідомлення про помилки, що виникають при генерації коду. Не існує способу закрити журнал зовсім, але його вікно може бути мінімізовано.

4.3 Чотири представлення моделі Rose

У моделі Rose підтримується чотири представлення (views) представлення варіантів використання, логічне подання представлення компонентів і уявлення розміщення. Кожне з них призначено для своїх цілей і для відповідної аудиторії. У наступних розділах цієї глави ми коротко розглянемо кожне з зазначених подань, а решти книги детально обговоримо містяться в них елементи моделі.

4.3.1 Представлення варіантів використання

Це подання містить усіх дійових осіб, всі варіанти використання і їх діаграми для конкретної системи. Воно може також містити деякі діаграми послідовності і кооперативні діаграми. На рис. 4.3 показано, як виглядає уявлення варіантів використання в браузері Rose.

Рисунок 4.3 - Представлення варіантів використання

Представлення варіантів використання містить:

- Діючих осіб.
- Варіанти використання.
- Документацію за варіантами використання, що деталізує відбуваються в них процеси (потоки подій), включаючи обробку помилок. Піктограмами зображуються

зовнішні файли, прикріплені до моделі Rose. Вид піктограми, залежить від програми, що використовується для документування потоку подій.

^ Діаграми варіантів використання. Зазвичай у системи буває кілька таких діаграм, кожна з яких показує підмножина дійових осіб та/або варіантів використання

^ Пакети, які є групами варіантів використання і/або дійових осіб.

4.3.2 Логічне представлення

Логічне представлення, показане на рис. 4.4, концентрується на тому, як система буде реалізовувати поведінку, описане у варіантах використання. Воно дає докладну картину складових частин системи і описує взаємодію цих частин. Логічне уявлення включає, крім іншого, конкретні необхідні класи, діаграми класів і діаграми станів. З їх допомогою конструюється детальний проект створюваної системи. *Логічне представлення містить:*

^ Класи.

^ Діаграми класів. Як правило, для опису системи використовується кілька діаграм класів, кожна з яких відображає деяку підмножину всіх класів системи.

^ Діаграми взаємодії, застосовувані для відображення об'єктів, що беруть участь в одному потоці подій варіанту використання.

^ Діаграми станів.

^ Пакети, які є групами взаємопов'язаних класів.

Рисунок 4.4 — Логічне

представлення системи

4.3.3 Представлення компонентів

Представлення компонентів містить:

^ Компоненти, є фізичними модулями коду.

^ Діаграми компонентів.

^ Пакети, які є групами пов'язаних компонентів.

4.3.4 Представлення розміщення

Останнє представлення Rose - це подання розміщення. Воно відповідає фізичному розміщення системи, яке може відрізнятися від її логічної архітектури. *В представлення розміщення входять: 1.* Процеси, які є потоками (threads), виконуваними у відведеній для них області пам'яті. 2. Процесори, що включають будь-які комп'ютери, здатні обробляти дані. Будь-який процес виконується на одному або декількох процесорах. 3. Пристрої, тобто будь-яка апаратура, не здатна

обробляти дані. До числа таких пристроїв належать, наприклад, термінали вводу-виводу і принтери. 3. Діаграма розміщення.

4.4 Параметри налаштування відображення

У *Rose є можливість налаштувати діаграми класів так, щоб:* Показувати всі атрибути і операції; Приховати операції; Приховати атрибути; Показувати тільки деякі атрибути або операції; Показувати операції разом з їх повними сигнатурами або тільки їх імена; Показувати чи не показувати видимість атрибутів і операцій; Показувати чи не показувати стереотипи атрибутів і операцій.

Значення кожного параметра за замовчуванням можна задати за допомогою вікна, що відкривається при виборі пункту меню Tools>Options. У *даного класу на діаграмі можна: 1.* Показати всі атрибути. 2. Приховати всі атрибути. 3. Показати тільки вибрані вами атрибути. 4. Придушити висновок атрибутів.

Придушення виведення атрибутів призведе не тільки до зникнення атрибутів з діаграми, а й до видалення лінії, що показує місце розташування атрибутів в класі. Існує два способи зміни параметрів подання атрибутів на діаграмі. Можна встановити потрібні значення у кожного класу індивідуально. Можна також змінити значення потрібних параметрів за замовчуванням до початку створення діаграми класів. Внесені таким чином зміни вплинуть тільки на новостворювані діаграми.

4.5 Контрольні питання

1. Яка структура і функції Rational Rose?
2. Що відноситься до основних елементів інтерфейсу Rational Rose?
3. Які основні представлення Rational Rose?

Тема 5. Правила розробки програмного забезпечення

5.1 Початок проектування

Проектування ПЗ є досить серйозною дисципліною, яка вивчає всі аспекти створення та просування ПЗ. Фахівці, які займаються проектуванням архітектури ПЗ, мають за своїми плечима досвід не одного року написання того чи іншого «живого» коду. Вони пройшли через величезні перешкоди всіляких помилок функціоналу їх додатків, а також зустріли достатню кількість різних технологій і платформ розробки. Ці люди справжні профі своєї справи, які по праву заслуговують назву для своєї професії як Архітектори ПЗ. Під початком проектування мається на увазі розгляд технічного завдання (ТЗ), що надається замовником і подальше складання, на базі отриманого завдання, моделі бізнес-логіки додатка, структури даних і їх відображення.

5.2 Розгляд технічного завдання

Розробка програмного забезпечення (англ. software engineering, software development) - це рід діяльності (професія) і процес, спрямований на створення і підтримку працездатності, якості та надійності ПЗ, використовуючи технології, методологію та практики з інформатики, управління проектами, математики, інженерії та інших областей знання. *Якщо розглядати структуру ТЗ детальніше, то можна виявити наступні ключові пункти:*

I. Функціональні призначення (або область застосування) - розкривається одна з особливостей використання програми, а саме середовище її роботи. То може бути інтернет-послуги, машинні апарати, телефони та інше.

2. Функціональні вимоги до самої програми - розписує те, що повинна робити програма. Наприклад, показувати прогнози погоди, приймати телефонні дзвінки та інше.

3. Необхідні вимоги до складу і параметрів внутрішнього середовища, а також їх надійність і безпеку **-** мається на увазі наявність певних складових всього ПЗ і те, яким вони повинні відповідати нормам безпеки і надійності. Нехай то, наприклад, метал або пластик, операційна система Linux або Windows, мова програмування Java або PHP. Ну і так далі.

4. Необхідні вимоги до сумісності - вказує на облік необхідних сумісностей ПЗ з іншими сферами роботи. Дані параметри повинні бути враховані при етапі проектування і розробки.

5. Документація - завершує весь етап створення запитаного ПЗ. Тут створюється необхідна документація для користувачів і для розробників, які змогли б продовжити подальшу модернізацію.

Виконання цих та інших пунктів підпорядковуються певним стандартам, за якими, у свою, чергу, створено відповідні специфікації. У двох словах, специфікація - це набір вимог і параметрів, яким повинна задовольняти розглянута сутність (предмет, область, сфера). Так само специфікація представляється собою якийсь список перерахованих умов, що задовольняють виробничий замовлення.

Після розбору ТЗ, відбувається етап обліку аналізу всіх витрат на виконання необхідних вимог. Розглядаються всі відповідні інструменти і платформи для реалізації пунктів з ТЗ. Після прийняття рішення про той чи інший вибір, дається завдання архітектору на планування організації побудови архітектури всього вихідного коду ПЗ. Тут можна виділити поняття «чорного ящика», яке має на увазі під собою аналіз зовнішніх характеристик програми, без втручання в її структуру. Далі починається докладний розгляд внутрішньої начинки проектованої системи з деталізацією всіх її зовнішніх властивостей.

Це все і є початковий етап проектування, де дуже важливо грамотно продумати всі ланцюжки дій з виконання ТЗ. При помилці в одній ланці, руйнується весь структурний ланцюжок, а це потягне за собою перегляд багатьох важливих питань.

5.3 Проблеми і правила розробки ПЗ

Найбільш поширеними проблемами, що виникають в процесі розробки ПЗ, вважають:

^ *Недолік прозорості.* У будь-який момент часу складно сказати, в якому стані знаходиться проект і який відсоток його завершення. Дана проблема виникає при недостатньому плануванні структури (чи архітектури) майбутнього програмного продукту, що найчастіше є наслідком відсутності достатнього фінансування проекту: програма потрібна, скільки часу займе розробка, якими є етапи, чи можна якісь етапи виключити або заощадити - наслідком цього процесу є те, що етап проектування скорочується.

^ *Недолік контролю.* Без точної оцінки процесу розробки зриваються графіки виконання робіт і перевищуються встановлені бюджети. Складно оцінити обсяг виконаної роботи і тієї, що залишилась. Дана проблема виникає на етапі, коли проект, завершений більш ніж наполовину, продовжує розроблятися після додаткового фінансування без оцінки ступеня завершеності проекту.

^ *Недолік моніторингу.* Неможливість спостерігати хід розвитку проекту не дозволяє контролювати хід розробки в реальному часі. За допомогою інструментальних засобів менеджери проектів приймають рішення на основі даних, що надходять в реальному часі. Дана проблема виникає в умовах, коли вартість навчання менеджменту володінню інструментальними засобами порівнянна з вартістю розробки самої програми.

^ *Неконтрольовані зміни.* У споживачів постійно виникають нові ідеї щодо розроблюваного програмного забезпечення. Вплив змін може бути суттєвим для успіху проекту, тому важливо оцінювати пропоновані зміни та реалізовувати тільки схвалені, контролюючи цей процес за допомогою програмних засобів. Дана проблема виникає внаслідок небажання кінцевого споживача використовувати ті чи інші програмні середовища. Наприклад, коли при створенні клієнт- серверної системи споживач висуває вимоги не тільки до операційної системи на комп'ютерах

- клієнтах, а й на комп'ютері - сервері.

^ *Недостатня надійність.* Найскладніший процес - пошук і виправлення помилок у програмах на ЕОМ. Оскільки число помилок у програмах заздалегідь невідомо, то заздалегідь невідома і тривалість налагодження програм і відсутність гарантій відсутності помилок в програмах.

^ Відсутність гарантій якості і надійності програм через відсутність гарантій відсутності помилок в програмах аж до формальної здачі програм замовникам. Дана проблема не є проблемою, що відноситься виключно до розробки ПЗ. Гарантія якості - це проблема вибору постачальника товару (продукту).

5.4 Контрольні питання

1. Які основні правила розробки програмного забезпечення?
2. Поняття технічного завдання.
3. Які ключові пункти технічного завдання?
4. Що таке розробка програмного забезпечення?
5. Найбільш поширеними проблемами, що виникають в процесі розробки ПЗ є?

Тема 6. Еволюція моделей життєвого циклу програмного забезпечення 6.1

6.1 Класичний життєвий цикл

Старою парадигмою процесу розробки ПЗ є класичний життєвий цикл (автор Уїнстон Ройс, 1970). Дуже часто класичний життєвий цикл називають каскадною або водопадом моделлю, підкреслюючи, що розробка розглядається як послідовність етапів, причому перехід на наступний, ієрархічно нижній етап відбувається тільки після повного завершення робіт на поточному етапі (рис. 6.1). Охарактеризуємо зміст основних етапів.

Рисунок 6.1 - Класичний життєвий цикл розробки ПЗ

Системний аналіз задає роль кожного елементу в комп'ютерній системі, взаємодія елементів один з одним.

Аналіз вимог відноситься до програмного елементу — програмного забезпечення. Уточнюються і деталізують його функції, характеристики і інтерфейс. Всі визначення документуються в *специфікації аналізу.* Тут же завершується рішення задачі планування проекту.

Проектування полягає в створенні представлень про: архітектуру ПЗ; модульну структуру ПЗ; алгоритмічну структуру ПЗ; структури даних; вхідний і вихідний інтерфейс (вхідних і вихідних форм даних). Початкові дані для проектування містяться в *специфікації аналізу,* тобто в ході проектування виконується трансляція вимог до ПЗ в безліч проектних уявлень. При вирішенні завдань проектування основна увага приділяється якості майбутнього програмного продукту.

Кодування полягає в перекладі результатів проектування в текст на мові програмування.

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

Супровід — це внесення змін до експлуатованого ПЗ. Цілі змін: виправлення помилок; адаптація до змін зовнішньою для ПЗ середовища; удосконалення ПЗ по вимогах замовника. Супровід ПЗ полягає в повторному застосуванні кожного з

попередніх кроків (етапів) життєвого циклу до існуючої програми але не в розробці нової програми.

Достоїнства класичного життєвого циклу: дає план і часовий графік по всіх етапах проекту, упорядковує хід конструювання. *Недоліки класичного життєвого циклу: 1.* реальні проекти часто вимагають відхилення від стандартної послідовності кроків. 2. Цикл заснований на точному формулюванні початкових вимог до ПЗ (реально на початку проекту вимоги замовника визначені лише частково). 3. Результати проекту доступні замовникові тільки в кінці роботи.

6.2 Макетування

Основна мета макетування — зняти невизначеності у вимогах замовника. *Макетування (прототипування)* — це процес створення моделі необхідного програмного продукту. *Модель може приймати одну з трьох форм:1.* Паперовий макет або макет на основі ПК (зображає або малює людино-машинний діалог). 2. Працюючий макет (виконує деяку частину необхідних функцій). 3. Існуюча програма (характеристики якої потім повинні бути покращенні). Макетування Грунтується на багатократному повторенні ітерацій, в яких беруть участь замовник і розробник (рис. 6.2).

Рисунок 6.2 - Макетування

Послідовність дій при макетуванні представлена на рис. 6.3. Макетування починається із збору і уточнення вимог до створюваного ПЗ Розробник і замовник зустрічаються і визначають всі цілі ПЗ, встановлюють, які вимоги відомі, а які належить до визначити. Потім виконується швидке проектування. У нім увага

зосереджується на тих характеристиках ПЗ, які повинні бути видимі користувачеві. Швидке проектування приводить до побудови макету. Макет оцінюється замовником і використовується для уточнення вимог до ПЗ.

Рисунок 6.3 - Послідовність дій при макетуванні

Ітерації повторюються до тих пір, поки макет не виявить всі вимоги замовника. *Гідність макетування:* забезпечує визначення повних вимог до ПЗ. *Недоліки макетування: 1.* Замовник може прийняти макет за продукт. 2. Розробник може прийняти макет за продукт. *Пояснимо суть недоліків.* Коли замовник бачить працюючу версію ПЗ, він перестає усвідомлювати, що деталі макету скріпляють «жувальною гумкою і дротом»; він забуває, що в гонитві за працюючим варіантом залишені невирішеними питання якості і зручності супроводу ПЗ. Коли замовникові говорять, що продукт повинен бути перебудований, він починає обурюватися і вимагати, щоб макет «в три прийоми» був перетворений на робочий продукт. Дуже часто це негативно позначається на управлінні розробкою ПЗ. *З іншого боку,* для швидкого отримання працюючого макету розробник часто йде на певні компроміси.

6.3 Стратегії конструювання ПЗ

Існують 3 стратегії конструювання ПЗ:

^ *одноразовий прохід* (стратегія водопаду) — лінійна послідовність етапів конструювання;

інкрементна стратегія. На початку процесу визначаються всі призначені для користувача і системні вимоги, частина конструювання, що залишилася, виконується у вигляді послідовності версій. Перша версія реалізує частину запланованих можливостей, наступна версія реалізує додаткові можливості поки не буде отримана повна система;

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

6.4 Інкрементна модель

Інкрементна модель є класичним прикладом інкрементної стратегії конструювання (рис. 6.4). Вона об'єднує елементи послідовної моделі водопаду з ітераційною філософією макетування. Кожна лінійна послідовність тут виробляє інкремент, що поставляється, ПЗ.

Рисунок 6.4 - Інкрементна модель

6.5 Швидка розробка додатків

Модель швидкої розробки додатків (Rapid Application Development) — другий приклад застосування інкрементної стратегії конструювання (рис. 6.5). RAD-модель забезпечує екстремально короткий цикл розробки. RAD — високошвидкісна адаптація лінійної послідовної моделі, в якій швидка розробка досягається за рахунок використання компонентно-орієнтованого конструювання. Якщо вимоги повністю визначені, а проектна область обмежена, RAD- процес дозволяє групі створити повністю функціональну систему за дуже короткий час (60-90 днів). RAD-подход орієнтований на розробку інформаційних систем і виділяє наступні етапи:

^ **бізнес-моделювання.** Моделюється інформаційний потік між бизнес-функциями. Шукається відповідь на наступні питання: Яка інформація керує бизнес-процессом? Яка генерується інформація? Хто генерує її? Де інформація застосовується? Хто обробляє її?

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

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

^ генерація додатку. Передбачається використання методів, орієнтованих на мови програмування 4-го покоління. Замість створення ПЗ за допомогою мов програмування 3-го покоління, RAD-процесс працює з повторно використовуваними програмними компонентами або створює повторно використовувані компоненти. Для забезпечення конструювання використовуються утиліти автоматизації;

^ тестування і об'єднання. Оскільки застосовуються повторно використовувані компоненти, багато програмних елементів вже протестовано. Це зменшує час тестування (хоча все нові елементи повинні бути протестовані).

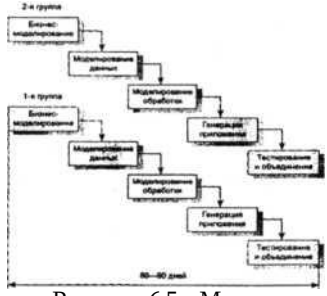

Рисунок 6.5 - Модель швидкої розробки додатків

Застосування RAD можливо у тому випадку, коли кожна головна функція може бути завершена за 3 місяці. Кожна головна функція адресується окремій групі розробників, а потім інтегрується в цілу систему. *Застосування R.4D має і свої недоліки, і обмеження: 1*. Для великих проектів в RAD потрібні істотні людські ресурси (необхідно створити достатню кількість груп). 2. RAD застосовна тільки для таких застосувань, які можуть декомпозироваться на окремі модулі і в яких продуктивність не є критичною величиною. 3. RAD не застосовна в умовах високих технічних рисок (тобто при використанні нової технології).

6.6 Спіральна модель

Спіральна модель — класичний приклад застосування еволюційної стратегії конструювання. Спіральна модель (автор Баррі Боєм, 1988) базується на кращих властивостях класичного життєвого циклу і макетування, до яких додається новий елемент, — аналіз риски, відсутній в цих парадигмах.

Рисунок 6.6 - Спіральна модель: / — початковий збір вимог і планування проекту; *2* — та ж робота, але на основі рекомендацій замовника; 3 — аналіз риски на основі початкових вимог; *4* — аналіз риски на основі реакції замовника; 5 — перехід до комплексної системи; 6 — початковий макет системи; 7 — наступний рівень макегу; *8* — сконструйована система; 9 — оцінювання замовником

Як показано на рис. 6.6, модель визначає чотири дії, що представляються чотирма квадрантами спіралі. 1. Планування — визначення цілей, варіантів і обмежень. 2. Аналіз риски — аналіз варіантів і розпізнавання/вибір риски.

3. Конструювання — розробка продукту наступного рівня. 4. Оцінювання — оцінка замовником поточних результатів конструювання. *Достоїнства спіральної моделі: 1.* Найреальніше (у вигляді еволюції) відображає розробку програмного забезпечення. 2. Дозволяє явно враховувати ризик на кожному

витку еволюції розробки. 3. Включає крок системного підходу в ітераційну структуру розробки. 4. Використовує моделювання для зменшення риски і вдосконалення програмного виробу. *Недоліки спіральної моделі: 1.* новизна (відсутня достатня статистика ефективності моделі). 2. Підвищені вимоги до замовника. 3. Труднощі контролю і управління часом розробки.

6.7 Компонентно-орієнтована модель

Компонентно-орієнтована модель є розвитком спіральної моделі і теж ґрунтується на еволюційній стратегії конструювання. У цій моделі конкретизується зміст квадранта конструювання — воно відображає той факт, що в сучасних умовах нова розробка повинна ґрунтуватися на повторному використанні існуючих програмних компонентів (рис. 6.7).

Рисунок 6.7 - Компонентно-орієнтована модель

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

Достоїнства компонептно-

орієнтованої моделі:

1. Зменшує на 30% час розробки програмного продукту. 2. Зменшує вартість програмної розробки до 70%. 3. Збільшує в півтора рази продуктивність розробки.

6.8 XP- процес

Екстремальне програмування (Extreme Programming, XP) - облегшений (рухливий) процес (або методологія), головний автор якого Кент Бек (1999). XP-групу утворюють до 10 співробітників, які розміщуються в одному приміщенні. Основна ідея XP - усунути високу вартість зміни, характерно для додатків з використанням об'єктів, патернів і реляційних баз даних. Базис XP утворюють перечисленні нижче методи:

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

2. Часта зміна версій - швидкий запуск в підприємство простої системи. Нові версії реалізуються в дуже короткому (двухтижневому) циклі.

3. Метафора - вся розробка проводиться на основі простої, загальнодоступної історії про те, як працює вся система.

4. Просте проектування - проектування виконується настільки просто, наскільки це можливо в даний момент.

5. Тестування - безперервне написання тестів для модулів, які повинні виконуватися бездоганно; замовники пишуть тести для демонстрації закінченості функцій.

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

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

8. Колективне володіння кодом - будь-який розробник може покращувати будь- який

код системи в будь-який час.

9. Безперервна інтеграція - система інтегрується і будується багато разів на день по мірі завершення кожного завдання.

10. 40-годинний тиждень -. як правило, працюють не більш, ніж 40 годин на тиждень.

11. Локальний замовник - у групі весь час повинен перебувати представник замовника, дійсно готовий відповідати на питання розробників.

12. Стандарти кодування - повинні витримуватися правила, що забезпечують однакове уявлення програмного коду у всіх частинах програмної системи.

6.9 Контрольні питання

1. Дайте визначення технології конструювання програмного забезпечення.
2. Які етапи класичного життєвого циклу ви знаєте?
3. Охарактеризуйте зміст етапів класичного життєвого циклу.
4. Поясните достоїнства і недоліки класичного життєвого циклу.
5. Які існують форми макетування?
6. Поясните достоїнства і недоліки інкрементної моделі.
7. Модель швидкої розробки додатків. Основні поняття та терміни. Поясните достоїнства і недоліки моделі швидкої розробки додатків.
8. Вкажіть схожість і відмінності спіральної моделі і класичного життєвого циклу.
9. Що таке спіральна модель? Що таке екстремальне програмування? Основні поняття та методи.

Тема 7. Аналіз, характеристика та структура програмного забезпечення

7.1 Вимоги до ПЗ (Software Requirements)

Вимоги - це властивості, якими повинне володіти ПЗ для адекватного визначення функцій, умов й обмежень виконання ПЗ, а також обсягів даних, технічного забезпечення й середовища функціонування. Вимоги відображають потреби людей (замовників, користувачів, розроблювачів), зацікавлених у створенні ПЗ. Замовник і розроблювач спільно проводять збір вимог, їхній аналіз, перегляд, визначення необхідних обмежень і документування. *Вимоги до продукту та до процесу* визначають умови функціонування та режими роботи ПЗ в операційному середовищі, обмеження на структуру й пам'ять комп'ютерів, на принципи взаємодії програм і комп'ютерів і т.п. *Функціональні вимоги* визначають призначення та функції системи, а не функціональні - умови виконання ПЗ й доступу до даних. *Системні вимоги описують* вимоги до програмної системи, що складаються із взаємопов'язаних програмних і апаратних підсистем і різних додатків.

Інженерія вимог до ПЗ - це дисципліна аналізу й документування вимог до ПЗ, що полягає в перетворенні запропонованих замовником вимог до системи в описі вимог до ПЗ і їх валідація. Вона базується на моделі процесу визначення вимог і діючих осіб, що забезпечують керування й формування вимог, а також на методах досягнення показників якості.

Якість і процес поліпшення вимог - це процес формулювання характеристик і атрибутів якості (надійність, реактивність та ін.), якими повинне володіти ПЗ, методи їхнього досягнення на етапах ЖЦ й оцінювання отриманих результатів.

Виявлення вимог - це процес виймання інформації з різних джерел (договорів, матеріалів аналітиків із декомпозиції завдань і функцій системи та ін.), проведення технічних заходів (співбесід, зборів та ін.) заради формування окремих вимог до продукту та до процесу розробки. Виконавець повинен узгодити вимоги із замовником.

Аналіз вимог - процес вивчення потреб і цілей користувачів, класифікація й перетворення їх до вимог до системи, апаратурі і ПЗ, установлення й дозвіл конфліктів

між вимогами, визначення пріоритетів, границь системи й принципів взаємодії із середовищем функціонування. *Функціональні вимоги* характеризують функції системи або її ПЗ, способи поведінки ПЗ в процесі виконання функцій і методи передачі та перетворення вхідних даних у результати. *Не функціональні вимоги* визначають умови й середовище виконання функцій (наприклад, захист і доступ до БД, таємність, взаємодія компонентів та ін.).

Специфікація вимог до ПЗ - процес формалізованого опису функціональних і не функціональних вимог, вимог до характеристики якості у відповідності зі стандартом якості ІБО/IEC9126-94, які будуть відпрацьовуватися на етапах ЖЦ ПЗ.

Валідація вимог - це перевірка викладених у специфікації виконавчих вимог для того, щоб шляхом відстеження джерел вимог переконатися, що вони визначають саме дану систему. Замовник і розроблювач ПЗ проводять експертизу сформованого варіанта вимог для того щоб розроблювач міг далі проводити проектування

Верифікація вимог - це процес перевірки правильності специфікацій вимог на їхню відповідність, несуперечність, повноту та здійсниміcть, а також на відповідність стандартам.

Керування вимогами - це керівництво процесами формування вимог на всіх етапах ЖЦ і включає керування змінами й атрибутами вимог, а також проведення моніторингу.

7.2 Проектування ПЗ (Software design)

Проектування ПЗ - це процес визначення архітектури, компонентів, інтерфейсів, інших характеристик системи й кінцевого складу програмного продукту.

Базова концепція проектування ПЗ - це методологія проектування архітектури за допомогою різних методів (об'єктного, компонентного й ін.), процеси ЖЦ (стандарт ISO/IEC12207) і техніки - декомпозиція, абстракція, інкапсуляція й ін.

До ключових питань проектування ПЗ відносяться: декомпозиція програм на функціональні компоненти для незалежного й паралельного їхнього виконання, принципи розподілу компонентів у середовищі виконання і їхньої н іаємодії між

собою, механізми забезпечення якості й живучості системи й ін.

При проектуванні архітектури ПЗ використається архітектурний стиль проектування, заснований на визначенні основних елементів структури - підсистем, компонентів і зв'язків між ними.

Архітектура проекту - багаторівневе подання структури системи й специфікація її компонентів. Архітектура визначає логіку окремих компонентів системи настільки детально, наскільки це необхідно задля написання коду, а іакож визначає зв'язки між компонентами.

Одним з найважливіших інструментів проектування архітектури є пшптерн - типовий конструктивний елемент ПЗ, що задає взаємодію об'єктів (компонентів) проектованої системи, а також ролі й відповідальності виконавців. Основною мовою опису цього елемента є UML.

Аналіз й оцінка якості проектування ПЗ включає заходи щодо аналізу сформульованих у вимогах атрибутів якості, оцінки різних аспектів ПЗ - кількості функцій, структура ПЗ, якості проектування за допомогою формальних метрик (функціонально-орієнтованому, структурних й об'єктно-

орієнтованих), а також проведення якісного аналізу результатів проектування шляхом статичного аналізу, моделювання й прототипирування.

Нотації проектування дозволяють представити опис об'єкта (елемента) ПЗ і його структуру, а також поведінку системи. Існує два типи нотацій: структурна, поведінкові й безліч різних їхніх подань.

Структурні нотації - це структурне, блок-схемне або текстове подання аспектів проектування структури ПЗ з об'єктів, компонентів, їхніх інтерфейсів і взаємозв'язків. До нотацій відносяться формальні мови специфікацій і проектування: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity-Relation Diagrams), IDL (Interface Description Language), Use Case Driven й ін.

Стратегія й методи проектування ПЗ. До стратегій відносяться: проектування знизу-вверх, зверху-вниз, абстрагування, використання патернів.

7.3 Конструювання ПЗ (Software Construction)

Конструювання ПЗ - створення працюючого ПЗ із залученням методів верифікації, кодування й тестування компонентів.

Зниження складності - це мінімізація, зменшення й локалізація складності конструювання. *Мінімізація складності* визначається обмеженими можливостями виконавців обробляти складні структури та великі обсяги інформації протягом тривалого періоду часу. *Зменшення складності* в конструюванні ПЗ досягається при створенні простого коду, що легко читати, задля додання більшої значимості цього коду, простоті тестування, продуктивності й задоволенню заданих критеріїв. *Локалізація складності* - це спосіб конструювання із застосуванням об’єктно-орієнтованого підходу, що лімітує інтерфейс об'єктів, спрощує їхню взаємодію, також перевірку правильності самих об'єктів і зв'язків між ними. Локалізація спрощує внесення змін, пов'язаних з виявленими помилками в коді, або джерелом помилок є середовище, у якій виконується код.

Попередження відхилень від стилю. Для вирішення різних завдань конструювання застосовуються різні стилі конструювання (лінгвістичний, формальний, візуальний). *Лінгвістичний стиль* заснований на використанні

словесних інструкцій і висловів для подань окремих елементів (конструкцій) програм. *Формальний стиль* використається для точного, однозначного й формального визначення компонентів системи. *Візуальний стиль* є найбільш універсальним стилем конструювання ПЗ, він дозволяє розроблювачам проекту уявляти конструйовуваний елемент у наочному виді.

Структуризація перевірок припускає, що побудова ПС повинне проводитися таким чином, щоб сама система допомагала вести пошук помилок, дефектів і причин збоїв при застосуванні різних методів перевірки, як на стадії незалежного тестування (наприклад, інженерами-тестувальниками), так і в процесі експлуатації, коли особливо важливо швидке виявлення й виправлення виникаючих помилок.

Використання зовнішніх стандартів. Конструювання ПЗ залежить від застосування зовнішніх стандартів, пов'язаних з мовами програмування, інструментальними засобами й інтерфейсами. *Управління конструюванням* - це керування процесом конструювання ПЗ, що базується на моделях конструювання, планування й внесення змін. Моделі конструювання включають набір операцій, послідовність дій і результатів.

7.4 Тестування ПЗ

Тестування ПЗ - це процес перевірки готової програми в статистиці (перегляди, інспекції, налагодження вихідного коду) і в динаміці шляхом прогону кінцевого набору тестових даних, що перевіряють різні шляхи

виконання програми й порівнянні отриманих результатів із заздалегідь запланованими. Існує дві форми перевірки коду - модульне й інтеграційне. При цьому використаються стандарти (IEEE 829-1996 й IEEE 1008-1987) перевірки й тестування модулів ПЗ. *Основна концепція тестування* описує базові терміни, ключові проблеми і їхній зв'язок з іншими областями знань. *Тестування* - це процес перевірки правильності програми в динаміці її виконання на тестових даних. При тестуванні виявляються недоліки: відмови (faults) і дефекти (defects) як причини порушення роботи програми, збої (failures) як небажані ситуації, помилки (errors), як наслідок збоїв й ін.

7.5 Супровід ПЗ (Software maintenance)

Супровід ПЗ - сукупність дій по забезпеченню роботи ПЗ, а також по внесенню змін у випадку виявлення помилок у процесі експлуатації, по адаптації ПЗ до нового середовища функціонування, а також по підвищенню продуктивності або поліпшенню інших характеристик ПЗ.

Ключові питання супроводу ПЗ. Основними із цих питань є управлінські, вимірювальні й вартісні. Сутність управлінських питань складається в контролі ПЗ в процесі модифікації, удосконалюванні функцій і недопущенні зниження продуктивності системи.

Еволюція ПЗ. Відомий фахівець в області ПЗ Дж. Леман (1970р.) запропонував розглядати супровід як еволюційну розробку програмних систем, оскільки здана в експлуатацію система не завжди є повністю завершеною, її треба змінювати протягом строку експлуатації.

Реінженерія - це вдосконалення застарілого ПЗ шляхом його реорганізації або реструктуризації, а також перепрограмуванням окремих елементів або налаштування параметрів на іншу платформу або середовище виконання зі збереженням зручності його супроводу.

Реверсна інженерія полягає у відновленні специфікації (графів викликів, потоків даних й ін.) по отриманому коду системи для спостереження за нею на більш високому рівні.

Рефакторинг - це реорганізація коду для поліпшення характеристик і

показників якості об'єктно-орієнтованих і компонентних програм без зміни їхньої поведінки.

7.6 Управління конфігурацією ПЗ

Управління конфігурацією (Software Configuration Management - SCM) складається з ідентифікації компонентів системи, визначенні функціональних і фізичних характеристик апаратного й програмного забезпечення для контролю ні внесенням змін і трасуванням конфігурації протягом ЖЦ.

Конфігурація системи - склад функцій, програмного й технічного забезпечення системи, можливі їхні комбінації залежно від наявності обладнання, загальносистемних засобів, позначених у технічній документації системи, і вимоги до продукту.

Управління процесом конфігурації. Це діяльність з контролю еволюції й ііінісності продукту при ідентифікації, контролю змін і забезпеченні звітною інформацією, що стосується конфігурації.

Ідентифікація конфігурації ПЗ полягає в документуванні функціональних і фізичних характеристик елементів конфігурації ПЗ, а також в оформленні ісхнічної документація на елементи конфігурації ПЗ.

Контроль конфігурації ПЗ складається в проведенні робіт з координації, і іисрдженні або відкиданні реалізованих змін в елементах конфігурації після формальної її ідентифікації, а також в аналізі вхідних компонентів у і опфігурацію та відповідності їхньої ідентифікації.

Облік статусу або стану конфігурації ПЗ проводиться за допомогою комплексу заходів, що дозволяють визначити ступінь зміни конфігурації, отриманої від розроблювана, а також правильність внесених змін у конфігурацію ПЗ при її супроводі.

Аудит конфігурації - це виконавча діяльність задля оцінки продукту й процесів на відповідність стандартам, інструкціям, планам і процедурам.

Управління версіями ПЗ - це відстеження наявної версії елемента конфігурації; зборка компонентів; створення нових версій системи на основі існуючих шляхом внесення змін у конфігурацію; узгодження версії продукту з вимогами та проведеними змінами на етапах ЖЦ; забезпечення оперативного доступу до інформації про елементи конфігурації й системи.

Базис (baseline) - формально позначений набір елементів ПЗ, зафіксований на етапах ЖЦ ПЗ. *Бібліотека ПЗ* - контрольована колекція об'єктів ПЗ й документації, призначена для полегшення процесу розробки, використання й супроводу ПЗ. *Зборка ПЗ* - об'єднання коректних елементів ПЗ й конфігураційних даних у єдину виконавчу програму.

7.7 Контрольні питання

1. Що таке вимоги до ПЗ?
2. З яких розділів складається область знань "Проектування ПЗ (Software Design)"?
3. Що таке конструювання ПЗ?
4. З яких розділів складається область знань " Тестування ПЗ (Software Testing)"?
5. З яких розділів складається область знань " Супровід ПЗ(Software maintenance)"?
6. З яких розділів складається область знань " Керування конфігурацією ПЗ"?

Тема 8. Архітектура програмного забезпечення 8.1

Введення в архітектуру програм

Протягом десятиліть розробники програмного забезпечення створювали свої проекти або з нуля, або використовуючи вже накопичений досвід, якщо їдкий якось вдавалося придбати. В даний час інтенсивно розвивається дисципліна програмних

архітектур та проектування.

Рисунок 8.1 - Схема процесів розробки програм: теми глави

Системна розробка - це процес аналізу і проектування, який розділяє додаток на апаратні і програмні компоненти. Деякі аспекти цієї декомпозиції іиктуються вимогами замовника, інші визначаються розробниками.

Створення архітектури - це проектування на найвищому рівні. Частину процесу проектування ми будемо називати детальним проектуванням.

8.2 Цілі вибору архітектури

Для конкретного проекту розробки програмного забезпечення може бути декілька відповідних архітектур, з яких необхідно вибрати кращу. Зазвичай буває складно задовольнити всі вимоги, оскільки архітектура може виконувати одну з вимог і не виконувати іншого. З цієї причини всім вимогам необхідно присвоїти пріоритети. Наведемо приклад списку основних цілей розробки.

^ Розширення - Полегшення додавання нових властивостей. Розширення визначає ступінь, в якій архітектура повинна підтримувати додавання нових можливостей в додаток. Найчастіше чим краще архітектура пристосована до розширення, тим більше складну структуру вона має і більше часу потрібно на розробку. Розширюваність зазвичай вимагає введення більш високих абстракцій в процес.

^ Зміни - Полегшення зміни вимог. Розробка з розрахунком на зміни переслідує інші цілі, хоча увазі застосування тих же прийомів проектування, що і для

забезпечення розширюваності.

^ Простота: простота розуміння та простота реалізації. ***Простота*** є метою проектування при будь-яких обставин. Проста архітектура, яка допускає розширення і зміни, є рідкістю, і її створення вимагає великих зусиль.

Ефективність: досягнення високої швидкості: виконання і (або) компіляції та досягнення малого розміру : об'єктного коду та (або) вихідного коду. До інших критерій, що використовуються при виборі архітектури, відносяться економія машинного часу і економія пам'яті.

8.3 Декомпозиція

Після невеликої практики досить просто створювати маленькі програми. Великі програми, однак, ставлять перед розробниками дуже важкі завдання, вирішення яких досить важко досягається на практиці. Принципова проблема систем програмного забезпечення - це їх складність. Складність не в сенсі кількості рядків коду, а в сенсі їх взаємозв'язку. Дуже хороший спосіб боротьби # зі складністю - розбиття задачі на підзадачі, що мають характерні властивості невеликих програм. З цієї причини декомпозиція (або модулярізація) є проблемою критичної важливості і одним з найцікавіших етапів розробки. Розробник першою справою повинен представити, як додаток буде працювати на вищому рівні, а потім розробити декомпозицію, відповідну ментальної моделі.

8.4 Модель та зразки проектування

Декомпозиція всього проекту на компоненти є суттєвим кроком, але нам ще належить набагато велика робота по створенню архітектури. Для початку нам необхідно узгодити варіанти використання, класи, переходи станів і декомпозицію (моделями). При створенні моделі класів доцільно розробляти і використовувати вже існуюче програмне забезпечення, яке утворює базис для сімейства східних додатків. Таке сімейство, зване каркасом. *Детальне проектування* - це повний обсяг робіт з проектування, виключаючи архітектуру і реалізацію. Воно містить визначення класів,

що пов'язують класи предметної області і класи архітектури. Замість того щоб «винаходити велосипед», ми намагаємося використовувати розробки, що вже довели свою ефективність у попередніх додатках. *Зразки проектування* - це шаблони взаємодіючих класів і методів, які вже показали своє значення для багатьох додатків.

8.5 Використання моделей

Зазвичай нам необхідно описати додаток з декількох точок зору. Це можна порівняти з архітектурою будинку, яка вимагає декількох проекцій, таких як план розташування ділянок землі, вертикальний і фронтальний види, план водопроводу і т. д. У світі програмування проекції називаються ***моделями.*** За останні роки в цій області з'явилося безліч прекрасних розробок. Різні моделі проектованого додатки показані на рис. 8.2 і 8.3. Багато позначення взяті з USDP.

Рисунок 8.2 - Моделі, що розглядають додатки з різних точок зору

Рисунок 8.3 - Моделі та їх частини

Модель варіантів використання являє собою колекцію варіантів використання. Вони пояснюють, що саме додаток повинен робити. Початкова версія варіантів використання підходить для використання в якості С вимог (іноді їх називають «ділові варіанти використання» або «варіанти використання предметної області»). В процесі реалізації проекту вони набувають властивості діаграм послідовності спеціального виду. їх реалізують в вигляді конкретних сценаріїв, які потім використовуються при тестуванні. *Модель класів*. Ми вже розглянули досить багато моделей класів (діаграм класів). Модель класів пояснює побудова блоків, з яких буде сформовано додаток. Моделі класів часто називають об'єктними моделями. Всередині моделі класів ми можемо показати методи і атрибути. *Модель* ***компонентів*** є колекцією діаграм потоків даних. Нони пояснюють, яким чином додаток буде працювати в термінах переміщення даних. *Модель* ***переходів*** станів являє собою колекцію діаграм переходів станів. Модель переходів станів визначає момент часу, в який додаток здійснює свою роботу. Усередині кожної моделі ми опрацьовуємо рівні деталізації, кількість яких зростає. Залежно від обсягів роботи ми можемо ітеративно застосовувати розподіл на рівні деталізації всередині кожної з моделей. Архітектура програми найчастіше виражається в термінах однієї з моделей і підтримується іншими моделями.

8.6 Класифікація архітектури

Шоу та Гарлан класифікували архітектури програмного забезпечення з точки юру практики. Іншими словами, вони зібрали разом зразки програмного забезпечення для різних архітектур. їх класифікація, трохи адаптована, показана нижче: 1. Архітектури потоків даних: Послідовні пакети. Канали і фільтри. 2. Незалежні компоненти: Паралельні взаємодіючі процеси. Клієнт- серверні системи. Системи, керовані подіями. 3. Віртуальні машини: Інтерпретатори. Системи, засновані на правилах. 4. Репозиторні архітектури. Бази даних. Гіпертекстові системи. Дошки оголошень. 5. Рівневі архітектури.

8.7 Зразки проектування

Зразок проектування - це знайдена досвідченим шляхом комбінація компонентів, зазвичай класів чи об'єктів, яка вирішує певні загальні проектувальні завдання. Скористаємося аналогією з архітектурою будинку і розглянемо задачу проектування усамітненого будівлі на великій території. Архітектура Ранчо (одноповерховий будинок) повністю задовольняє цим вимогам. Зауважте, що Ранчо вказує на загальну ідею проектування, що передбачає безліч реалізацій, і зовсім не є незмінним безліччю планів будинку. Гамма представив увазі спільноті розробників зразки проектування в тепер уже класичній книзі. Гамма розглядає двадцять три зразка проектування, розділяючи їх на структурну, креативну і поведінкову категорії. Структурні зразки проектування мають справу із способами представлення об'єктів (такими, як дерева або зв'язні списки). Вони зручні у багатьох випадках, оскільки дозволяють користуватися безліччю об'єктів як єдиним цілим. Креативний зразок проектування пов'язаний зі способами створення складних об'єктів, таких як лабіринти і дерева. Поведінковий зразок проектування дозволяє нам стежити за поведінкою об'єктів, наприклад, видаючи звіт про колекцію об'єктів у певному порядку. Зразки проектування можуть бути застосовані на рівні архітектури та (або) на рівні детального проектування.

8.8 Компоненти

У другій половині 90-х років сильно зріс інтерес до поняття «компонент». Компонентами є повторно використовувані об'єкти, які не вимагають знання програмного забезпечення, що використовує їх. Наочним прикладом технології компонентів можуть служити об'єкти COM і Java Beans. Компоненти можуть бути об'єктами в звичайному розумінні об'єктно-орієнтованого програмування, але з невеликими поправками на забезпечення їх автономності. Одні компоненти використовуються іншими за допомогою агрегування і взаємодіють в основному за

допомогою подій. Узагальнення зв'язків між каркасом, архітектурою, детальним проектуванням, моделями та зразками

проектування показано на рис. 8.4. Зразки проектування можуть бути використані і на рівні каркаса, і всередині проектних класів на рівні архітектури, і на рівні детального проектування.

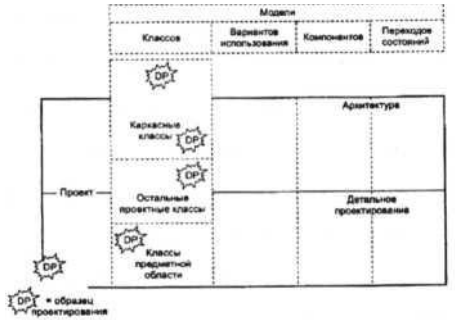

Рисунок 8.4 - Зв'язки між каркасами, архітектурою, проектуванням і моделями

8.9 Типи архітектури і їх моделі

Розробник програмного забезпечення створює ментальну модель, що описує роботу програми, обмежуючись при цьому п'ятьма-сімома компонентами (вельми приблизна оцінка). Результат проектування, звичайно, визначається в основному додатком, проте багато що може бути запозичене з раніше розроблених архітектур, так само як в проект підвісного моста увійде безліч переваг раніше розроблених підвісних мостів. У цьому розділі ми детально розглянемо архітектури і вкажемо зразки проектування, які можуть допомогти в реалізації цих архітектур.

8.10 Архітектури, засновані на потоках даних

Для представлення деяких додатків найкращим чином підходять потоки даних між процесами обробки даних. Таке уявлення ілюструють діаграми потоків

даних (DFD - Data Flow Diagram). Кожен процес обробки даних на діаграмі потоків даних проектується незалежно від інших. Дані приходять з різних джерел, наприклад від користувача, і, в кінцевому підсумку, повертаються до користувача або в приймачі даних, такі як база даних рахунків. Протягом десятиліть потоки даних були найбільш загальним способом відображення архітектур, і можна з упевненістю сказати, що вони не втратять актуальності і в майбутньому. Для розробників цілком природно уявляти собі, що дані подорожують від одного вузла до іншого і обробляються в кожному вузлі. Недоліком діаграм потоків даних є те, що їх відображення на програмний код не цілком ясно, причому незалежно від того, чи йде мова про об'єктно-орієнтованому коді, чи ні.

8.11 Рівневі архітектури

Рівень архітектури - це логічно пов'язана колекція артефактів програмного забезпечення, зазвичай - пакети класів. У загальному вигляді рівень використовує не більше одного рівня і разом з тим використовується не більше ніж одним рівнем. Побудова додатків послідовно рівень за рівнем сильно спрощує процес. Деякі рівні, наприклад каркаси, можуть використовуватися в декількох додатках.

8.12 Контрольні питання

1. Яка схема процесів розробки програм?
2. Що таке декомпозиція? Основні цілі.
3. Поняття детальне проектування?
4. Що таке моделі? їх різновид.
5. Що таке зразок проектування?
6. Процедура вибору архітектури?

Тема 9. Проектування архітектури програмного забезпечення

9.1 Аналіз області рішень

Після того, як розібралися в предметній області, зрозуміли, що потрібно під майбутньої програмної системи, далі досліджують можливі способи вирішення тих завдань, які поставлені у вимогах. Таким чином, явно чи неявно, проводиться *аналіз області рішень***.** Метою цієї діяльності є розуміння, чи можна взагалі вирішити поставлені перед розроблюваної системою завдання, за яких умов і обмеженнях це можна зробити, як вони вирішуються, якщо рішення є, а якщо ні - чи не можна придумати спосіб його знайти або отримати хоча б приблизне рішення. Коли визначено принципові способи вирішення всіх поставлених завдань основною проблемою стає спосіб організації програмної системи, який дозволив би реалізувати всі ці рішення і при цьому задовольнити вимогам програми. Шуканий спосіб організації ПЗ у вигляді системи взаємодіючих *компонентів* називають *архітектурою,а* процес її створення - *проектуванням архітектури ПЗ.*

9.2 Архітектура ПЗ

Під *архітектурою ПЗ* розуміють набір внутрішніх структур ПЗ, які видно з різних точок зору і складаються з *компонентів,* їх зв'язків і можливих взаємодій між *компонентами,* а також доступних ззовні властивостей цих *компонентів.* Під *компонентом* в цьому визначенні мається на увазі досить довільний структурний елемент ПЗ, який можна виділити, визначивши інтерфейс взаємодії між цим *компонентом* і всім, що його оточує. Зазвичай при розробці ПЗ термін "компонент" - це одиниця розгортання, найменша частина системи, яку можна включити або не включити до її складу. Такий компонент
також має певний інтерфейс і задовольняє деякому набору правил, званому компонентної моделлю. Там, де можливі непорозуміння, буде вказано, в якому сенсі вживається цей термін. У визначенні *архітектури* згадується набір структур, а не одна структура. Це означає, що в якості різних аспектів архітектури,

різних поглядів на неї виділяються різні структури, що відповідають різним аспектам взаємодії компонентів.

Архітектура ПЗ являє собою набір структур або *уявлень,* що мають різні рівні абстракції і що показують різні аспекти (структуру класів ПЗ, структуру розгортання, тобто прив'язки компонентів ПЗ до фізичних машинам, можливі сценарії взаємодій компонентів), що об'єднуються зіставленням всіх представлених даних із структурними елементами ПЗ. Архітектура визначає більшість характеристик якості ПЗ в цілому. Архітектура служить також основним засобом спілкування між розробниками, а також між розробниками і всіма іншими особами, зацікавленими в даному ПЗ. Вибір архітектури задає спосіб реалізації вимог на високому рівні абстракції. Саме архітектура майже повністю визначає такі характеристики ПЗ як надійність, переносимість і зручність супроводу. Вона також значно впливає на зручність використання і ефективність ПЗ, які, однак, сильно залежать і від реалізації окремих компонентів. Значно менше вплив архітектури на функціональність - зазвичай задану функціональність можна реалізувати, використавши абсолютно різні архітектури. Тому вибір між тією або іншою архітектурою визначається більшою мірою саме нефункціональними вимогами і необхідними властивостями ПЗ з точки зору зручності супроводу та переносимості. При цьому для побудови гарної архітектури треба враховувати можливі протиріччя між вимогами до різних характеристиках і вміти вибирати компромісні рішення, що дають прийнятні значення за всіма показниками. Так, для ^ підвищення ефективності в загальному випадку вигідніше використовувати монолітні архітектури, в яких виділено невелике число компонентів (в межі - єдиний компонент). Цим забезпечується економія як пам'яті, оскільки кожен компонент зазвичай має свої дані, а тут число компонентів мінімально, так і часу роботи, оскільки можливість оптимізувати роботу алгоритмів обробки даних є також тільки в рамках одного компонента.

З іншого боку, для підвищення зручності супроводу, навпаки, краще розбити систему на велике число окремих маленьких компонентів, з тим щоб кожен з них вирішував свою невелику, але чітко певну частину загальної задачі. І Іри цьому, якщо виникають зміни у вимогах або проекті, їх зазвичай можна звести до зміни в постановці однієї, рідше двох або трьох таких підзадач і змінювати тільки

відповідають за вирішення цих підзадач компоненти.

З третього боку, для підвищення надійності краще використовувати або невеликий набір простих компонентів, або дублювання функцій, тобто зробити кілька компонентів відповідальними за вирішення однієї підзадачі. Зауважимо, однак, що помилки в ПЗ найчастіше носять невипадковий характер. Вони повторювані, на відміну від апаратного забезпечення, де помилки пов'язані часто з випадковими змінами характеристик середовища і можуть бути подолані простим дублюванням компонентів без зміни їх внутрішньої реалізації. Тому при такому забезпеченні надійності треба використовувати досить сильно відрізняються способи вирішення однієї і тієї ж задачі в різних компонентах.

Іншим прикладом суперечливих вимог служать характеристики зручності використання і захищеності. Чим сильніше захищена система, тим більше перевірок, процедур ідентифікації та пр. потрібно проходити користувачам. Відповідно, тим менш зручна для них робота з такою системою. При розробці реальних систем доводиться шукати деякий розумний компроміс, щоб зробити систему досить захищеною і здатною поставити відчутну перепону для несанкціонованого доступу до її даних і, в той же час, не відлякати користувачів складністю роботи з нею.

Список стандартів, що регламентують опис архітектури, яке є основною складовою проектної документації на ПЗ, виглядає так:

^ IEEE 1016-1998 Recommended Practice for Software Design Descriptions (Рекомендовані методи описів проектних рішень для ПЗ).

^ IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems (Рекомендовані методи опису архітектури програмних систем). Основний зміст цього стандарту зводиться до визначення набору понять, пов'язаних з архітектурою програмної системи. Це, перш за все, саме поняття архітектури як набору основоположних принципів організації системи, втілених у наборі її компонентів, зв'язки їх один з одним і між ними і оточенням системи, а також принципів проектування та розвитку системи. Це визначення, на відміну від

даного на початку цього лекції, робить акцент не на наборі структур в основі архітектури, а на принципах її побудови. Стандарт IEEE 1471 визначає також *уявлення архітектури (architectural description)* як узгоджений набір документів, що описує архітектуру з точки зору певної групи зацікавлених осіб за допомогою набору моделей. Архітектура може мати кілька уявлень, що відбивають інтереси різних груп зацікавлених осіб. Стандарт рекомендує для кожного подання фіксувати відображені в ньому погляди та інтереси, ролі осіб, які зацікавлені в такому погляді на систему, причини, що обумовлюють необхідність такого розгляду системи, невідповідності між елементами одного подання або між різними уявленнями, а також різну службову інформацію про джерела інформації, датах створення документів. *Стандарт IEEE 1471 відзначає необхідність використання архітектури системи для вирішення таких завдань, як наступні:*

> Аналіз альтернативних проектів системи.

> Планування перепроектування системи, внесення змін до її **&** організацію.

> Спілкування з приводу системи між різними організаціями, залученими в її розробку, експлуатацію, супровід, що здобувають систему або продають її.

> Вироблення критеріїв приймання системи при її здачі в експлуатацію.

> Розробка документації щодо її використання і супроводу, включаючи навчальні та маркетингові матеріали.

> Проектування і розробка окремих елементів системи.

> Супровід, експлуатація, управління конфігураціями та внесення змін і поправок.

> Планування бюджету та використання інших ресурсів в проектах, пов'язаних з розробкою, супроводом або експлуатацією системи.

> Проведення оглядів, аналіз і оцінка якості системи.

9.3 Розробка і оцінка архітектури на основі сценаріїв

При проектуванні архітектури системи на основі вимог, зафіксованих у вигляді варіантів використання, перші можливі кроки полягають у

наступному.

Наділення компонентів

^ Вибирається набір "основних" *сценаріїв використання* - Найбільш істотних і виконуваних частіше за інших.

^ Виходячи з досвіду проектувальників, обраного архітектурного стилю (див. наступну лекцію) і вимог до переносимості та зручності супроводу системи визначаються компоненти, що відповідають за певні дії в рамках цих *сценаріїв,* тобто за вирішення певних підзадач.

^Кожен *сценарій використання* системи представляється у вигляді послідовності обміну повідомленнями між отриманими компонентами.

^При виникненні додаткових добре виділених підзадач додаються нові компоненти, і сценарії уточнюються.

Визначення інтерфейсів компонентів

^ Для кожного компонента в результаті виділяється його інтерфейс - набір повідомлень, які він приймає від інших компонентів і посилає їм.

^ Розглядаються "неосновні" сценарії, які так само розбиваються на послідовності обміну повідомленнями з використанням, по можливості, вже визначених інтерфейсів.

^ Якщо інтерфейси недостатні, вони розширюються.

^ Якщо інтерфейс компонента занадто великий, або компонент відповідає за дуже багато, він розбивається на більш дрібні.

Уточнення набору компонентів

^ Там, де це необхідно в силу вимог ефективності чи зручності супроводу, декілька компонентів можуть бути об'єднані в один.

^ Там, де це необхідно для зручності супроводу або надійності, один компонент може бути розділений на декілька.

^ Досягнення потрібних властивостей. Все це робиться до тих пір, поки не виконаються наступні умови:

^ Всі ***сценарії використання*** реалізуються у вигляді послідовностей

обміну повідомленнями між компонентами в рамках їх інтерфейсів.

^ Набір компонентів достатній для забезпечення всієї потрібної функціональності, зручний для супроводу або портирования на інші платформи і не викликає помітних проблем продуктивності.

^ Кожен компонент має невеликий і чітко окреслене коло вирішуваних завдань і строго певний, збалансований за розміром інтерфейс.

На основі можливих *сценаріїв використання* або модифікації системи можливий також аналіз характеристик архітектури та оцінка її придатності для поставлених завдань або порівняльний аналіз декількох архітектур. Це так званий метод аналізу *архітектури ПЗ* (Software Architecture Analysis Method, SAAM). *Основні його кроки наступні.*

^ *Визначити набір сценаріїв дій користувачів або зовнішніх систем*, що використовують деякі можливості, які можуть вже плануватися для реалізації в системі або бути новими. Сценарії мають бути значущими для конкретних зацікавлених осіб, будь то користувач, розробник, відповідальний за супроводження, представник контролюючої організації . Чим повніше набір сценаріїв, тим вище буде якість аналізу. Можна також оцінити частоту появи і важливість сценаріїв, можливий збиток від неможливості їх виконати.

^ *Визначити архітектуру (або декілька порівнюваних архітектур).* Це має бути зроблено у формі, зрозумілою всім учасникам оцінки.

^ *Класифікувати сценарії.* Для кожного сценарію з набору має бути визначено, чи підтримується він вже даної архітектурою чи для його підтримки потрібно вносити в неї зміни. Сценарій може підтримуватися, тобто його виконання не зажадає внесення змін ні в один з компонентів, або ж не підтримуватися, якщо його виконання вимагає змін в описі поведінки одного або декількох компонентів або змін в їх інтерфейсах. Підтримка сценарію означає, що особа, зацікавлена в його виконанні, оцінює ступінь підтримки як достатню, а необхідні при цьому дії - як досить зручні.

^ *Оцінити сценарії.* Визначити, які з сценаріїв повністю підтримуються розглянутими архітектурами. Для кожного непідтримуваного сценарію треба

визначити необхідні зміни в архітектурі - внесення нових компонентів, зміни в існуючих, зміни зв'язків і способів взаємодії. Якщо є можливість, варто оцінити трудомісткість внесення таких змін.

^ *Виявити взаємодію сценаріїв.* Визначити які компоненти потрібно змінювати для непідтримуваних сценаріїв; якщо потрібно змінювати один компонент для підтримки декількох сценаріїв - такі сценарії називають взаємодіючими. Потрібно оцінити смислові зв'язки між взаємодіючими сценаріями. Мала зв'язаність за змістом між взаємодіючими сценаріями означає, що компоненти, в яких вони взаємодіють, виконують слабо пов'язані між собою завдання і їх варто декомпозировать. Компоненти, в яких взаємодіють багато (більше двох) сценаріїв, також є можливими проблемними місцями.

^*Оцінити архітектуру в цілому* (або порівняти кілька заданих архітектур). Для цього треба використовувати оцінки важливості сценаріїв і ступінь їх підтримки архітектурою.

9.4 Діаграмі при проектуванні архітектури ПЗ

Діаграми UML діляться на дві групи - *статичні і динамічні діаграми.*

Статичні діаграми представляють або постійно присутні в системі сутності і зв'язки між ними, або сумарну інформацію про сутності і зв'язках, або сутності та зв'язки, що існують в якийсь певний момент часу. Вони не показують способів поведінки цих сутностей. До цього типу належать ***діаграми класів***, об'єктів, компонентів і ***діаграми розгортання.***

Динамічні діаграми описують відбуваються в системі процеси. До них відносяться діаграми діяльності, сценаріїв, ***діаграми взаємодії*** і діаграми станів.

9.5 Контрольні питання

1. Що розуміють під архітектурою ПЗ?
2. Список стандартів, що регламентують опис архітектури?

3. Які групи діаграмі використовуються при проектуванні архітектури ПЗ?
4. Які діаграми належать до статичних діаграм?
5. Які діаграми належать до динамічних діаграм?

Тема 10. Стратегії и методи проектування програмного забезпечення

10.1 Стратегії та методи проектування програмного забезпечення (Software Design Startegies and Methods)

Існують різні загальні стратегії, що допомагають у проведенні робіт з проектування. На відміну від загальних стратегій, методи проектування більш · специфічні і, в основному, пропонують і надають нотації (або набори нотацій) для використання спільно з цими методами, а також процеси, яких необхідно дотримуватися в рамках використовуваного методу. Методи тут є більш загальними поняттями, це - ***методології,*** сконцентровані на процесі (зокрема, проектування) і передбачають дотримання певних правил і угодам, в тому числі по використовуваних виразних засобів. Такі методи корисні як інструмент систематизації і передачі знань у вигляді загального фреймворка (тобто комплексного набору понять, підходів, технік і інструментів) не тільки для окремих фахівців, але для команд і проектних груп програмних проектів.

10.2 Загальні стратегії (General Strategies)

Це звичайно часто згадувані і загальноприйняті стратегії: "Розділяй-і-володарюй"; проектування "зверху-вниз" і "знизу-вверх"; абстракція даних і приховування інформації; ітеративний і інкрементальний підхід та інші.

10.3 Функціонально-орієнтоване або структурне проектування Function-Oriented - Structured Design)

Це один з класичних методів проектування, в якому декомпозиція

сфокусована на ідентифікації основних програмних функцій і, потім, детальної розробки та уточнення цих функцій "зверху-вниз". Структурний проектування, зазвичай, використовується після проведення структурного аналізу з застосуванням діаграм потоків даних і пов'язаним описом процесів. Дослідники пропонують різні стратегії і метафори чи підходи для трансформації DFD в програмну архітектуру, подану у формі структурних схем. Наприклад, порівнюючи управління і поведінку з одержуваним ефектом.

10.4 Об'єктно-орієнтоване проектування (Object-Oriented Design)

Являє собою безліч методів проектування, що базуються на концепції об'єктів. Дана область активно еволюціонує з середини 80-х років, грунтуючись на поняттях об'єкта (сутності), методу (дії) і атрибута (характеристики). Тут головну роль грають поліморфізм інкапсуляція, в той час, як в компонентно- орієнтованому підході більше значення надається метаінформації, наприклад, із застосуванням технології відображення. Хоча коріння об'єктно-орієнтованого проектування лежать в абстракції даних, так званий responsibility-driven design або проектування на основі <функціональної> відповідальності за SWEBOK може розглядатися як альтернатива об'єктно-орієнтованому проектуванню. Таке протиставлення - досить суперечливе питання, так як функціональна відповідальність настільки ж близька принципам сучасного об'єктно- орієнтованого проектування, наскільки і абстракція даних. Це питання еволюціонування поглядів і ступеня їх консерватизму.

10.5 Проектування на основі структур даних (Data-Structure-Centered Design)

У даному підході фокус сконцентрований переважно на структурах даних, якими управляє система, ніж на функціях системи. Інженери з програмного забезпечення часто спочатку описують структури даних входів (inputs) і виходів (outputs), а, потім, розробляють структуру управління цими даними (або,

наприклад, їх трансформації).

10.6 Компонентне проектування (Component-Based Design)

Програмні компоненти є незалежними одиницями, які володіють однозначно-визначеними (well-defined) інтерфейсами і залежностями (зв'язками) і можуть збиратися і розгортатися незалежно один від одного. Даний підхід покликаний вирішити завдання використання, розробки та інтеграції таких компонент з метою підвищення повторного використання активів (як архітектурних, так і у формі коду). Компонентно-орієнтоване проектування є однією з найбільш динамічно розвиваються концепцій проектування і може розглядатися як провісник і основа *сервісно- орієнтованого підходу (Service-Oriented Architecture, SOA)* в проектуванні. Зокрема, нотація UML 2.0 вже дозволяє вирішувати ряд питань, пов'язаних з візуальним поданням відповідних архітектурних рішень, де *сервіси (служби)* можуть розглядатися як публікована функціональність одиночних компонентів і груп компонентів, об'єднаних у більш "великі" блоки, що забезпечують надання відповідної сервісної функціональності.

10.7 Контрольні питання

1. Стратегії та методи проектування програмного забезпечення?
2. Що таке функціонально-орієнтоване або структурне проектування
3. Що таке проектування на основі структур даних?
4. Що таке об'єктно-орієнтоване проектування?
5. Що таке компонентне проектування?

Тема 11. Стандарти та інструментальні засоби при виборі архітектури програмного забезпечення

11.1 Інструментальні засоби

Для полегшення процесу розробки ПЗ використовується безліч автоматизованих інструментальних засобів. Деякі з них представляють собою колекцію класів з різними взаємозв'язками. Прикладами таких колекцій можуть служити Rational Rose від Rational Corporation і Together від Object International. Ці інструменти полегшують побудову об'єктних моделей, а також їх з'єднання з відповідним вихідним кодом і діаграмами. Для вибору інструментальних засобів моделювання складається список вимог до них. Цей процес аналогічний процесу аналізу вимог для розробки програмного додатку. ***Наведемо список деяких вимог до інструментів моделювання.***

^ [необхідно] Полегшення зображення об'єктних моделей і діаграм

> Швидке створення класів

> Легке редагування класів

> Зміна масштабу зображення усередині частин моделі

^ [бажано] Можливість швидкого переходу від об'єктної моделі до вихідного коду

^ [необхідно] Повинен коштувати не більше $ X для одного користувача.

^ [не обов'язково] Можливість зворотного проектування (тобто створення об'єктної моделі з вихідного коду).

11.2 Високорівневі та низькорівневі інструментальні засоби

Пакети інструментальних засобів найчастіше намагаються охопити і архітектуру, і детальне проектування, і реалізацію. Різні продавці розробляють системи з можливістю використання гіперпосилань між вихідним кодом і документацією. Інструментальні засоби, орієнтовані на реалізацію, подібні javadoc, можуть бути хорошим доповненням для процесу розробки. Клієнт - серверні

інструменти, такі як Powerbuilder, цілком придатні для визначення архітектури, хоча вони визначають і реалізації. Javadoc дуже корисний при навігації по пакетах, оскільки він надає алфавітний список всіх класів, а також їх ієрархію.

Інтерактивні середовища розробки (IDE) укомплектовані компіляторами і використовуються як інструменти часткового моделювання. Об'єктно- оріснтовані IDE в основному показують спадкування в ієрархічній формі. Цей факт привертає розробників через близькість цих інструментів до процесів компіляції та відлагодження. Однак IDE зазвичай мають недостатньо широкий спектр можливостей, щоб полегшити побудову архітектури та проектувальних роботу. Інструментальні засоби складання компонентів дозволяють створювати додатки за допомогою перетягування значків, що представляють елементи процесів. Середовища JavaBeans являють собою приклад таких інструментальних засобів. У таких середовищах об'єкти Java, класи яких відповідають стандарту Java Beans, можна взяти з бібліотек або створити самостійно і зв'язати через події. Стандарт Java Beans був створений для полегшення таких простих зборок за допомогою графічних інструментальних засобів. Основна незручність при використанні інструментів моделювання пов'язано із залежністю проекту від третьої сторони - продавця. Додатково до всього, крім складності самого додатка та проекту розробник повинен турбуватися про життєздатність продавця. Якщо продавець збанкрутує або оновлення для інструментальних засобів стане занадто дорогим - як це відіб'ється на проекті? Але незважаючи на все це популярність інструментів моделювання зростає в усьому світі. Тривалість їх використання обмежується продуктивністю і економічними факторами.

11.3 Стандарт 1EEE/AM81 для опису проекту

Стандарт IEEE 1016-1987 (знову затверджений в 1993 році) для проектної документації програмного забезпечення (SDD - Software Design Document) містить керівництво по складанню та ведення документації з розробки. Зміст цього документа надається далі. Керівництво, включене в стандарт IEEE 1016.1-1993, пояснює, яким чином SDD може бути складена для різних стилів архітектур. Розділи стандарту 1-5 можуть бути віднесені до архітектури програмного забезпечення, а розділ 6 - до детального проектування.

1. Введення

1.1. Ціль

1.2.Опис проекту

1.3. Визначення, скорочення та терміни

2. Посилання

3. Опис дскомпозиції

3.1 .Модульна декомпозиція

3.1.1. Опис модуля 1

3.1.2. Опис модуля 2

3.2. Декомпозиція на паралельні процеси

3.2.1. Опис процесу 1

3.2.2. Опис процесу 2

3.3. Декомпозиція даних

3.3.1. Опис блока даних 1

3.3.2. Опис блока даних 2

4. Опис залежностей

4.1. Міжмодульні залежності 4*

4.2. Міжпроцесні залежності

4.3. Залежності всередині даних

5. Опис інтерфейсу

5.1. Модульний інтерфейс

5.1.1. Опис модуля 1

5.1.2. Опис модуля 2

5.2. Інтерфейс процесів

5.2.1. Опис процесу 1

5.2.2. **Опис процесу 2 6.**

Детальне проектування

6.1. Детальне проектування модулів

6.1.1. Модуль 1: деталі

6.1.2. Модуль 2: деталі

6.2. Детальне проектування даних

6.2.1. Блок даних 1: деталі

6.2.2. Блок даних 2: деталі

11.4 Контроль якості при виборі архітектури

Персонал, який здійснює контроль якості, повинен брати участь в оцінках архітектури. Крім того, він розробляє плани тестування для всіх компонентів архітектури, починаючи з того моменту, як ці компоненти визначені.

11.5 Метрики для вибору архітектури

Більшість додатків можуть бути реалізовані за допомогою різних архітектур. Деякі варіанти можуть бути набагато краще за інших. Такі важливі рішення, як вибір архітектури, не приймаються без первинної розробки та порівняння альтернатив. Запропоновані архітектури ретельно аналізуються, оскільки усунення дефектів на ранній стадії коштує набагато менше, ніж їх

виправлення під час реалізації проекту. У цьому розділі ми запропонуємо метрики для вибору архітектури, а в наступному розглянемо приклади вибору найбільш підходящої архітектури. Один із способів вибору - привласнити ваги необхідних характеристиках і призначити нечіткий коефіцієнт якості для кожного кандидата.

11.6 Перевірка архітектури за допомогою варіантів використання

Варіанти використання отримують з вимог замовника. Тому варіанти використання не можуть враховувати архітектуру програми з простої причини - вона ще не визначена. Після вибору архітектури корисно повернутися до розгляду варіантів використання і перевірити, чи адекватно їх підтримує архітектура.

11.7 Контрольні питання

1. Стандарти при виборі архітектури програмного забезпечення.
2. Інструментальні засоби при виборі архітектури програмного забезпечення.
3. Стандарт 1EEE/AM81 для опису проекту
4. Контроль якості при виборі архітектури?
5. Метрики для вибору архітектури?
6. Перевірка архітектури за допомогою варіантів використання?

Тема 12. Патерни в розробці програмного забезпечення

12.1 Визначення та класифікація патернів

При реалізації проектів з розробки програмних систем і моделювання бізнес-процесів зустрічаються ситуації, коли рішення проблем у різних проектах мають подібні структурні риси. Спроби виявити схожі схеми або сіруктури в рамках об'єктно-орієнтованого аналізу і проектування привели до появи поняття патерну,

яке з абстрактної категорії перетворилося на неодмінний атрибут сучасних CASE-засобів. Патерни розрізняються ступенем деталізації та рівнем абстракції. *Пропонується наступна загальна класифікація патернів за категоріями їх застосування:*

^ *Архітектурні патерни (Architectural patterns)* - безліч попередньо визначених підсистем зі специфікацією їх відповідальності, правил і базових принципів встановлення відносин між ними. Призначені для специфікації фундаментальних схем структуризації програмних систем. Найбільш відомими і Штернами цієї категорії є патерни GRASP (General Responsibility Assignment Software Patter). Патерни цієї категорії систематизував і описав К. Ларман.

^ *Патерни проектування (Design patterns)* спеціальні схеми для у точнення структури підсистем або компонентів програмної системи і відносин між ними. Описують загальну структуру взаємодії елементів програмної системи, які реалізують вихідну проблему проектування в конкретному контексті. Найбільш відомими патернами цієї категорії є патерни GoF (Gang of l our), названі на честь Е. Гамми, Р. Хелма, Р. Джонсона і Дж. Вліссідес, які і истематизували їх і представили загальний опис. Патерни GoF включають в себе 23 патерни. Ці патерни не залежать від мови реалізації, але їх реалізація іалежить від області додатка.

^ *Патерни аналізу (Analysis patterns)* - спеціальні схеми для представлення загальної організації процесу моделювання. Відносяться до однієї або декількох предметних областей і описуються в термінах предметної області. Найбільш відомими патернами цієї групи є патерни бізнес-моделювання ARIS (Architecture of Integrated Information Systems), які характеризують абстрактний рівень представлення бізнес-процесів.

^ *Патерни тестування (Test patterns)* - спеціальні схеми для представлення загальної організації процесу тестування програмних систем. До цієї категорії патернів відносяться такі патерни, як тестування чорного ящика, білого ящика, окремих класів, системи. Патерни цієї категорії систематизував і

описав М. Гранд. Деякі з них реалізовані в інструментальних засобах, найбільш відомими з яких є IBM Test Studio. У зв'язку з цим патерни тестування іноді називають стратегіями або схемами тестування.

^ *Патерни реалізації (Implementation patterns)* - сукупність компонентів та інших елементів реалізації, що використовуються в структурі моделі при написанні програмного коду. Ця категорія патернів ділиться на наступні підкатегорії: патерни організації програмного коду, патерни оптимізації програмного коду, патерни стійкості коду, патерни розробки графічного інтерфейсу користувача. Патерни цієї категорії описані в роботах М. Гранда, К. Бека, Дж. Тідвелл. Деякі з них реалізовані в популярних інтегрованих середовищах програмування у формі шаблонів створюваних проектів.

12.2 Патерни проектування в нотації мови UML

У сфері розробки програмних систем найбільше застосування отримали патерни проектування GoF, деякі з них реалізовані в популярних середовищах ***ф*** програмування. При цьому патерни проектування можуть бути представлені в наочній формі за допомогою розглянутих позначень мови UML. Патерн проектування в контексті мови ІЖЬ являє собою параметризовану кооперацію риюм з описом базових принципів її використання. При зображенні патерну використовується позначення параметризовано'! кооперації мови UML (рис. 12.1), яка позначається пунктирним еліпсом. У правий верхній кут еліпса вбудований пунктирний прямокутник, в якому перераховані параметри кооперації, яка представляє той чи інший патерн.

Рисунок 12.1 - Зображення патерну у формі параметризованої кооперації

У подальшому параметри патерну можуть бути замінені різними класами, щоб отримати реалізацію патерну в рамках конкретної кооперації. Ці параметри специфікують використовувані класи у формі ролей класів у розглянутій підси- і гемі. При зв'язуванні або реалізації патерна будь-яка лінія позначається ім'ям параметра патерну, яке є ім'ям ролі відповідної асоціації. На додаток до діаграм кооперації особливості реалізації окремих патернів представляються за допомо- і ою діаграм послідовності. Патерни проектування дозволяють вирішувати різні шндання, з якими постійно стикаються проектувальники об'єктно-орієнтованих додатків.

12.3 Шаблон проектування «Модель-подання-контролер»

ModeI-view-controller (МУС, «Модель-уявлення-поведінка», «Модель-подання-контролер») — схема використання декількох шаблонів проектування (рис. 12.2), за допомогою яких модель даних програми, користувальницький Інтерфейс і взаємодія з користувачем розділені на три окремих компонента так, що модифікація одного з компонентів надає мінімальний вплив на інші. Дана схема проектування часто використовується для побудови архітектурного каркаса, коли переходять від теорії до реалізації в конкретній предметній області.

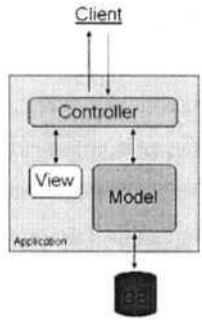

Рисунок 12.2 - Шаблон проектування MVC

Концепція MVC була описана в 1979 році Трюгве Реенскаугом (англ. Tryave Reenskaua), тоді працюючим над мовою програмування Smalltalk в Xerox PARC. Потім Джим Алтофф з командою розробників реалізували версію MVC для бібліотеки класів Smalltalk-80. В оригінальній концепції була описана сама ідея і роль кожного з елементів моделі, уявлення і контролера. Але зв'язки між ними були описані без конкретизації. *Крім того, розрізняли дві основні модифікації:*

1. Пасивна модель - модель не має жодних способів впливати на уявлення або контролер, і використовується ними як джерело даних для відображення. Всі зміни моделі відслідковуються контролером і він же відповідає за перемальовування уявлення, якщо це необхідно. Така модель частіше використовується в структурному програмуванні, так як в цьому випадку модель надає просто структуру даних, без методів їх обробних.

2. Активна модель - модель оповіщає уявлення про те, що в ній відбулися іміни, а уявлення, які зацікавлені в оповіщенні, підписуються на ці повідомлення. Це дозволяє зберегти незалежність моделі як від контролера, так і від уявлення.

Класичною реалізацією концепції MVC прийнято вважати версію саме з активною моделлю. З розвитком об'єктно-орієнтованого програмування та поняття про шаблони проектування був створений ряд модифікацій концепції MVC, які при

реалізації у різних авторів можуть відрізнятися від оригінальної. Так, наприклад, Еріан Берми в 2004 році описав приклад узагальненого МУС.

12.3.1 Призначення

Основна мета застосування цієї концепції полягає в поділі бізнес-логіки (моделі) від її візуалізації (уявлення, виду). За рахунок такого поділу підвищується можливість повторного використання. Найбільш корисне застосування даної концепції в тих випадках, коли користувач повинен бачити зі ж самі дані одночасно в різних контекстах та/або з різних точок зору. З*окрема, виконуються наступні завдання:*

^ До однієї моделі можна приєднати кілька видів, при цьому не зачіпаючи реалізацію моделі. Наприклад, деякі дані можуть бути одночасно представлені у вигляді електронної таблиці, гістограми і кругової діаграми.

^ Не торкаючись реалізацію видів, можна змінити реакції на дії користувача (натискання мишею на кнопці, введення даних), для цього досить використовувати інший контролер.

^ Ряд розробників спеціалізується тільки в одній з областей : або розробляють графічний інтерфейс, або розробляють бізнес - логіку. Тому можливо добитися того, що програмісти, що займаються розробкою бізнес-
логіки (моделі), взагалі не будуть інформовані про те, яке уявлення буде використовуватися.

12.3.2 Концепція

Концепція MVC дозволяє розділити дані, подання та обробку дій користувача на три окремих компонента :

^ Модель (англ. Model). Модель надає знання: дані і методи роботи з цими

даними, реагує на запити, змінюючи свій стан. Не містить інформації, як ці знання можна візуалізувати.

^ Представлення, вид (англ. View). Відповідає за відображення інформації (візуалізацію). Часто як уявлення виступає форма (вікно) з графічними елементами.

^ Контролер (англ. Controller). Забезпечує зв'язок між користувачем і системою : контролює введення даних користувачем і використовує модель та подання для реалізації необхідної реакції.

Для реалізації схеми Model-View-Controller використовується досить велика кількість шаблонів проектування (залежно від складності архітектурного рішення), основні з яких «спостерігач», «стратегія», «компонувальних ».

12.4 Модель уявлення архітектури "4+1"

Досить важливу роль у розвитку підходів до опису програмної архітектури зіграла модель "4 +1" ("The 4+1 View Model of Architecture"), яка була запропонована Філіпом Кручтеном (Philippe Kruchten) з компанії Rational ще в 1995 році (рис. 12.3). Дана методика позиціонувалася, як спосіб опису архітектури систем, заснованих на активному використанні ПЗ, хоча ідеї, закладені в цю методику, можуть використовуватися і в більш широкому контексті архітектури підприємства - що, власне, і сталося на практиці.

Рисунок 12.3 - Модель "4+1

Модель пропонує простий і зрозумілий спосіб опису архітектури складних систем, який полягає у використанні п'яти різних категорій або уявлень (views). *Чотирма основними уявленнями в цій методиці є наступні:*

^ *Логічне подання.* Є об'єктною моделлю проектування (в тому випадку, якщо використовується об'єктно-орієнтована модель проектування) - Основною метою логічного представлення в цій методиці є опис функціональних вимог: що система повинна виконувати в термінах кінцевих користувачів. Для цього подання використовуються різні абстрактні конструкції, такі як об'єкти і класи об'єктів. Для їх ілюстрування можуть пк застосовуватися діаграми класів (в нотації мови UML) або, наприклад, діаграми "сутність - зв'язок", якщо в розробці програми домінують дані.

^ *Процесне уявлення.* Описує питання паралельного виконання і ісинхронізації процесів - враховує деякі нефункціональні вимоги до системи, пт мочаючи продуктивність і доступність.

^ *Фізичне представлення.* Описує розміщення програмних компонент ■ нсгсми на апаратних платформах і аспекти, пов'язані з фізичним розташуванням системи — Фізичне представлення, в основному, розглядає нефункціональні вимоги, такі як доступність, надійність, стійкість, продуктивність, масштабованість.

^ *Представлення рівня розробки.* Описує статичну організацію програмної системи в середовищі розробки - Подання рівня розробки описує фактичну організацію модулів системи, поділ її на підсистеми, які можуть розроблятися незалежно.

Це подання в якомусь сенсі є надлишковим і перетинається з чотирма попередніми, але воно важливе з наступних причин: 1. Сценарії використання дозволяють ідентифікувати елементи архітектури, які потрібні для ефективно працюючої системи. 2. За допомогою сценаріїв можна виконувати перевірку і ілюстрацію того, що архітектура є працездатною і повною. Це також є основою для проведення тестування архітектурного прототипу.

12.5 Контрольні питання

1. Загальна класифікація патернів за категоріями їх застосуванн?
2. Поняття шаблону проектування «Модель-подання-контролер».
3. Яка концепція та призначення шаблону проектування «Модель-подання-контролер»?
4. Що таке модель уявлення архітектури "4+1"?

Тема 13. Архітектурні шаблони і стилі

13.1 Поняття архітектурного стилю

Архітектурний стиль, іноді званий *архітектурним шаблоном* - це набір принципів, високорівнева схема, що забезпечує абстрактну інфраструктуру для сімейства систем Архітектурний стиль покращує секціонування і сприяє повторному використанню дизайну завдяки забезпеченню рішень часто зустрічаються проблем. Архітектурні стилі і шаблони можна розглядати як набір принципів, що формують додаток.

Девид Гарпан (David Garlan) и Мери Шоу (Mary Shaw) визначають архітектурний стиль як: «...сімейство систем з точки зору схеми організації структури. Точніше кажучи, архітектурний стиль визначає набір компонентів і їгднань, які можуть використовуватися в примірниках цього стилю, а також ряд обмежень за їх можливим сполученням. Сюди можуть відноситися топологічні обмеження на архітектурні рішення (наприклад, не використовувати цикли). Опис стилю також може включати й інші обмеження, іакі як, скажімо, необхідність обробки семантики виконання».

Розуміння архітектурних стилів забезпечує декілька переваг. Найголовніше з них - спільна мова. Також вони дають можливість вести діалог, не торкаючись

технологій, тобто обговорювати схеми і принципи, не вдаючись у деталі. Наприклад, архітектурні стилі дозволяють порівнювати схему клієнт / сервер з n-рівневої схемою додатки. Архітектурні стилі можна організувати по їх фокусу. *Далі перераховані основні фокусні області та відповідні архітектурні стилі:*

Категорія	Архітектурні стилі
зв'язок	Сервісно-орієнтована архітектура (SOA), шина повідомлень
розгортання	Клієнт / сервер, N-рівнева, 3-рівнева
предметна область	Дизайн на основі предметної області (Domain Driven Design)
структура	Компонентна, об'єктно-орієнтована, багаторівнева архітектура

13.2 Огляд основних архітектурних стилів

Список типових архітектурних стилів:

Архітектурний стиль/парадигма	Опис
Клієнт / сервер	Система поділяється на два додатки, де клієнт виконує запити до сервера. У багатьох випадках в ролі сервера виступає база даних, а логіка додатки представлена процедурами зберігання.
Компонентна архітектура	Дизайн додатка розкладається на функціональні або логічні компоненти з можливістю повторного використання, предоставляющие ретельно опрацьовані інтерфейси зв'язку.
Дизайн на основі предметної області '	Об'єктно-орієнтований архітектурний стиль, орієнтований на моделювання сфери діловоі активності і визначальний бізнес-об'єкти на підставі сутностей цієї сфери.
Багатошарова архітектура	Функціональні області додатку поділяються на багатошарові групи (рівні).
Шина повідомлень	Архітектурний стиль, який наказував використаній програмної системи, яка може приймати і відправляти повідомлення по одному або більше каналах зв'язку, так що додатки отримують можливість взаємодіяти, не маючи конкретними відомостями один про одного.
N-рівнева / 3-рівнева	Функціональність виділяється в окремі сегменти, в чому аналогічно багатошаровому стилю, але в даному випадку сегменти фізично розташовуються на різних комп'ютерах.
Об'єктно-орієнтована	Парадигма проектування, заснована на розподілі відповідальності додатки або системи між окремими багаторазово використовуваними і самостійними об'єктами, що містять дані і поведінку.
Сервісно-орінетрірованная архітектура (SOA)	Описує додатки, що надають і споживають функціональність у вигляді сервісів за допомогою контрактів і повідомлень.

13.3 Поєднання архітектурних стилів

Архітектура програмної системи практично ніколи не обмежена лише одним архітектурним стилем, часто вона є поєднанням архітектурних стилів, що утворюють

повну систему. Наприклад, може існувати БОА-дизайн, що складається із сервісів, при розробці яких використовувалася багатошарова архітектура і об'єктно-орієнтована архітектурний стиль. Поєднання архітектурних стилів також корисно при побудові Інтернет Веб-додатків, де можна досягти ефективного розподілу функціональності за рахунок застосування багатошарового архітектурного стилю. Таким чином можна відокремити логіку подання від бізнес-логіки і логіки доступу до даних. Вимоги безпеки організації можуть обумовлювати або 3 - рівневе розгортання програми, або розгортання з більш ніж трьома рівнями. Рівень представлення може розгортатися у прикордонній мережі, що розташовується між внутрішньою мережею організації та зовнішньою мережею. Як модель взаємодії на рівні подання може застосовуватися шаблон подання з відділенням (різновид багатошарового стилю), така як Model View Controller (MVC). Також можна вибрати архітектурний стиль SOA і реалізувати зв'язок між Веб- сервером і сервером додатків за допомогою обміну повідомленнями. Створюючи настільний додаток, можна реалізувати клієнт, який буде відправляти запити до програми на сервері. У цьому випадку розгортання клієнта і сервера можна виконати за допомогою архітектурного стилю клієнт / сервер і використовувати компонентну архітектуру для подальшого розкладання дизайну на незалежні компоненти, що надають відповідні Інтерфейси. Застосування об'єктно-орієнтованого підходу до цих компонентів підвищить можливості повторного використання, тестування і гнучкість. На вибір архітектурних стилів впливає безліч факторів. Сюди входять здатність організації до проектування та реалізації, можливості і досвід розробників, а також обмеження інфраструктури та організації. Інформація, наведена в наступних розділах, допоможе при виборі відповідних стилів для додатків.

13.3.1 Архітектура клієнт/сервер

Клієнт/серверна архітектура описує розподілені системи, що складаються з

окремих клієнта і сервера і з'єднує їх мережі. Найпростіша форма системи клієнт / сервер, звана 2-рівневої архітектурою - це серверний додаток, до якого безпосередньо звертаються безліч клієнтів. Історично архітектура клієнт/сервер являє собою настільний додаток з графічним UI, обмінюватися даними з сервером бази даних, на якому у формі збережених процедур розташовується основна частина бізнес-логіки, або з виділеним файловим сервером. Якщо

розглядати більш узагальнено, архітектурний стиль клієнт/сервер описує відносини між клієнтом і одним або більше серверами, де клієнт ініціює один або більше запитів (можливо, з використанням графічного UI), очікує відповіді та обробляє їх при отриманні. Зазвичай сервер авторизує користувача і потім проводить обробку, необхідну для отримання результату. Для зв'язку з клієнтом сервер може використовувати широкий діапазон протоколів і форматів даних. *До інших різновидів стилю клієнт/сервер відносяться:*

^ **Системи клієнт-черга-клієнт.** Цей підхід дозволяє клієнтам обмінюватися даними з іншими клієнтами через чергу на сервері. Клієнти можуть читати дані з і відправляти дані на сервер, який виступає в ролі простої черги для зберігання даних. Завдяки цьому клієнти можуть розподіляти і синхронізувати файли і відомості. Іноді таку архітектуру називають пасивною чергою.

^ **Одноранговi (Peer-to-Peer, P2P) додатки.** Створений на базі клієнт-черга-клієнт, стиль P2P дозволяє клієнту і серверу обмінюватися ролями з метою розподілу і синхронізації файлів і даних між безліччю клієнтів. Ця схема розширює стиль клієнт / сервер, додаючи множинні відповіді на запити, спільно використовувані дані, виявлення ресурсів і стійкість при видаленні учасників мережі.

^ **Сервери додатків.** Спеціалізований архітектурний стиль, при якому додатки і сервіси розміщуються і виконуються на сервері, і тонкий клієнт виконує доступ до них через браузер або спеціальне встановлене на клієнті ПЗ. Прикладом є клієнт, який працюєте додатком, що виконується на сервері, через таке середовище як Terminal Services (Служби терміналів).

*Осно**вні переваги архітектурного стилю клієнт/сервер:***

^ **Велика безпеку.** Всі дані зберігаються на сервері, який зазвичай забезпечує більший контроль безпеки, ніж клієнтські комп'ютери.

^ **Централізований доступ до даних**. Оскільки дані зберігаються тільки на сервері, адміністрування доступу до даних набагато простіше, ніж у будь-яких інших архітектурних стилях.

^ **Простота обслуговування.** Ролі та відповідальність обчислювальної системи розподілені між декількома серверами, що спілкуються один з одним по мережі. Завдяки цьому клієнт гарантовано залишається необізнаним і не схильним до впливу подій, що відбуваються з сервером (ремонт, оновлення або переміщення).

13.3.2 Компонентна архітектура

Компонентна архітектура описує підхід до проектування і розробки систем з використанням методів проектування програмного забезпечення. Основна увага в цьому випадку приділяється розкладанню дизайну на окремі функціональні або логічні компоненти, що надають чітко визначені інтерфейси, містять методи, події і властивості. Забезпечується більш високий рівень абстракції, і не відбувається концентрації уваги на таких питаннях, як протоколи зв'язку або загальний стан. ***Основний принцип компонентного стилю - застосування компонентів, що володіють такими якостями:***

^ **Придатність для повторного використання.** Як правило, компоненти проектуються із забезпеченням можливості їх повторного використання в рі тих сценаріях різних додатків. Однак деякі компоненти створюються і спеціально для конкретного завдання.

^ **Заміннність.** Компоненти можуть без праці замінюватися іншими подібними компонентами.

^ **Незалежність від контексту.** Компоненти проектуються для роботи в рі зних

середовищах і контекстах. Спеціальні відомості, такі як дані про стан, Повинні не включатися або вилучатися компонентом, а передаватися в нього.

^Розширюваність. Компонент може розширювати існуючі компоненти для забезпечення нового поведінки.

^ Інкапсуляція. Компоненти надають інтерфейси, що дозволяють викликає стороні використовувати їх функціональність, не розкриваючи при цьому деталі внутрішніх процесів або внутрішні змінні або стан.

^Незалежність. Компоненти проектуються з мінімальними залежностями від інших компонентів. Таким чином, компоненти можуть бути розгорнуті в будь-якої зручної середовищі без впливу на інші компоненти або системи.

Основні переваги компонентного архітектурного стилю:

^ Простота розгортання. Існуючі версії компонентів можуть замінюватися новими сумісними версіями, не надаючи впливу на інші компоненти або систему в цілому.

^ Менша вартість. Використання компонентів сторонніх виробників дозволяє розподіляти витрати на розробку та обслуговування.

^Простота розробки. Для забезпечення заданої функціональності компоненти реалізують широко відомі інтерфейси, що дозволяє вести розробку без впливу на інші частини системи.

^ Можливість повторного використання. Застосування багаторазово використовуваних компонентів означає можливість розподілу витрат на розробку та обслуговування між кількома додатками або системами.

^ Спрощення з технічної точки зору. Компоненти спрощують систему через використання контейнера компонентів і його сервісів. Як приклади сервісів, що надаються контейнером, можна навести активацію компонентів, управління життєвим циклом, організацію черги викликів методів, обробку подій і транзакції.

13.3.3 Проектування на основі предметної області

Проектування на основі предметної області (Domain Driven Design, DDD) об'єктно-орієнтований підхід до проектування ПрО, заснований на предметній області, її елементах, поведінці і відносинах між ними. Метою є створення програмних систем, які є реалізацією лежить в основі предметної області, шляхом визначення моделі предметної області, вираженої мовою фахівців у цій галузі. Модель ПрО може розглядатися як каркас, на підставі якого будуть реалізовуватися рішення. Для застосування DDD необхідно чітко розуміти предметну область, яку передбачається моделювати, або мати здібності для оволодіння такими знаннями. При створенні моделі предметної області група розробки нерідко працює у співпраці з фахівцями в даній області. Архітектори, розробники і фахівці в даній області мають різною підготовкою і в багатьох ситуаціях використовуватимуть різні мови для опису своїх цілей, бажань і вимог. Проте в рамках DDD вся група домовляється використовувати тільки одна мова, орієнтований на предметну область і виключає всі технічні жаргонізми. Як ядро ПЗ виступає модель предметної області, яка є прямою проекцією цього загального мови; з її допомогою шляхом аналізу мови група швидко знаходить прогалини в ПЗ. Створення спільної мови це не просто вправа з отримання відомостей від фахівців та їх застосування. Досить часто в групах виникають проблеми з обміном інформацією не тільки з причини нерозуміння мови предметної області, але також і через невизначеність мови самого по собі. Процес DDD має на меті не тільки реалізацію використовуваної мови, але також поліпшення та уточнення мови предметної області. Це, в свою чергу, позитивно відбивається на створюваному ПрО, оскільки модель є прямою проекцією мови предметної області.

Основними перевагами стилю DDD є:

^ *Обмін інформацією.* Всі учасники групи розробки можуть використовувати модель предметної області та описувані нею сутності для передачі відомостей і вимог предметної області за допомогою спільної мови

предметної області, не вдаючись до технічного жаргону.

^ **Розширюваність.** Модель предметної області часто є модульною і гнучкою, що спрощує оновлення і розширення при зміні умов і вимог.

^**Зручність тестування.** Об'єкти моделі предметної області характеризуються слабкою пов'язаністю і високої зв'язність, що полегшує їх тестування.

13.3.4 Багатошарова архітектура

Багаторівнева архітектура забезпечує угруповання пов'язаної функціональності додатку в різних шарах, що вишиковуються вертикально, поверх один одного. Функціональність кожного шару об'єднана спільною роллю або відповідальністю. Шари слабо пов'язані, і між ними здійснюється явний обмін даними. Правильне поділ програми на шари допомагає підтримувати суворе поділ функціональності, що в свою чергу, забезпечує гнучкість, а також зручність і простоту обслуговування. Багатошарова архітектура описана як перевернута піраміда повторного використання, в якій кожен шар агрегує відповідальності та абстракції рівня, розташованого безпосередньо під ним. При строгому поділі на шари компоненти одного шару можуть взаємодіяти тільки з компонентами того ж шару або компонентами шару, розташованого прямо підданим шаром. Більш вільний поділ на шари дозволяє компонентам шари взаємодіяти з компонентами того ж і всіх нижчих шарів. Шари додатки можуть розміщуватися фізично на одному комп'ютері (на одному рівні) або бути розподілені по різних комп'ютерів (п- рівнів), і зв'язок між компонентами різних рівнів здійснюється через строго певні інтерфейси.

Загальні принципи проектування з використанням багатошарової архітектури:

^ **Абстракція.** Багатошарова архітектура представляє систему як єдине ціле, забезпечуючи при цьому досить деталей для розуміння ролей і иідповідальностей окремих верств і відносин між ними.

^ **Інкапсуляція**. Під час проектування немає необхідності робити будь-які припущення про типи даних, методи і властивості або реалізації, оскільки всі ці деталі приховані в рамках шару.

^ **Чітко визначені функціональні шари.** Поділ функціональності між шарами дуже чітка. Верхні шари, такі як шар уявлення, посилають команди нижнім верствам, таким як бізнес-шар і шар даних, і можуть реагувати на події, що виникають в цих шарах, забезпечуючи можливість передачі даних між шарами вгору і вниз.

^**Висока зв'язність.** Чітко визначені межі відповідальності для кожного шару і гарантоване включення в шар тільки функціональності, безпосередньо пов'язаної з його завданнями, допоможе забезпечити максимальну зв'язність в рамках шару.

^**Можливість повторного використання.** Відсутність залежностей між нижніми і верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.

^**Слабке зв'язування**. Для забезпечення слабкого зв'язування між шарами зв'язок між ними грунтується на абстракції і події.

Прикладами багатошарових додатків можуть служити бізнес-додатки(line-of-business, LOB), такі як системи бухгалтерського обліку та управління замовниками; Веб-додатки та Веб-сайти підприємств.

13.3.5 Архітектура, заснована на шині повідомлень

Заснована на шині повідомлень архітектура описує принцип використання програмної системи, яка може приймати і відправляти повідомлення по одному або більше каналах зв'язку, забезпечуючи, таким чином, додаткам можливість взаємодії без необхідності знання конкретних деталей один про одного. Це стиль проектування, в якому взаємодії між додатками здійснюються шляхом передачі (зазвичай асинхронної) повідомлень через загальну шину. У типових реалізаціях архітектури, заснованої на шині повідомлень, використовується або маршрутизатор повідомлень, або шаблон Publish/Subscribe (Публікація/Підписка) і система обміну

повідомленнями, така як Message Queuing (Черга повідомлень). Багато реалізації складаються з окремих додатків, обмін даними між якими здійснюється шляхом відправки і прийому повідомлень за загальними схемами та інфраструктурі. *Шина повідомлень забезпечує можливість обробляти :*

^Засноване на повідомленних взаємодію. Вся взаємодія між додатками грунтується на повідомленнях, використовують відомі схеми.

^ Складну логіку обробки. Складні операції можуть виконуватися як частина багатокрокового процесу шляхом поєднання низки менших операцій, кожна з яких підтримує певні завдання.

^ Зміни логіки обробки. Взаємодія з шиною реалізується за загальними схемами і з застосуванням звичайних команд, що забезпечує можливість вставки або видалення додатків на шині для зміни використовуваної для обробки повідомлень логіки.

^Інтеграцію з різними інфраструктурами. Використання моделі зв'язку допомогою повідомлень, заснованої на загальних стандартах, дозволяє взаємодіяти з додатками, розробленими для різних інфраструктур, таких як Microsoft. NET і Java.

Шини повідомлень використовуються для забезпечення складних правил обробки вже протягом багатьох років. Такий дизайн забезпечує архітектуру, що підкачується, яка дозволяє вводити додатки в процес або покращувати масштабованість, підключаючи до шини кілька примірників одного і того ж додатка. *До різновидів шини повідомлень відносяться:*

^Сервісна шина підприємства (Enterprise Service Bus, ESB). ESB грунтується на шині повідомлень і використовує сервіси для обміну даними між шиною і компонентами підключеними до шини. Зазвичай ESB забезпечує сервіси для перетворення одного формату в іншій, забезпечуючи можливість зв'язку між клієнтами, що використовують несумісні формати повідомлень.

^ Шина Інтернет- сервісів (Internet Service Bus, 1SB). Подібна до сервісної шині підприємства, але додатки розміщуються не в мережі підприємства, а в хмарі.

Основна ідея ISB - використання Уніфікованих ідентифікаторів ресурсів (Uniform Resource identifiers, URIs) і політик, керуючих логікою маршрутизації через програми та сервіси в хмарі.

^ **Розширюваність.** Можливість додавати або видаляти програми з шини без впливу на існуючі програми.

^ **Невисока складність.** Додатки спрощуються, тому що кожному з них необхідно знати лише, як обмінюватися даними з шиною.

Гнучкість. Приведення набору додатків, складових складний процес, або схем зв'язку між додатками у відповідність мінливих бізнес-вимогам або вимогам користувача просто шляхом внесення змін в конфігурацію або параметри, керуючі маршрутизацією.

^ **Слабке зв'язування.** Крім надається додатком інтерфейсу для зв'язку з шиною повідомлень, немає ніяких інших залежностей з самим додатком, що забезпечує можливість зміни, поновлення і заміни його іншим додатком, що надають, такий же інтерфейс.

^Масштабованість. Можливість підключення до шини безлічі екземплярів однієї програми для забезпечення одночасної обробки безлічі запитів.

^Простота додатки. Незважаючи на те, що реалізація шини повідомлень ускладнює інфраструктуру, кожному додатку доводиться підтримувати лише одне підключення до шини повідомлень, а не безліч підключень до інших додатків.

13.3.6 N-рівнева/3-рівнева архітектура

N-рівнева і 3-рівнева архітектура є стилями розгортання, що описують поділ функціональності на сегменти, в чому аналогічно багатошарової архітектурі, але в даному випадку ці сегменти можуть фізично розміщуватися на різних комп'ютерах, їх називають рівнями. Дані архітектурні стилі були створені на базі компонентно-орієнтованого підходу і, як правило, для зв’язку використовують

методи платформи, а не повідомлення. Характеристиками N- рівневої архітектури додатки є функціональна декомпозиція додатки, сервісні компоненти та їх розподілене розгортання, що забезпечує підвищену масштабованість, доступність, керованість і ефективність використання ресурсів. Кожен рівень абсолютно незалежний від всіх інших, крім тих, з якими він безпосередньо сусідить. 14- йому рівню потрібно лише знати, як обробляти запит від n +1 рівня, як передавати цей запит на n - 1 рівень (якщо такий є), і як обробляти результати запиту. Для забезпечення кращої масштабованості зв'язок між рівнями зазвичай асинхронна.

Основними перевагами М-рівневого/З-рівневого архітектурного стилю є:

^Зручність підтримки. Рівні не залежать один від одного, що дозволяв виконувати оновлення або зміни, не надаючи впливу на додаток в цілому.

^ Масштабованість. Рівні організовуються на підставі розгортання шарів, і ому масштабувати додаток досить просто.

^ Гнучкість. Управління та масштабування кожного рівня може виконуватися незалежно, що забезпечує підвищення гнучкості.

^Доступність. Додатки можуть використовувати модульну архітектуру, мка дозволяє використовувати в системі легко масштабовані компоненти, що підвищує доступність.

13.3.7 Об'єктно-орієнтована архітектура

Об'єктно-орієнтована архітектура - це парадигма проектування, заснована на поділі відповідальностей додатки або системи на самостійні придатні для повторного використання об'єкти, кожен з яких містить дані і поведінку, що відносяться до цього об'єкту. При об'єктно-орієнтованому проектуванні система розглядається не як набір підпрограм і процедурних команд, а як набори взаємодіючих об'єктів. Об'єкти відособлені, незалежні і слабо пов'язані; обмін даними між ними відбувається через

інтерфейси шляхом виклику методів і властивостей інших об'єктів і відправлення/прийому повідомлень. *Основними принципами об'єктно-орієнтованого архітектурного стилю є:*

^Абстракція. Дозволяє перетворити складну операцію в узагальнення, що зберігає основні характеристики операції. Наприклад, абстрактний інтерфейс може бути широко відомим описом, що підтримує операції доступу до даних через використання простих методів, таких як Get (Отримати) і Update (Оновити). Інша форма абстракції - метадані, що використовуються для забезпечення зіставлення двох форматів структурованих даних.

^Композиція. Об'єкти можуть бути утворені іншими об'єктами і за бажанням можуть приховувати ці внутрішні об'єкти від інших класів або надавати їх як прості інтерфейси.

^ Успадкування. Об'єкти можуть успадковуватися від інших об'єктів і використовувати функціональність базового об'єкта або перевизначати її для реалізації нової поведінки. Більш того, спадкування спрощує обслуговування та оновлення, оскільки зміни, що вносяться в базовий об'єкт, автоматично поширюються на всі успадковані від нього об'єкти.

^ Інкапсуляція. Об'єкти надають функціональність тільки через методи, властивості і події і приховують внутрішні деталі, такі як стан і змінні, від інших об'єктів. Це спрощує оновлення або заміну об'єктів і дозволяє виконувати ці операції без впливу на інші об'єкти і код, потрібно лише забезпечити сумісні інтерфейси.

^ Поліморфізм. Дозволяє перевизначати поведінку базового типу, що підтримує операції в додатку, шляхом реалізації нових типів, які є взаємозамінними для існуючого об'єкта.

^ Відділення. Об'єкти можуть бути відокремлені від споживача шляхом визначення абстрактного інтерфейсу, реалізованого об’єктом і зрозумілого споживачеві. Це дозволяє забезпечувати альтернативні реалізації, не надаючи впливу на споживачів інтерфейсу.

До основних переваг об'єктно-орієнтованої архітектури належать:

^ Зрозумілість. Забезпечується більш близьке відповідність додатки реальним об'єктам, що робить його більш зрозумілим.

^ Можливість повторного використання. Забезпечується можливість повторного використання через поліморфізм і абстракцію.

^Тестування. Забезпечується поліпшене тестування через інкапсуляцію.

^ Розширюваність. Інкапсуляція, поліморфізм і абстракція гарантують, що зміни в поданні даних не вплинуть на інтерфейси, що надаються об'єктами, що могло б обмежити можливості зв'язку і взаємодії з іншими об'єктами.

^ Висока зв'язність. Розміщуючи в об'єкті тільки функціонально близькі методи та функції і використовуючи для різних наборів функцій різні об'єкти, можна досягти високого рівня зв'язності.

13.3.8 Сервісно-орієнтована архітектура (SOA)

Сервісно-орієнтована архітектура (Service-oriented architecture, SOA) забезпечує можливість надавати функціональність програми у вигляді набору сервісів і створювати додатки, що використовують програмні сервіси. Сервіси слабо пов'язані, тому що використовують засновані на стандартах інтерфейси, які можуть бути викликані, опубліковані і виявлені. Основне завдання сервісів в SOA-надання схеми та взаємодії з додатком допомогою повідомлень через інтерфейси, областю дії яких є додаток, а не компонент або об'єкт. Не слід розглядати SOA-сервіс як компонентний постачальник сервісів. SOA- архітектура може забезпечити упаковку бізнес-процесів в сервіси, що підтримують можливість взаємодії і використовують для передачі інформації широкий діапазон протоколів і форматів даних. Клієнти та інші сервіси можуть виконувати доступ до локальних сервісів, що виконується на тому ж рівні, або до віддалених сервісів по мережі. ***Основними принципами архітектурного стилю SOA є:***

^ **Сервіси автономні.** Обслуговування, розробка, розгортання і контроль версій кожного сервісу відбувається незалежно від інших.

^ **Сервіси можуть бути розподілені.** Сервіси можуть розміщуватися в будь-якому місці мережі, локально або віддалено, якщо мережа підтримує необхідні протоколи зв'язку.

^ **Сервіси слабо пов'язані**. Кожен сервіс абсолютно не залежить від інших і може бути замінений або оновлений без впливу на додатки, його використовують, за умови надання сумісного інтерфейсу.

^ **Сервіси спільно використовують схему і контракт, але не клас. При** обміні даними сервіси спільно використовують контракти і схеми, але не внутрішні класи.

^ **Сумісність заснована на політиці.** Політика, в даному випадку, означає опис характеристик, таких як транспорт, протокол і безпеку.

Основними перевагами SOA-архітектури є:

^ **Узгодження предметних областей.** Повторне використання загальних сервісів зі стандартними інтерфейсами розширює технологічні та бізнес- можливості, а також скорочує вартість.

^ **Абстракція.** Сервіси є автономними, доступ до них здійснюється за формальною контрактом, що забезпечує слабке зв'язування і абстракцію.

^ **Можливість виявлення.** Сервіси можуть надавати опису, що дозволяє іншим програмам і сервісам виявляти їх і автоматично визначати інтерфейс.

^ **Можливість взаємодії.** Оскільки протоколи та формати даних грунтуються на галузевих стандартах, постачальник і споживач сервісу можуть створюватися і розгортатися на різних платформах.

^ **Раціоналізація.** Сервіси забезпечують певну функціональність, усуваючи необхідність її дублювання в додатках.

13.4 Контрольні питання

1. Поняття архітектурного стилю.
2. Перечисліть основні архітектурних стилі. їх особливості та принципи.

Тема 14. Аналіз якості та оцінка програмного дизайну. Нотації та засоби підтримки проектування

14.1 Атрибути якості (Quality Attributes)

Існує цілий спектр різних атрибутів, що допомагають оцінити і домогтися якісного дизайну. Ці атрибути можуть описувати багато характеристик системи та елементів дизайну — "тестування", "переносимість", " модифікованість ", "продуктивність", "безпека" і т.п. Важливо розуміти, що обговорювані атрибути стосуються тільки дизайну (як результату), але не проектування (як процесу). *В принципі, всі ці атрибути можна розбити на кілька груп:*

^ застосовні до run-time, тобто до часу виконання системи; наприклад, середній час відгуку системи дозволяє оцінити якість дизайну з точки зору продуктивності;

^ орієнтовані на design-time, тобто дозволяють оцінювати якість одержуваного дизайну ще на етапі проектування або, в загальному випадку, аж до тестування, включно; наприклад, середня навантаженість класів бізнес- методами;

^ атрибути якості архітектурного дизайну як такого, наприклад, концептуальна цілісність дизайну, несуперечливість, повнота, завершеність; наприклад, будь-який визначений бізнес-метод є викликається, тобто створений не просто тому що може знадобитися в майбутньому, а визначений відповідно до вимог або необхідний для реалізації дизайну в обраному архітектурному стилі.

Необхідно розуміти, що існують атрибути, які складно виміряти. Наприклад, портування або безпека. Не варто плутати атрибути якості дизайну з атрибутами

якості, які фігурують ряд вимог, що пред'являються до системи. Частина з них може відображатися один на одного і нести еквівалентне
смислове навантаження, деякі можуть бути пов'язані, велика частина атрибутів є специфічною саме для дизайну і не пов'язана з вимогами. Наприклад, якщо ми використовуємо платформу J2EE (Java 2 Enterprise Edition) і орієнтуємося на використання компонентою моделі EJB (Enterprise JavaBeans), існують ознаки хорошого дизайну, специфічні для даної платформи і компонентної моделі, але абсолютно ніяк не зв'язані з якими-небудь вимогами до створюваної на цій платформі програмній системі. Якщо повернутися до вимірюваних атрибутам якості, вони описуються певними метриками . Наведений вище приклад з кількістю бізнес-методів на клас є метрикою, яка дозволяє оцінити атрибути якості "модифікованості" і "складність" системи .

14.2 Аналіз якості і техніки оцінки (Quality Analysis and Evaluation Techniques)

В індустрії поширені багато інструментів, техніки та практики, що допомагають домогтися якісного дизайну:

^ огляд дизайну (software design review); наприклад, неформальний огляд архітектури членами проектної команди;

^ статичний аналіз (static analysis); наприклад , трасування з вимогами;

^ симуляція і прототипування (simulation and prototyping) - динамічні техніки перевірки дизайну в цілому або окремих його атрибутів якості; наприклад, для оцінки продуктивності використовуваних архітектурних рішень при симуляції навантаження , близькою до прогнозованим піковим.

14.3 Вимірювання (Measures)

Також відомі як метрики. Можуть бути використані для кількісної оцінки очікувань щодо різних аспектів конкретного дизайну, наприклад, розміру <проекту>, структури (її складності) або якості (наприклад, в контексті вимог, що пред'являються до продуктивності). Найчастіше, все метрики поділяють за двома категоріями: функціонально-орієнтовані і об'єктно-орієнтовані.

14.4 Нотації проектування (Software Design Notations)

Нотація є угода про представлення. Часто під нотацією розуміють візуальне (графічне) подання. ***Нотація може задаватися:***

^ стандартом; наприклад, OMG UML - Unified Modeling Language, розвивається консорціумом OMG (Object Management Group, http://www.omg.org):

^ загальноприйнятою практикою; наприклад, в extreme Programming часто використовуються картки функціональної відповідальності та зв'язків класу - Class Responsibility Collaborator або CRC Card (CRC за своєю природою є текстовою, тобто невізуальною нотацією);

^ внутрішнім методом проектної команди ("будемо малювати і позначати так...").

Певні нотації використовуються на стадії концептуального проектування, ряд нотацій орієнтований на створення детального дизайну, більшість може використовуватися на обох стадіях. Крім того, нотації найчастіше використовують в контексті (вибір нотації може бути обумовлений таким контекстом) застосовуваної методології або підходу.

14.5 Структурні описи, статичний погляд (Structural Descriptions, static view)

Наступні нотації, в основному (але, не завжди), є графічними, описуючи і

представляючи структурні аспекти програмного дизайну. Найчастіше вони стосуються основних компонентів і зв'язків між ними (статичних зв'язків, наприклад, таких як відносини "один-до-багатьох").

^ Мови опису архітектури (Architecture description language, ADL): текстові мови, часто - формальні, які використовуються для опису програмної архітектури в термінах компонентів і конекторів (спеціалізованих компонентів, реалізують не функціональність, але забезпечують взаємозв'язок функціональних компонентів між собою і з "зовнішнім світом");

^ Діаграми класів та об'єктів (Class and object diagrams): використовуються для представлення набору класів і <статичних> зв'язків між ними (наприклад, успадкування);

^ Діаграми компонентів або компонентні діаграми (Component diagrams); ***в*** певній степені аналогічні діаграмам класів, однак, в силу специфіки концепції або поняття компонента*, звичайно, представляються в іншій візуальній формі. Тут необхідно відмітити різницю в поняттях класу (або об'єкта) і компонента: компонент розглядається як фізично реалізований елемент програмного забезпечення, несущій <в певній степені> самодостатню логіку і реалізований як конгломерат інтерфейсу і його реалізації (часто, у виді комплексу класів);

^ Картки <функціональиої> відповідальності та зв'язків класу (Class responsibility collaborator card, CRC): використовуються для позначення імені класу, його відповідальності (тобто, що він повинен робити) і інших сутностей (класів, компонентів, акторів / ролей і т.п .), з якими він пов'язаний; часто їх називають картками "клас-обов'язок-кооперація";

^ Діаграми розгортання (Deployment diagrams): використовується для представлення (фізичних) вузлів, зв'язків між ними та моделювання інших фізичних аспектів системи;

^ Діаграми сутність-зв'язок (Entity-relationship diagram, ERD або ER): використовується для представлення концептуальної моделі даних, що зберігаються

в процесі роботи інформаційної системи;

v· Мови опису / визначення інтерфейсу (Interface Description Languages, ll)L): мови, подібні мов програмування, що не включають можливостей опису логіки системи і призначені для визначення інтерфейсів програмних компонентів (імен і типів експортуються або публікованих операцій);

^ Структурні діаграми Джексона (Jackson structure diagrams): використовуються для опису структур даних в термінах послідовності, вибору та ітерацій (повторень);

^ Структурні схеми (Structure charts): описують структуру викликів в програмах (який модуль викликає, ким і як викликаємо).

14.6 Поведінкові опису, динамічний погляд (Behavioral Descriptions, dynamic view)

Наступні нотації і мови (частина з яких - графічні, частина - текстові) використовуються для опису динамічної поведінки програмних систем та їх компонентів. Багато з цих нотацій успішно використовуються для проектування деталей дизайну, але не тільки для цього.

^ Діаграми діяльності або операцій (Activity diagrams): використовуються для опису потоків робіт і управління;

^ Діаграми співробітництва (Collaboration diagrams): показують динамічну взаємодію, що відбувається в групі об'єктів і приділяють особливу увагу об'єктам, зв'язків між ними та повідомленнями, якими обмінюються об'єкти допомогою цих зв'язків;

^ Діаграми потоків даних (Data flow diagrams, DFD): описують потоки даних усередині набору процесів (не в термінах процесів операційного середовища, але в розумінні обміну інформацією в бізнес-контексті);

^ Таблиці і діаграми <прийняття> рішень (Decision tables and diagrams): використовуються для представлення складних комбінацій умов і дій (операцій);

^ Блок-схеми та структуровані блок-схеми (Flowcharts and structured flowcharts): застосовуються для представлення потоків управління (контролю) і пов'язаних операцій;

^ Діаграми послідовності (Sequence diagrams): використовуються для показу взаємодій всередині групи об'єктів з акцентом на часовій послідовності повідомлень / викликів;

^ Діаграми переходу і карти станів (State transition and statechart diagrams): застосовуються для опису потоків управління переходами між станами;

^ Формальні мови специфікації (Formal specification languages): текстові мови, що використовують основні поняття з математики (наприклад, множини) для суворого і абстрактного визначення інтерфейсів і поведінки програмних компонентів, часто в термінах перед- і пост-умов;

Псевдокод і програмні мови проектування (Pseudocode and program design languages, PDL): мови, що використовуються для опису поведінки процедур і методів, в основному на стадії детального проектування; подібні структурним мовам програмування.

14.7 Контрольні питання

1. Що таке аналіз якості програмного дизайну.?
2. Основні оцінки програмного дизайну?
3. Які нотації та засоби підтримки проектування відомі?

Тема 15. Методи аналізу архітектури

15.1 Метод аналізу компромісних архітектурних рішень (комплексний підхід до оцінки архітектури)

Метод аналізу компромісних архітектурних рішень (Architecture Tradeoff Analysis Method, ATAM) - комплексна універсальна методика оцінки програмної архітектури. Відповідно до назви цей метод виявляє ступінь реалізації в архітектурі тих чи інших завдань за якістю, а також (виходячи з припущення про те, що будь-яке архітектурне рішення впливає відразу на декілька завдань за якістю) механізм їх взаємодії - іншими словами, їх взаємозамінність. Основне призначення ATAM полягає в тому, щоб виявити комерційні завдання, поставлені в контексті розробки системи і проектування архітектури. Укупі з участю зацікавлених осіб це допомагає фахівцям з оцінки сфокусуватися на тих елементах архітектури, які відіграють першорядну роль для реалізації згаданих завдань.

15.2 Етапи методу аналізу компромісних архітектурних рішень

Операції оцінки за методом A TAM розпадаються на чотири етапи.

^ На нульовому етапі - «Встановлення партнерських відносин і підготовка» керівники групи оцінки проводять неофіційні наради з основними відповідальними за проект особами та опрацьовують подробиці майбутньої роботи. Представники проекту присвячують фахівців з оцінки в суть проекту, тим самим підвищуючи кваліфікацію деяких з них. Ці дві групи беруть узгоджені логістичні рішення: де і коли зустрічатися, хто надасть лекційні плакати і т. д. Крім того, вони узгоджують попередній перелік зацікавлених осіб (перераховуючи їх не по іменах, а за ролями), встановлюють терміни та одержувачів зведеного звіту. Вони також організують постачання фахівців з оцінки архітектурною документацією - якщо, звичайно, така існує і може виявитися корисною. Нарешті, керівник групи оцінки пояснює

керівнику проекту та архітектору, яку інформацію їм слід надати на першому етапі, і при необхідності допомагає їм скласти відповідні презентації.

^ На першому та другому етапах проводиться безпосередньо оцінка - все занурені в аналітичні операції. До початку цих етапів учасники групи оцінки повинні ознайомитися з документацією з архітектури, отримати достатнє уявлення про систему, знати задіяні архітектурні методики і орієнтуватися в першорядних атрибутах якості. На першому етапі, маючи намір приступити до збору й аналізу інформації, учасники групи оцінки зустрічаються з особами, відповідальними за проект (як правило, зустріч триває весь день).

^ На другому етапі до фахівців приєднуються зацікавлені в архітектурі особи, і протягом приблизно двох днів вони проводять аналітичні заходи спільно.

^ Третій етап займає доробка - група оцінки становить письмово і надає одержувачам зведений звіт. По суті, учасники займаються самоперевіркою і вносять в результати своєї роботи різного роду корективи. На заключній нараді групи обговорюються успіхи і труднощі. Учасники вивчають звіти, видані їм на першому і другому етапах, і заслуховують виступ спостерігача за процесом. По суті, вони займаються пошуком шляхів удосконалення по частині виконання своїх ролей, з тим щоб проводити подальші оцінки з меншими зусиллями і з більш високою ефективністю. Дії, виконані в період оцінки учасниками трьох груп, ретельно реєструються. По закінченні кількох місяців керівник групи повинен зв'язатися із замовником оцінки, для того щоб оцінити довгострокові результати її проведення і порівняти витрати з вигодами.

Чотири етапи A TAM, їх учасники та приблизний графік представлені в табл. 15.1.

Таблиця 15.1 - Етапи ATAM та їх характеристики

Етап	Операції	Учасники	Середня тривалість
0	Встановлення партнерських відносин і підготовка	Керівництво групи оцінки та основні відповідальні за проект особи	Проходить у неформальній обстановці, згідно конкретної ситуації; може тривати кілька тижнів
1	Оцінка	Група оцінки і відповідальні за проект особи	1 день з наступною перервою тривалістю від 2 до 3 тижнів
2	Оцінка (продовження)	Група оцінки, відповідальні за проект особи та зацікавлені особи	2 дня
3	доопрацювання	Група оцінки і замовник оцінки	1 тиждень

15.3 Метод аналізу вартості та ефективності (кількісний підхід до прийняття архітектурно-проектних рішень)

Метод аналізу вартості та ефективності (Cost Benefit Analysis Method, CBAM) базується на методі ATAM та забезпечує моделювання витрат і вигод, пов'язаних з прийняттям архітектурно-проектних рішень, і сприяє їх оптимізації. Методом CBAM оцінюються технологічні та економічні фактори, а іакож самі архітектурні рішення.

15.3.1 Контекст прийняття рішення

Всі програмні архітектори та відповідальні особи прагнуть довести до максимуму різницю між вигодами, отриманими від системи, і вартістю реалізації її проектного рішення. Будучи логічним продовженням методу ATAM, CBAM грунтується на його артефактах. Контекст CBAM зображений на рис. 15.1.

Оскільки архітектурні стратегії обмежуються різноманітними технічними і а економічними факторами, стратегії, застосовувані програмними архітекторами і проектувальниками, повинні бути поставлені в залежність від комерційних завдань програмної системи. Прямий економічний фактор - це вартість реалізації системи. Технічними чинниками є характеристики системи -

іншими словами, атрибути якості. У атрибутів якості також є економічний аспект - вигоди, одержувані від їх реалізації.

Рисунок 15.1- Контекст методу аналізу вартості та ефективності (CBAM)

ATAM допомагає виявити ряд основних архітектурних рішень, значущих в контексті сформульованих зацікавленими особами сценаріїв атрибутів якості. Ці рішення призводять до реакції з боку атрибутів якості, точніше кажучи, окремих рівнів готовності, продуктивності, безпеки, практичності, модифікованості і т. д. З іншого боку, кожне архітектурне рішення пов'язане з певними витратами (вартістю). Наприклад, досягнення бажаного рівня готовності шляхом резервування апаратних засобів мають на увазі один вид витрат, а реєстрація в файлах на диску контрольних точок - інший. Ці архітектурні рішення призводять до (імовірно різним) вимірним рівням готовності, які мають певну цінність для компанії-розробника системи. Можливо, її керівництво вважає, що зацікавлені особи заплатять велику суму за систему з високою готовністю (якщо, приміром, це телефонний комутатор або програмне забезпечення для медичного спостереження), або боїться загрузнути в судових розглядах у разі відмови системи (цілком розумно, якщо мова йде про програму управління антиблокувальної гальмівною системою автомобіля).

ATAM виявляє архітектурні рішення, прийняті щодо розглянутої системи, і встановлює їх зв'язок з комерційними завданнями і кількісною мірою реакції атрибутів якості. Беручи ці дані на озброєння, CBAM допомагає виявити пов'язані з такими рішеннями витрати і вигоди. Ґрунтуючись на цій інформації, зацікавлені особи можуть прийняти остаточні рішення щодо резервування апаратної частини введення контрольних точок і всіх інших тактик, спрямованих на підвищення

готовності системи. Цілком можливо, що вони віддадуть перевагу сконцентрувати ресурси, які, як відомо, обмежені, на реалізацію якогось іншого атрибута якості - наприклад, на поліпшення співвідношення вигод і витрат за рахунок підвищення продуктивності. Через обмеженість бюджету розроблення та оновлення системи кожна архітектурне рішення за великим рахунком змагається за право на існування з усіма іншими.

Подібно фінансовому консультантові, який ніколи прямо не вкаже, у що вкладати гроші, CBAM не замінює собою рішень, прийнятих зацікавленими особами. Він лише допомагає їм встановити і документувати витрати і вигоди архітектурних інвестицій, усвідомити невизначеність цього «портфеля». На цій основі зацікавлені особи можуть приймати раціональні рішення, що задовольняють їх потреби і зводять до мінімуму ризики. Метод CBAM виходить з припущення про те, що архітектурні стратегії (як сукупність архітектурних тактик) впливають на атрибути якості системи, а ті, в свою чергу, надають зацікавленим особам деякі вигоди. Ці вигоди ми називаємо корисністю (utility). Будь-яка архітектурна стратегія відрізняється тієї чи іншої корисністю для зацікавлених осіб. З іншого боку, є витрати (вартість) і час, які необхідно витратити на реалізацію цієї стратегії. Відштовхуючись від цієї інформації, метод CBAM допомагає зацікавленим особам у процесі вибору архітектурних стратегій, характеризуються максимальним прибутком на інвестований капітал (return on investment, ROI), - іншими словами, найбільш вигідних з точки зору співвідношення вигод і витрат.

15.3.2 Реалізація CBAM

На рис. 15.2. зображена діаграма процесів, що становлять основу CBAM. Перші чотири етапи супроводжуються коментарями з вказівкою відносного числа розглянутих сценаріїв. Поступово їх число зменшується, таким чином, зацікавлені особи зосереджуються на тих сценаріях, які, на їх думку, в

контексті ROI здобудуть найбільше значення.

Етап 1 . Критичний аналіз сценаріїв . Відбір третини від загальної кількості сценаріїв шляхом розстановки пріоритетів	N сценаріїв
Етап 2 . Уточнення сценаріїв . Визначення рівнів реакції атрибутів якості для найкращого , найгіршого , поточного і бажаного варіантів сценаріїв	N / 3 сценаріїв
Етап 3 . Упорядкування сценаріїв згідно пріоритетам. Виняток половини сценаріїв	N / 3 сценаріїв
Етап 4 . Визначення корисності для поточного і бажаного рівнів кожного з сценаріїв	N / 6 сценаріїв
Етап 5 . Розробка для сценаріїв архітектурних стратегій і визначення рівнів реакції атрибутів якості	
Етап 6 . Визначення очікуваної корисності архітектурної стратегії шляхом інтерполяції	
Етап 7 . Визначення загальної вигоди, отриманої від архітектурної стратегії	
Етап 8 . Відбір архітектурних стратегій на основі ROI і з урахуванням обмежень за витратами	
Етап 9 . Наочне підтвердження отриманих результатів	

Рисунок 15.2 - Діаграма процесів CBAM

Етап 1. Критичний аналіз сценаріїв. Критичний аналіз сценаріїв проводиться в рамках ATAM; на цьому ж етапі зацікавлені особи можуть формулювати нові сценарії. Пріоритети розставляються відповідно з потенціалом сценаріїв у контексті виконання комерційних завдань системи; за результатами етапу для подальшого розгляду відбирається третину від загального початкового числа сценаріїв.

Етап 2. Уточнення сценаріїв. Уточненню піддаються сценарії, відібрані за результатами першого етапу; основна увага при цьому приділяється їх значень стимулу-реакції. Для кожного сценарію встановлюються найгірший, поточний, бажаний і найкращий рівні реакції атрибута якості.

Етап 3. Розстановка сценаріїв згідно пріоритетам. Кожній' зацікавленій особі виділяється 100 голосів, які він береться розподілити між сценаріями, виходячи з їх бажаних значень реакції. Після підрахунку голосів шія подальшого аналізу залишається тільки половина сценаріїв. Сценарієм з найвищим рангом присвоюється вага 1.0, і, відштовхуючись від нього, значення ваги встановлюються для всіх інших сценаріїв. Саме ці значення згодом задіються при обчисленні загальної вигоди стратегії. Крім іншого, на розглянутому етапі складається список атрибутів якості, які зацікавлені особи вважають значущими.

Етап 4. Встановлення корисності. Для сценаріїв, що залишилися після проведення етапу 3, визначаються значення корисності всіх рівнів реакції

(найгіршого, поточного, бажаного, найкращого) атрибута якості.

Етап 5. Розробка для сценаріїв архітектурних стратегій і встановлення їх бажаних рівнів реакції атрибута якості. Розробка (або фіксація розроблених) архітектурних стратегій, орієнтованих на реалізацію обраних сценаріїв, і визначення очікуваних рівнів реакції атрибута якості, враховуючи ту обставину, що одна архітектурна стратегія іноді надає дію на кілька сценаріїв, розрахунки необхідно провести для кожного з порушених сценаріїв.

Етап 6. Визначення корисності очікуваних реактивних рівнів атрибута якості шляхом інтерполяції. Виходячи із встановлених значень корисності (відображених на кривій корисності) для розглянутої архітектурної стратегії визначається корисність бажаного рівня реакції атрибута якості. Ця операція проводиться для кожного з перерахованих на етапі 3 значущих атрибутів якості.

Етап 7. Розрахунок загальної вигоди, отриманої від архітектурної стратегії. Значення корисності «поточного» рівня віднімається з бажаного рівня і нормалізується виходячи з поданих на третьому етапі голосів. Підсумовуються вигоди, отримані від конкретної архітектурної стратегії, по всіх сценаріях і для всіх значущих атрибутів якості.

Етап 8. Відбір архітектурних стратегій з урахуванням ПО/, а також обмежень по вартості і часу. Для кожної архітектурної стратегії визначаються вартісні і тимчасові чинники. Значення ЯОІ для стратегій визначається як відношення вигоди до витрат. Архітектурні стратегії упорядковуються за рангом згідно значень ЇЮІ; згодом бюджет в першу чергу витрачається на вищі за рангом стратеги.

Етап 9. Інтуїтивне підтвердження результатів. Перевіряється відповідність обраних архітектурних стратегій комерційним завданням компанії. Якщо спостерігаються протиріччя, шукаємо недогляди під час проведення аналізу. У разі якщо протиріччя значні, перераховані етапи проводяться повторно.

15.4 Контрольні питання

1. Які методи аналізу компромісних архітектурних рішень?
2. Основні етапи методу аналізу компромісних архітектурних рішень?
3. Що являє собою метод аналізу вартості та ефективності?

ЛІТЕРАТУРА

1. Брауде Э. Технология разработки програмного обеспечения. - СПб.іПитер, 2004. - 655 с.

2. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. - М.: Финансы и статистика, 2000.

3. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя, 2001.

4. Якобсон А., Буч Г., Рамбо Дж - Унифицированный процесс разработки программного обеспечения, 2002

5. Салливан Э. - Время-деньги. Создание команды разработчиков программного обеспечения, 2002.

6. Зелькович М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. - М.: Мир, 1982 - 386.

7. Макконнелл С. Профессиональная разработка программного обеспечения. Санкт-Петербург, 2007.

8. Константайн, Л. Разработка программного обеспечения / - СПб. : Питер, 2004. - 592 с.

9. Соммервилл И. Инженерия программного обеспечения. - М., 2002. 618 с

10. Орлик С. Программная инженерия. Проектирование программного обеспечения. Software Design. - М., 2005. 12 с

11. Тамре Л. - Введение в тестирование программного обеспечения. М: Вильямс, 2003 - 368с.

12. Технология разработки программного обеспечения (Гагарина Л.Г. и др.). М:Инфра- М, 2008-400с.

НАВЧАЛЬНЕ ВИДАННЯ

Конспект лекцій з дисципліни „Архітектура та проектування ПЗ” для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Інженерія програмного забезпечення» із спеціальності 121 – «Інженерія програмного забезпечення»

Укладачі:
доц., к.т.н. Завгородній В.В.
доц., к.т.н. Ялова К.М.

Підписано до друку_____
Формат_*А4*_ Обсяг_ др. арк.
Наклад ____ прим. Замовлення ____.
51918, м. Кам'янське
вул. Дніпробудівська, 2.