МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

**М.І. Цюцюра**

# АРХІТЕКТУРА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій
для студентів спеціальності
121 "Інженерія програмного забезпечення",

Київ 2020

# ЛЕКЦІЯ 1-1. Архітектура програмного забезпечення

1. Поняття архітектури ПЗ. Цілі архітектури. Принципи проектування архітектури.
2. Проектування архітектури. Компоненти. Складання плану реалізації моделі предметної області програмного забезпечення. Варіанти складання моделей проектів.
3. Класифікація архітектури.
4. Типи архітектури і їх моделі. Архітектурні шаблони і стилі. Поєднання архітектурних стилів. Компонентна архітектура.
5. Багатошарова архітектура. N-рівнева/3-рівнева архітектура. Сервісно-орієнтована архітектура.
6. Розробка і оцінка архітектури на основі сценаріїв.
7. Статичні та динамічні діаграми при проектуванні архітектури ПЗ.

*(жовтим маркером на самостійну роботу)*

## 1. Поняття архітектури ПЗ. Цілі архітектури. Принципи проектування архітектури.

"Архітектура" і "Інженерія", як види людської діяльності, існували задовго до появи комп'ютерних технологій. Насамперед, ці види діяльності пов'язував процес створення проекту - прототипу, прообразу передбачуваного або можливого об'єкта. Іншими словами проектування містить у своєму складі поняття "архітектура" і "Інженерія", а проектування програмного забезпечення деяким відрізняється в цьому сенсі від проектування, наприклад, будівель і споруд. Тенденції розвитку будівельної архітектури останніх *десятиліть пов'язані з максимальною функціональністю проектованих об'єктів*. Архітектурне проектування ПЗ також переслідує аналогічну мету.

Згідно енциклопедії «Вікіпедія», ***архітектура програмного забезпечення***– це представлення системи програмного забезпечення, що дає інформацію про компоненти складають систему, про взаємозв'язки між цими компонентами і правилах, що регламентують ці взаємозв'язки, яке призначене для ефективної розробки проекту такої системи.

***Проектування програмного забезпечення***, у свою чергу, передбачає вироблення властивостей системи на основі аналізу постановки задачі (моделей предметної області (Domain Design) і вимог до ПЗ), а також досвіду проектувальника.

Автори книги "*Порождающее программирование: методы, инструменты, применение*" К. Чарнецький і У. Айзенекер [1] визначають ***проектування архітектури ПЗ*** як "високорівневе проектування, метою якого є створення гнучкої структури, що задовольняє всім основним вимогам і передбачає деяку ступінь свободи реалізації.

Як правило, з тих деталей, які менш інших схильні до змін, формується «скелет». При цьому всі інші деталі робляться якомога більш гнучкими, з тим, щоб

згодом їх можна було без праці оновити. Втім, зміни іноді вносяться навіть у скелет".

*Слайд 6.*

***Архітектура*** **-** це базова організація системи, втілена в її компонентах, їхніх відносинах між собою і з оточенням, а також принципи, що визначають проектування та розвиток системи.

***Архітектура*** - це набір значимих рішень з приводу організації системи програмного забезпечення, набір структурних елементів і їх інтерфейсів, за допомогою яких компонується система, разом з їх поведінкою, обумовленим у взаємодії між цими елементами, компонування елементів в поступово укрупняющися підсистеми, а також стиль архітектури який направляє цю організацію - елементи та їхні інтерфейси, взаємодії та компоновку.

***Архітектура програми*** або комп'ютерної системи - це структура або структури системи, які включають елементи програми, видимі ззовні властивості цих елементів і зв'язки між ними.

*Слайд 7.*

***Архітектура ПЗ – це артефакт***, що представляє собою результат процесу розробки програмного забезпечення. Елементи архітектури ПЗ і моделі їх з'єднання призначені для задоволення вимог до проектованих системам. У проекті архітектури ПЗ повинні бути враховані функціональні та нефункціональні вимоги до ефективності, витривалості, розширюваності, відмовостійкості, продуктивності, можливості повторного використання, а також адаптування розробляється ПО.

***Архітектурний проект ПЗ***, дозволяє оперативно визначити, наскільки даний програмний продукт відповідає пропонованим до нього вимогам.

*Слайд 8.*

***Метою архітектурного проектування*** предметної області є наступні артефакти:

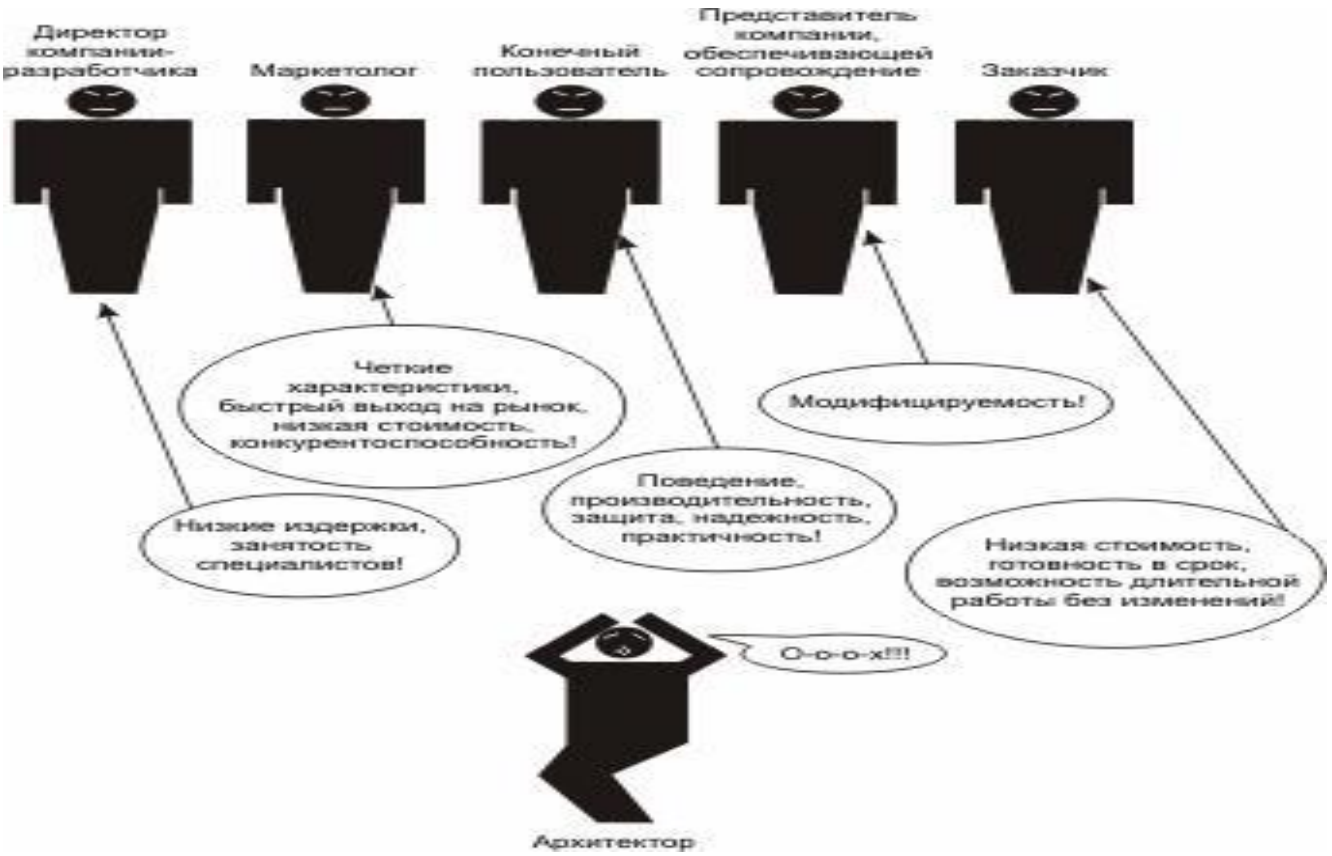
1. розробка архітектури множини (сімейства) систем, які входять до даної предметної області;
3. складання плану реалізації моделі предметної області;
4. реалізація моделі предметної області.

*Слайд 9.*

## 2. Проектування архітектури. Компоненти. Складання плану реалізації моделі предметної області програмного забезпечення. Варіанти складання моделей проектів.

### *2.1.* ***Проектування архітектури***

Вплив на архітектуру ПЗ зацікавлених в системі осіб (на рис)




*Слайд 10.*

***Вплив на архітектуру надає компанія-розробник.*** Компанії роблять прямі інвестиції в різні активи - зокрема в існуючі варіанти архітектури і засновані на них продукти. Кожен наступний проект у такому випадку мислиться як продовження ряду подібних систем, а в його кошторисі закладено активне повторне використання наявних коштів.

Дотримуючись своїм стратегічним завданням, компанії іноді роблять довгострокові інвестиції в інфраструктуру; в такому випадку передбачувана система мислиться як один із засобів фінансування та розширення цієї інфраструктури.

Певний вплив на програмну архітектуру надає організаційна структура компанії-розробника.

*Слайд 11.*

***Вплив на архітектуру досвіду і навичок архітекторів.***

Позитивний досвід - повторення його в наступних роботах.

Негативний досвід - відмова використовувати його в наступних роботах.

Архітектори люблять експериментувати з новими зразками (pattern) і методиками.

*Слайд 12.*

***Вплив на архітектуру технічної бази***

Підготовка та досвід архітектора проявляється, зокрема, в його роботі з технічною базою (technical environment).

До технічній базі можна віднести:

- методи роботи, прийняті в даній галузі
- прийоми програмної інженерії, поширені в професійному співтоваристві, в яке входить архітектор.

*Слайд 13.*

**Варіативність факторів врливу на архитектуру**

*Слайд 14.*

До складу архітектурного проекту ПЗ входять:

- опис елементів, з яких складається дана система;
- схеми взаємодій між цими елементами;
- документація зразків (patterns), на основі яких здійснюється їх компоновка;
- список і зміст обмежень (вимог), характерних для цих зразків.

У будівництві мовою опису проекту є архітектурно-будівельні креслення і об'ємні моделі, а також текстові описи об'єктів, що зводяться і технологій їх зведення.

*Слайд 15.*

Ілюстративними засобами вираження характеристик ПЗ, в архітектурному проекті використовуються різні нотації:

- блок-схеми (схеми алгоритмів);
- ER-діаграми, UML-діаграми;
- DFD-діаграми;
- макети.

Кожна підсистема ПЗ, що складається із сукупності її ***компонентів*** і взаємодій між ними, повинна бути детально описана у відповідній частині проект з використанням цих нотацій. Оскільки така підсистема може виступати в якості складового елементу більш масштабної системи, в архітектурному проекті ПЗ обов'язково міститься докладний опис укрупнених частин системи за допомогою цих же засобів опису проекту.

*Слайд 16.*

Щодо вжитого терміна ***"компонент"***, Питер Илес (старший разработчик архитектуры информационных технологий, IBM) в статье "Что такое архитектура программного обеспечения?" [2] пише, що "... велика частина визначень архітектури не визначає терміну" компонент ", і стандарт IEEE 1471 – не виняток, оскільки навмисно залишає це поняття невизначеним, щоб воно відповідало певній множині тлумачень, можливих для конкретної галузі.

***Компонент*** - абстрактна одиниця інструкцій програмного забезпечення і внутрішніх станів, яка забезпечує трансформацію даних через свої інтерфейси.

***Компонент*** може бути:

- логічним або фізичним,
- технологічнонезалежним або технологічно-зв'язаним;
- крупно-або дрібногранульованим ...

***Проектування*** взагалі, а також проектування ПЗ, ***є прикладним видом діяльності***. Оскільки в будь-якому з варіантів, проектування - це мистецтво створення того, чого немає в природі, архітектор (проектувальник) ПЗ повинен оволодіти мистецтвом проектування та самовираження, що дозволяє учасникам і замовникам проекту "будувати" необхідне ПЗ і управляти цим "будівництвом" і експлуатацією подальшою еволюцією системи.

**На лабораторну роботу №1**

### *2.2. Складання плану реалізації моделі предметної області програмного забезпечення*

Реалізація моделі предметної області являє собою архітектурне моделювання проектованого об'єкта. План реалізації, складений архітектором ПЗ, регламентує способи отримання конкретних систем із загальної архітектури і компонентів. *Архітектурна частина проекту складання* ***моделі*** *предметної області* містить описи:

- інтерфейсу замовника для запуску конкретних підсистем;
- процесів складання компонентів;
- обробки запитів на зміни і розробку;
- вимірювань, супроводу та оптимізації бізнес-процесів.

Дана частина проекту описує збірку розроблення об'єкта з імовірним використанням автоматизованих засобів для *складання моделі*. Рівень автоматизації збирання залежить від безлічі факторів і пов'язаний як з програмно-технічною оснащеністю проекту, так і з наявністю достатнього рівня застосовуваних *артефактів* предметної області (у тому числі із застосуванням повторного коду).

Загалом можливі наступні ***варіанти складання моделей проектів***:

1) ***Збірка додатків з компонентів*** проводиться вручну.

   До складу проекту також входять:
   - *опису архітектури і компонентів;*
   - *реалізації предметно-орієнтованих мов;*
   - *керівництва по розміщенню графічних користувацьких інтерфейсів.*

2) Для ***складання компонентів*** застосовуються різноманітні інструментальні засоби:
   - засоби пошуку та перегляду компонентів,
   - описи застосування генераторів для автоматизації певних аспектів розробки додатків.

   До складу проекту також входять:

- *опису архітектури і компонентів,*
- *реалізації предметно-орієнтованих мов,*
- *реалізації керівництва по розміщенню графічних користувацьких інтерфейсів,*
- *генераторів і інфраструктури, за допомогою якої проводиться пошук,*
- *класифікація, поширення компонентів. Документація формується автоматично;*

3) ***Автоматичне складання моделі*** об'єкта із застосуванням інструментальних засобів для замовника (коштів породжує програмування), за допомогою яких формується запит на необхідне ПЗ.

Виробництво програми може бути досягнуто одним запитом, якщо в ньому не втримується частин, створюваних традиційними методами розробки.

До складу проекту також входять:

- *опису архітектури і компонентів;*
- *реалізації предметно-орієнтованих мов;*
- *керівництва по розміщенню графічних користувацьких інтерфейсів; генераторів і інфраструктури, за допомогою якої проводиться пошук;*
- *класифікація, поширення компонентів і тому подібні операції;*
- *організації процесів виробництва додатків.*

Вся документація також формується автоматично.

### *2.3. Реалізація моделі предметної області ПЗ*

На даному етапі проводиться реалізація архітектури,компонентів і плану реалізації за допомогою методик, що містяться впроекті.

*Слайд 17*

## 3. Класифікація архітектури.

**Вибір архітектури визначає спосіб реалізації вимог** на високому рівні абстракції. Саме архітектура майже повністю визначає такі характеристики ПЗ як надійність, переносимість і зручність супроводу.

**Архітектура значно впливає і на зручність використання і ефективність ПЗ**, які визначаються також і реалізацією окремих компонентів. Значно менше вплив архітектури на функціональність – зазвичай для реалізації заданої функціональності можна використовувати *різні архітектури*.

Тому вибір між тією або іншою архітектурою визначається насамперед саме нефункціональними вимогами і необхідними властивостями ПЗ в аспектах зручності супроводу та переносимості.

При цьому **для побудови гарної архітектури** треба *враховувати можливі протиріччя між вимогами до різних характеристик і вміти вибирати компромісні рішення*, що дають прийнятні значення за всіма показниками.

*Слайд 18*

*По-перше*, для підвищення ефективності в загальному випадку вигідніше використовувати монолітні архітектури, в яких виділено невелике число компонентів (або 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, зазвичай, тільки в рамках одного компонента.

*Слайд 19*

*З другого боку*, для підвищення зручності супроводу, навпаки, краще розбивати систему на велике число окремих компонентів, з тим, щоб кожен з них вирішував свою невелику, але чітко визначену частину загальної задачі. При цьому, якщо виникають зміни у вимогах або проекті, їх зазвичай можна звести до зміни однієї-кількох таких підзадач, і, відповідно, змінювати тільки ті компоненти, що відповідають за вирішення цих підзадач.

*Слайд 20*

*З третього боку*, для підвищення надійності краще використовувати дублювання функцій, тобто зробити кілька компонентів відповідальними за вирішення однієї підзадачі. Причому, оскільки помилки в ПЗ найчастіше носять невипадковий характер (тобто вони повторювані, на відміну від апаратного забезпечення, де помилки пов'язані насамперед з випадковими змінами характеристик середовища і можуть бути подолані простим дублюванням компонентів, без зміни їх внутрішньої реалізації), краще використовувати різноматітні способи вирішення однієї і тієї ж задачі в різних компонентах.

*Слайд 21*

Список стандартів, що регламентують опис архітектури та проектну документацію взагалі, виглядає так:

- IEEE 1016-1998 Recommended Practice for Software Design Descriptions;
- IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.

*Слайд 22*

## 4. Типи архітектури і їх моделі. Зразки проектування. Архітектурні шаблони і стилі. Поєднання архітектурних стилів. Компонентна архітектура.

### *4.2. Зразки проектування та архітектурні стилі*

На основі наявного досвіду дослідниками і практиками розробки ПЗ вироблено деякий безліч типових архітектур, знайомство з якими дозволяє не винаходити велосипед для вирішення досить відомих завдань. Подібні типові рішення на рівні архітектури називаються архітектурними стилями. Точніше, архітектурний стиль визначає набір типів компонентів системи і набір шаблонів їхнього взаємодій з передачі даних або управління. Різні архітектурні стилі підходять для вирішення різних завдань в плані забезпечення нефункціональних вимог, хоча одну і ту ж функціональність можна реалізувати, використовуючи різні стилі.

*Слайд 23*

***Архітектурні стилі*** є зразками проектування на рівні архітектури. Зразок проектування (design pattern) –***це шаблон рішення*** часто зустрічається завдання проектування, який можна використовувати всякий раз, коли ця задача виникає. ***Зразки проектування*** поділяються залежно від масштабу рішень на архітектурні, що визначають можливу декомпозицію системи в цілому або великих підсистем, області відповідальності підсистем і правила їх взаємодії, проектні, що визначають шаблон взаємодій групи компонентів, зазвичай в рамках деякої підсистеми, для вирішення деякої загальної задачі проектування в повторюваному контексті, і ідіоми, що визначають спосіб використання мовних конструкцій для вирішення подібних завдань.

Приклади архітектурних стилів приведені в наступній таблиці.

*Слайд 24-25*

***Таблиця 1.1*** Стиль або зразок. Контекст використання. Приклади

| **Стиль або зразок** | **Контекст використання** | **Приклади** |
|---|---|---|
| – Конвеєр обробки даних (data flow) | Система видає добре певні вихідні дані в результаті обробки добре певних вхідних, при цьому процес обробки не залежить від часу, застосовується багаторазово, однаково до будь даними на вході. Важливою властивістю є чітко визначена структура даних та підтримка можливості інтеграції з іншими системами | |
| – Пакетна обробка | Один висновок виробляється на основі читання деякого набору даних на вході, проміжні перетворення послідовні | Виконання тестів |
| – • Канали і фільтри | Потрібно забезпечити перетворення безперервних потоків даних, перетворення Інкрементальний, наступне може бути розпочато до закінчення попереднього, можливе додавання додаткових перетворень | Утиліти UNIX, компілятори |
| – Замкнутий цикл управління | Потрібно забезпечити постійне управління в умовах погано передбачуваних впливів оточення, особливо, якщо система повинна реагувати на зовнішні фізичні | Системи управління рухом |

| | | |
|---|---|---|
| | події | |
| – Виклик-повернення (call-return) | Порядок виконання дій досить визначений, компонентам немає чого витрачати час на очікування звернення від інших | |
| – Процедурна декомпозиція | Дані незмінні, процедури роботи з ними можуть трохи змінюватися, можуть виникати нові | Основна схема побудови програм для мов C, Pascal, Ada |
| – Абстракние типи даних | Важливі можливості внесення змін та інтеграції з іншими системами, в системі багато даних, структура яких може мінятися | Бібліотеки компонентів |
| – Багаторівнева система | Важливі переносимість і можливість багаторазового використання, є природне розшарування системи на специфічні тільки для неї функції та функції загального характеру, специфічні для платформи | Протоколи (модель OSI і реальні) |
| – Незалежні компоненти | Можливо розпаралелювання роботи і ізпользованія декількох машин, система природно розбивається на слабо пов'язані невеликі компоненти, робота яких може бути організована майже незалежно | |
| – Клієнт-сервер | Завдання, які вирішуються природно розподіляються між ініціаторами та обработчиками запитів, можлива зміна зовнішнього представлення даних і способів їх обробки | Основна модель бізнес-додатків |
| – Розподілені об'єкти | Можливість використання розподіленої архітектури і численні дані з мінливою структурою | |
| – Інтерактивні системи | Необхідність досить швидко реагувати на дії користувача, мінливість користувальницького інтерфейсу | |
| – Дані-подання-обробник | Зміни в зовнішньому поданні досить вірогідні, одна і та ж інформація представляється по-різному в декількох місцях, система повинна швидко реагувати на зміни даних | Document-View в MFC (Microsoft Foundation Classes) |
| – Представлення-абстракція-управління | Інтерактивна система на основі агентів, що мають власні стани і користувальницький інтерфейс, можливе додавання нових агентів | |
| – Системи на основі сховища даних | Основні функції системи пов'язані зі зберіганням, обробкою і представленням великих кількостей даних | |
| – Репозиторій | Порядок роботи визначається потоком зовнішніх подій і цілком визначений | Средовище розробки і CASE-системи |
| – Дошка оголошень | Спосіб вирішення завдання в цілому невідомий або занадто трудомісткий, але відомі методи, частково вирішуючи задачу, композиція яких здатна видавати прийнятні результати, можливо додавання нових споживачів даних або обробників | Системи розпізнавання тексту |

## 5. Багатошарова архітектура. N-рівнева/3-рівнева архітектура. Сервісно-орієнтована архітектура.

## 6. Розробка і оцінка архітектури на основі сценаріїв.

UML. Види діаграм UML

Для представлення архітектури (точніше різних вхідних в неї структур) зручно використовувати графічні мови. На даний момент найбільш опрацьованим і найбільш широко використовуваним з них є уніфікована мова моделювання (Unified Modeling Language, UML), хоча на високому рівні абстракції архітектуру системи зазвичай описують просто набором іменованих прямокутників, з'єднаних лініями і стрілками, що представляють можливі зв'язки.

Деякі такі мови закріплені у вигляді стандартів, наприклад

• IEEE 1320.1-1998 (R2004) Standard for Functional Modeling Language - Syntax and Semantics for IDEF0

Цей стандарт описує мову ієрархічних діаграм потоків даних.

• IEEE 1320.2-1998 (R2004) Standard for Conceptual Modeling Language - Syntax and Semantics for IDEF1X97 (IDEFobject)

Цей стандарт описує мова опису структурованих даних на основі понять сутності, зв'язку та атрибуту суті, використовуваний, зокрема, для моделювання баз даних.

UML пропонує використовувати для опису архітектури 8 видів діаграм. Не всяка діаграма на UML описує архітектуру - 9-й вид діаграм, діаграми варіантів використання, не відносяться до архітектурних уявленнями, крім того, і інші види діаграм можна використовувати для опису внутрішньої структури компонентів або сценаріїв дій користувачів і інших елементів, до архітектури не відносяться .

• Статичні структури, які відображують постійно присутні в системі сутності й зв'язки між ними, або сумарну інформацію про сутності і зв'язках, якої сутності і зв'язки, що існують в якийсь момент часу

о Діаграми класів. Показують типи сутностей системи, атрибути типів (поля та операції) і можливі зв'язки між ними, а також відносини типів між собою за спадкоємства.

Найбільш часто використовуваний вид діаграм.

о Діаграми об'єктів. Показують об'єкти системи та їх зв'язку, в деякому конкретному стані або сумарно.

Використовуються рідко.

о Діаграми компонентів. Це компоненти у вузькому сенсі, компоненти «фізичного» уявлення системи - файли з вихідним кодом, динамічно підгружаємі бібліотеки, HTML-сторінки і ін. Вони визначають розбиття системи на набір сутностей, расматриваться як атомарні з погляду її збірки і конфігураційного управління.

Використовуються рідко.

о Діаграми розгортання. Показують прив'язку (у деякий момент часу або постійну) компонентів системи (під вузькому сенсі) до фізичних пристроїв - машинам, процесорам, принтерам, маршрутизаторів та ін.

Використовуються рідко.

• Динамічні структури, що описують відбуваються в системі процеси

о Діаграми діяльностей. Показують набір процесів-діяльностей і потоки даних, що передаються між ними, а також можливі їх синхронізації один з одним.

Використовуються досить часто.

о Діаграми сценаріїв. Показують можливі сценарії обміну повідомленнями / викликами в часі між різними компонентами системи (у широкому сенсі). Ці діаграми є підмножиною іншого графічного мови - мови діграмм послідовностей повідомлень (Message Sequence Charts, MSC).

Використовуються майже так само часто, як діаграми класів.

о Діаграми взаємодії. Показують ту ж інформацію, що і діаграми сценаріїв, але прив'язують обмін повідомленнями / викликами ні до часу, а до зв'язками між компонентами.

Використовуються рідко.

о Діаграми станів. Показують можливі стану окремих компонентів або системи в цілому, переходи між ними у відповідь на будь-які події і виконувані при цьому дії.

Використовуються досить часто.

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

1) Вибирається набір «основних» сценаріїв використання – найбільш істотних і часто використовуваних.

2) Визначаються, виходячи з досвіду проектувальників, обраного архітектурного стилю і вимог до переносимості та гнучкості, компоненти відповідають за певні дії – рішення певних підзадач – в рамках цих сценаріїв.

3) Сценарії розбиваються на послідовності обміну повідомленнями між отриманим компонентами.

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

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

6) Розглядаються «неосновні» сценарії, які так само розбиваються на послідовності обміну повідомленнями з використанням, по можливості, уже визначених інтерфейсів.

7) Якщо інтерфейси недостатні – вони розширюються.

8) Якщо інтерфейс компонента дуже великий або компонент відповідає за занадто багато чого - він розбивається на більш дрібні.

9) Там, де це необхідно в силу вимог ефективності або зручності супроводу, кілька компонентів можуть бути об'єднані в один.

10) Все це робиться до тих пір, поки не виконаються наступні умови:

*a) Всі сценарії використання реалізуються у вигляді послідовностей обміну повідомленнями між компонентами в рамках їх інтерфейсів.*

*b) Набір компонентів достатній для забезпечення всієї потрібної функціональності, досить зручний для супроводу і з точки зору переносимості і не викликає помітних проблем з ефективністю.*

*c) Кожен компонент має невеликий, чітко визначене коло вирішуваних завдань і чітко визначений, збалансований за розміром інтерфейсів.*

На основі можливих сценаріїв використання або модифікації системи можливий також аналіз характеристик архітектури та оцінка її придатності або

порівняльний аналіз декількох архітектур. Це так званий **метод аналізу архітектури ПЗ (Software Architecture Analysis Method, SAAM)**.

Основні його кроки наступні.

1. Визначити набір сценаріїв дій користувачів або зовнішніх систем, або сценаріїв використання деяких можливостей, які можуть вже матися на вістеме або бути новими. Сценарії повинні бути значущими для конкретних зацікавлених осіб, будь то користувач, розробник, відповідальний за супровід, представник контролюючої організації тощо. Чим повніше набір сценаріїв, тим вищою буде якість аналізу. Можна оцінити частоту появи, важливість сценаріїв.

2. Визначити архітектуру (або декілька порівнюваних архітектур). Це має бути зроблено в зрозумілій всім учасникам оцінки формі.

3. Класифікувати сценарії. Для кожного сценарію з набору повинно бути визначено, чи підтримується він даної архітектурою або потрібно вносити до неї зміни, щоб цей сценарій став виконаємо. 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 до їх інтерфейсах. Підтримка сценарію означає, що особа, зацікавлена в його виконання оцінює ступінь підтримки як достатню, а необхідні при цьому дії як досить зручні.

4. Оцінити сценарії. 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.

5. Виявити взаємодію сценаріїв. Визначити які компоненти потрібно змінювати для непідтримуваних сценаріїв, компоненти, які потрібно змінювати для підтримки декількох сценаріїв - такі сценарії називають взаємодіючими, оцінити ступінь смислової пов'язаності взаємодіючих сценаріїв.

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 декомпозувати.

Компоненти, в яких взаємодіють багато сценаріїв також є можливими проблемними місцями.

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

## 7. Статичні та динамічні діаграми при проектуванні архітектури ПЗ.

Приклади діаграм UML для простої системи ведення банківських рахунків.

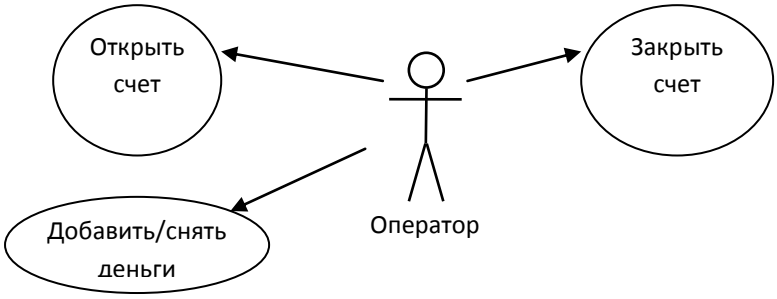



**Рисунок 1. Діаграма варіантів використання (не архітектурна).**

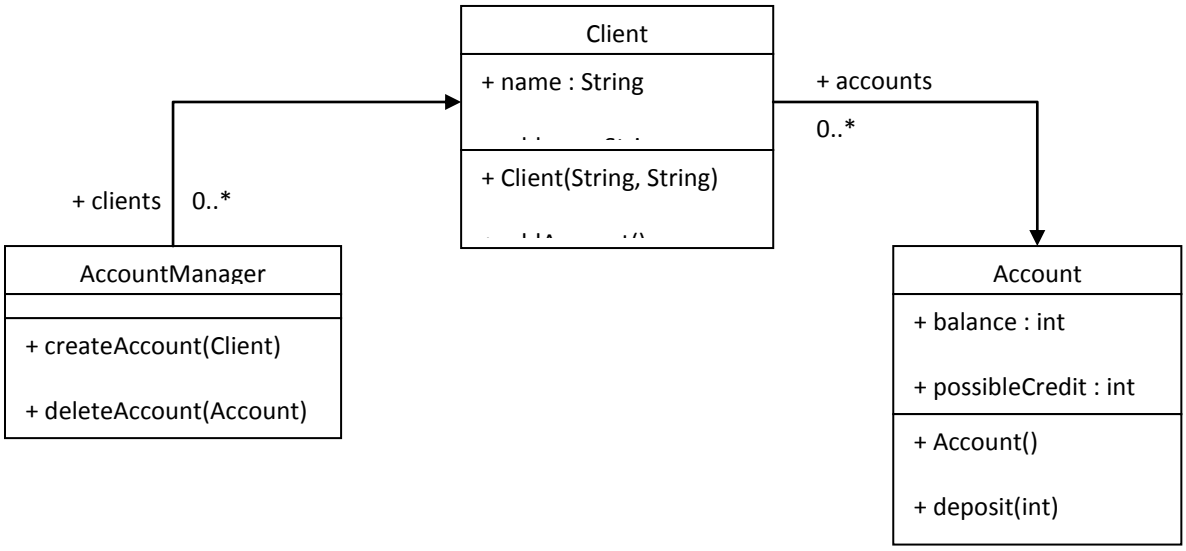



**Рисунок 2. Діаграма класів**

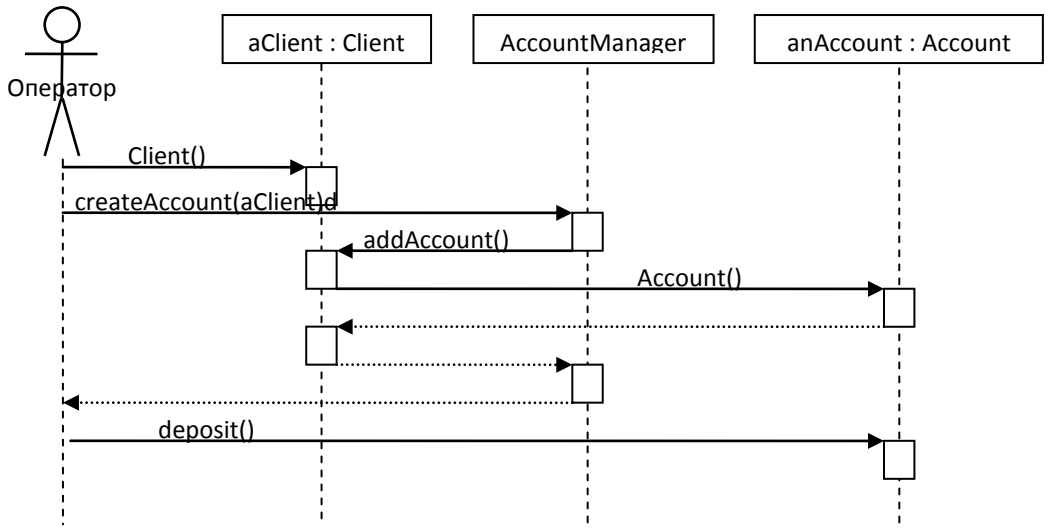



**Рисунок 3. Діаграма сценаріювідткриття рахунку**

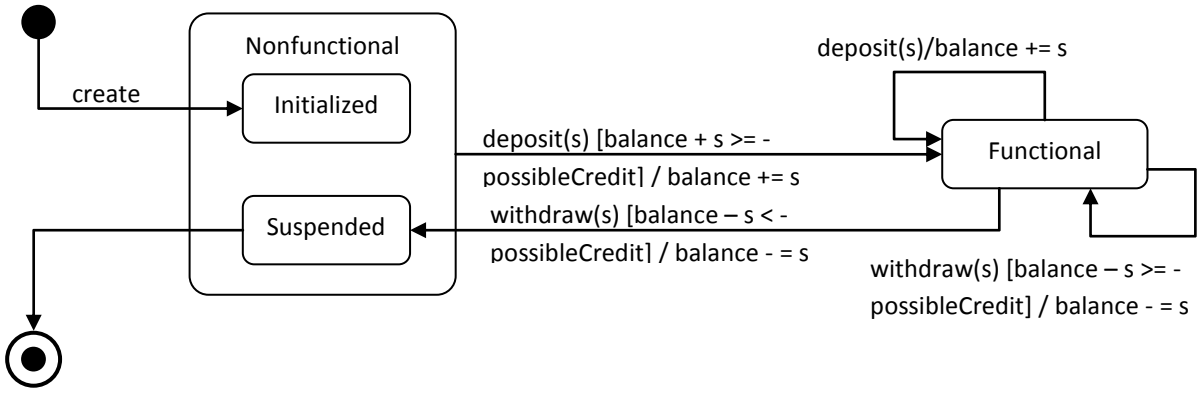



**Рисунок 4. Діаграма станів рахунку**

## Запитання

1) Що таке архітектура ПО?
2) Які артефакти є метою проектування архітектури ПЗ "?
3) Яка роль архітектора при створенні ПЗ?
4) У якому порядку (черговості) виконуються процеси проектування ПО: проектування архітектури систем предметної області, складання плану реалізації моделі та реалізація моделі?
5) Назвіть склад архітектурної частини проекту розробки програмного забезпечення.
6) Яка роль архітектора при створенні ПЗ?

## Питання для самостійної роботи

1. Принципи проектування архітектури.
2. Типи архітектури і їх моделі.
3. Багатошарова архітектура. N-рівнева/3-рівнева архітектура. Сервісно-орієнтована архітектура.

## *Література по темі «Архітектура ПО»*


1. Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение. СПб.: Питер. 2005 .
2. Илес П. Что такое архитектура программного обеспечения? http://www.interface.ru/home.asp?artId=2116
3. Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы, СПб.: Символ-Плюс, 2001.
4. И. Соммервилл. Инженерия программного обеспечения. Вильямс, 2002.
5. Э. Дж. Брауде. Технология разработки программного обеспечения. Питер, 2004.
6. М. Фаулер и др. Архитектура корпоративных программных приложений. Вильямс, 2004.
7. М. Фаулер, К. Скотт. UML в кратком изложении. М., Мир, 1999.
8. Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML.Руководство пользователя. М., ДМК, 2000.
9. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Питер-ДМК, 2001.
10. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. Wiley, 2002.
11. L. Bass, P. Clements, R. Kazman. Software Architecture in Practice. 2-nd edition, Addison-Wesley, 2003.

**Лекція 2. Архітектура ПЗ. Технології програмування. компонентний підхід. Зразки проектування.**

1) Архітектура ПЗ
2) Розробка і оцінка архітектури
3) UML. Види діаграм UML
4) Зразки проектування і архітектурні стилі

## 1. Архітектура ПЗ

Під архітектурою ПЗ зазвичай розуміють набір внутрішніх структур ПЗ, що складаються з компонентів, зв'язків і можливих взаємодій між ними, а також видимих ззовні властивостей цих компонентів.

Під компонентом в цьому визначенні мається на увазі досить довільний структурний елемент ПО, який можна виділити шляхом визначення інтерфейсу взаємодії між цим компонентом і всім, що його оточує. Термін «компонент» в розробці ПЗ найчастіше (далі, під час обговорення UML і технології EJB) має дещо інший, більш вузький зміст - це одиниця складання системи, її розгортання і конфігураційного управління, то, що не може бути розділене на більш дрібні елементи при розгортанні або постачання системи. Там, де можливі непорозуміння, буде вказано, в першому, широкому або в другому, вузькому сенсі вживається цей термін.

Архітектура ПЗ схожа на набір карт певної території - карти мають різні масштаби, на них показані різні елементи (адміністративно-політичний поділ, рельєф і тип місцевості - ліс, степ, пустеля, болотиста місцевість і ін., Економічна діяльність і зв'язку), але вони об'єднуються тим, що всі відомості, представлені на них, співвідносяться з географічним положенням.

Точно так же архітектура ПЗ являє собою набір структур або уявлень, що мають різні рівні абстракції (аналог масштабу географічних карт) і показують різні аспекти (структуру класів ПО, структуру розгортання, тобто прив'язки компонентів ПО до фізичних машинам, можливі сценарії взаємодій компонентів і пр.), що об'єднуються прив'язкою всіх представлених даних до структурних елементів ПЗ.

Архітектура важлива перш за все тому, що саме вона визначає більшість характеристик якості ПЗ в цілому. Архітектура служить також основним засобом спілкування між розробниками, а також і між усіма особами, зацікавленими в даному ПЗ.

Вибір архітектури визначає спосіб реалізації вимог на високому рівні абстракції. Саме архітектура майже повністю визначає такі характеристики ПЗ як надійність, переносимість і зручність супроводу. Архітектура значно впливає і на зручність використання і ефективність ПЗ, які визначаються також і реалізацією окремих компонентів. Значно менше вплив архітектури на функціональність - зазвичай для реалізації заданої функціональності можна використовувати різні архітектури.

Тому вибір між тією або іншою архітектурою визначається перш за все саме нефункціональними вимогами і необхідними властивостями ПЗ в аспектах зручності супроводу і 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, зазвичай, тільки в рамках одного компонента.

З іншого боку, для підвищення зручності супроводу, навпаки, краще розбивати систему на велике число окремих компонентів, з тим, щоб кожен з них вирішував свою невелику, але чітко визначену частину спільного завдання. При цьому, якщо виникають зміни у вимогах або проект, їх зазвичай можна звести до зміни однієї-кількох таких підзадач, і, відповідно, змінювати тільки відповідають за вирішення цих підзадач компоненти.

З третього боку, для підвищення надійності краще використовувати дублювання функцій, тобто зробити кілька компонентів відповідальними за вирішення однієї підзадачі. Причому, оскільки помилки в ПЗ найчастіше носять невипадковий характер (тобто вони повторювані, на відміну від апаратного забезпечення, де помилки пов'язані перш за все з випадковими змінами характеристик середовища і можуть бути подолані простим дублюванням компонентів, без зміни їх внутрішньої реалізації), краще використовувати досить сильно розрізняються способи вирішення однієї і тієї ж задачі в різних компонентах.

Список стандартів, що регламентують опис архітектури та проектну документацію взагалі, виглядає так:

- IEEE 1016-1998 Recommended Practice for Software Design Descriptions
- IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.

## 2. Розробка і оцінка архітектури

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

1) Вибирається набір «основних» сценаріїв використання - найбільш істотних і часто використовуваних
2) Визначаються, виходячи з досвіду проектувальників, обраного архітектурного стилю (див. Далі) і вимог до переносимості та гнучкості, компоненти відповідають за певні дії - рішення певних підзадач - в рамках цих сценаріїв:
   - Сценарії розбиваються на послідовності обміну повідомленнями між отриманим компонентами
   - При виникненні додаткових добре виділених підзадач, додаються нові компоненти, і сценарії уточнюються
   - 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 занадто багато - він розбивається на більш дрібні
- Там, де це необхідно в силу вимог ефективності або зручності супроводу, кілька компонентів можуть бути об'єднані в один
- 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 необхідної функціональності, досить зручний для супроводу і з точки зору переносимості і не викликає помітних проблем з ефективністю
- Кожен компонент має невеликий, чітко визначене коло вирішуваних завдань і чітко визначений, збалансований за розміром інтерфейс

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

1. Визначити набір сценаріїв дій користувачів або зовнішніх систем, або сценаріїв використання деяких можливостей, які можуть вже бути в вістеме або бути новими. Сценарії повинні бути значущими для конкретних зацікавлених осіб, будь то користувач, розробник, відповідальний за супровід, представник контролюючої організації та ін. Чим повніше набір сценаріїв, тим вищою буде якість аналізу. Можна оцінити частоту появи, важливість сценаріїв.

2. Визначити архітектуру (або кілька порівнюваних архітектур). Це повинно бути зроблено в зрозумілою всім учасникам оцінки формі.

3. Класифікувати сценарії. Для кожного сценарію з набору повинно бути визначено, чи підтримується він даної архітектурою або потрібно вносити в неї зміни, щоб цей сценарій став виконаємо. Сценарій може підтримуватися, тобто його виконання не вимагає внесення змін ні в один з компонентів, або ж не підтримуватися, якщо його виконання вимагає змін в описі поведінки одного або несолько компонентів або змін в їх інтерфейсах. Підтримка сценарію означає, що особа, зацікавлена в його виконання оцінює ступінь підтримки як достатню, а необхідні при цьому дії як досить зручні.

4. Оцінити сценарії. 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.

5. Виявити взаємодію сценаріїв. Визначити які компоненти потрібно змінювати для непідтримуваних сценаріїв, компоненти, які потрібно змінювати для підтримки декількох сценаріїв - такі сценарії називають взаємодіючими, оцінити ступінь смислової пов'язаності взаємодіючих сценаріїв.

Мала зв'язаність за змістом між взаємодіючими сценаріями означає, що компоненти, в яких вони взаємодіють, виконують слабо пов'язані між собою завдання та їх варто декомпозировать.

Компоненти, в яких взаємодіють багато сценаріїв також є можливими проблемними місцями.

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

## 3. UML. Види діаграм UML

Для уявлення архітектури (точніше різних входять до неї структур) зручно використовувати графічні мови. На даний момент найбільш опрацьованим і найбільш широко використовуваним з них є уніфікована мова моделювання (Unified Modeling Language, UML), хоча на високому рівні абстракції архітектуру системи зазвичай описують просто набором іменованих прямокутників, з'єднаних лініями і стрілками, що представляють можливі зв'язки.

Деякі такі мови закріплені у вигляді стандартів, наприклад

• IEEE 1320.1-1998 (R2004) Standard for Functional Modeling Language - Syntax and Semantics for IDEF0

Цей стандарт описує мову ієрархічних діаграм потоків даних.

• IEEE 1320.2-1998 (R2004) Standard for Conceptual Modeling Language - Syntax and Semantics for IDEF1X97 (IDEFobject)

Цей стандарт описує мову опису струтурірованних даних на основі понять сутності, зв'язку і атрибута сутності, використовуваний, зокрема, для моделювання баз даних.

UML пропонує використовувати для опису архітектури 8 видів діаграм. Не всяка діаграма на UML описує архітектуру - 9-й вид діаграм, діаграми варіантів використання, не відносяться до архітектурних уявленням, крім того, і інші види діаграм можна використовувати для опису внутрішньої структури компонентів або сценаріїв дій користувачів і інших елементів, до архітектури не відносяться .

• Статичні структури, що відображають постійно присутні в системі суті і зв'язку між ними, або сумарну інформацію про сутності і зв'язках, або сутності та зв'язки, що існують в якийсь момент часу

о Діаграми класів. Показують типи сутностей системи, атрибути типів (поля і операції) і можливі зв'язки між ними, а також відносини типів між собою по спадкоємства.

Найбільш часто використовуваний вид діаграм.

о Діаграми об'єктів. Показують об'єкти системи та їх зв'язку, в деякому конкретному стані або сумарно.

Використовуються рідко.

о Діаграми компонентів. Це компоненти у вузькому сенсі, компоненти «фізичного» уявлення системи - файли з вихідним кодом, динамічно подгражаемие бібліотеки, HTML-сторінки тощо. Вони визначають розбиття системи на набір сутностей, розглядаються як атомарні з точки зору її збірки і конфігураційного управління.

Використовуються рідко.

о Діаграми розгортання. Показують прив'язку (в певний момент часу або постійну) компонентів системи (у вузькому сенсі) до фізичних пристроїв - машинам, процесорам, принтерів, маршрутизаторів тощо.

Використовуються рідко.

• Динамічні структури, що описують відбуваються в системі процеси

о Діаграми діяльностей. Показують набір процесів-діяльностей і потоки даних, що передаються між ними, а також можливі їх синхронізації один з одним.

Використовуються досить часто.

о Діаграми сценаріїв. Показують можливі сценарії обміну повідомленнями / викликами в часі між різними компонентами системи (в широкому сенсі). Ці діаграми є підмножиною іншого графічного мови - мови діграмм послідовностей повідомлень (Message Sequence Charts, MSC).

Використовуються майже так само часто, як діаграми класів.

о Діаграми взаємодії. Показують ту ж інформацію, що і діаграми сценаріїв, але прив'язують обмін повідомленнями / викликами ні до часу, а до зв'язками між компонентами.

Використовуються рідко.

о Діаграми станів. Показують можливі стану окремих компонентів або системи в цілому, переходи між ними у відповідь на будь-які події і виконувані при цьому дії.

Використовуються досить часто.

Приклади діаграм UML для простої системи ведення банківських рахунків.

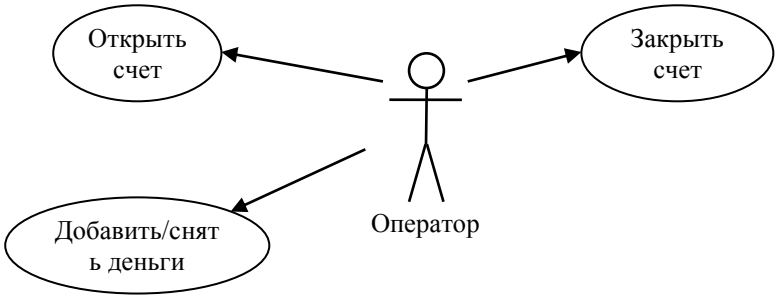



**Рисунок 1. Діаграма варіантів використання (не архітектурна).**

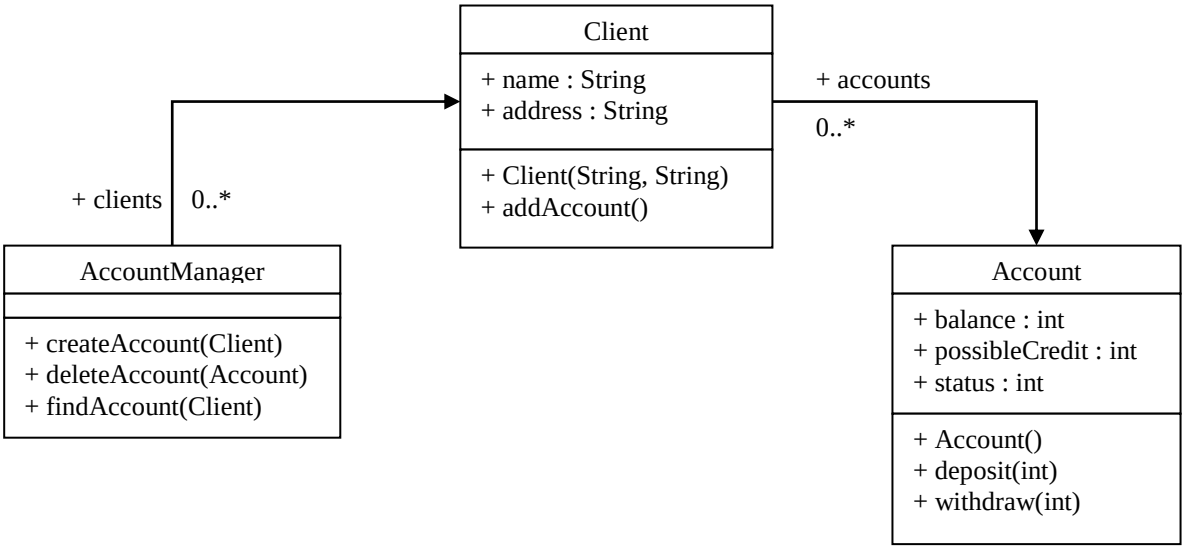



**Рисунок 2. Діаграма класів.**

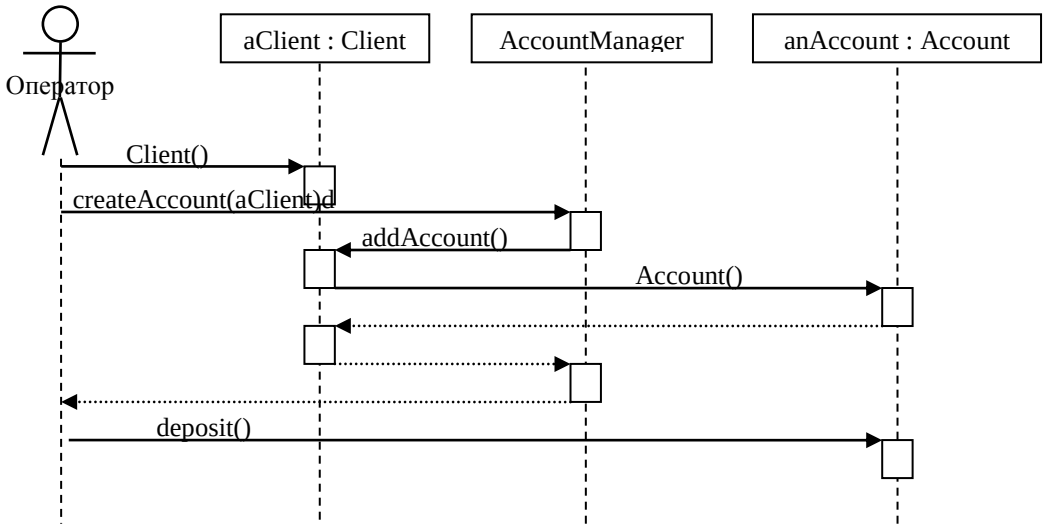



**Рисунок 3. Діаграма сценарію відкриття рахунка.**

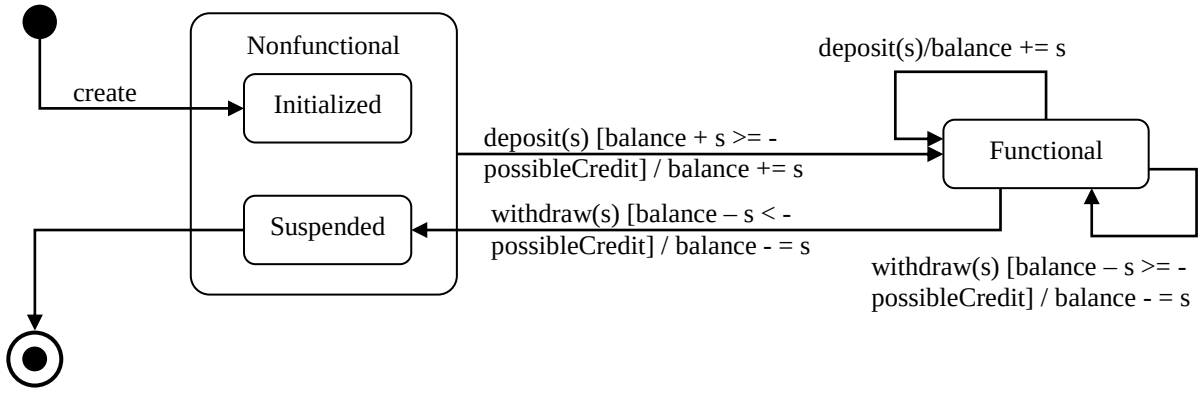



**Рисунок 4. Діаграма станів рахунку.**

## 4. Зразки проектування і архітектурні стилі

На основі наявного досвіду дослідниками і практиками розробки ПО вироблено деякий безліч типових архітектур, знайомство з якими дозволяє не винаходити велосипед для вирішення досить відомих задач. Подібні типові рішення на рівні архітектури називаються архітектурними стилями. Точніше, архітектурний стиль визначає набір типів компонентів системи і набір шаблонів їх взаємодій з передачі даних або управління. Різні архітектурні стилі підходять для вирішення різних завдань в плані забезпечення не функціональних вимог, хоча одну і ту ж функціональність можна реалізувати, використовуючи різні стилі.

Архітектурні стилі є зразками проектування на рівні архітектури. Зразок проектування (design pattern) - це шаблон рішення часто зустрічається завдання проектування, який можна використовувати всякий раз, коли ця задача виникає. Зразки проектування поділяються залежно від масштабу рішень на архітектурні, що визначають можливу декомпозицію системи в цілому або великих підсистем, області відповідальності підсистем і правила їх взаємодії, проектні, що визначають шаблон взаємодій групи компонентів, зазвичай в рамках деякої підсистеми, для вирішення деякої загальної задачі проектування в повторюється контексті, і ідіоми, що визначають спосіб використання мовних конструкцій для вирішення подібних завдань.

| ***Стиль или образец*** | ***Контекст использования*** | ***Примеры*** |
|---|---|---|
| **Конвейер обробки даних (data flow)** | **Система видає вихідні дані в результаті обробки добре певних вхідних, при цьому процес обробки не залежить від часу, застосовується багаторазово, однаково до будь-яких даних на вході.**<br>**Важливою властивістю є чітко визначена структура даних і підтримка можливості інтеграції з іншими системами** | |
| • Пакетна обробка | Один висновок виробляється на основі читання деякого набору даних на вході, проміжні перетворення послідовні | Виконання тестів |
| • Канали і фільтри | Потрібно забезпечити перетворення безперервних потоків даних, перетворення Інкрементальний, наступне може бути розпочато до закінчення попереднього, можливо додавання додаткових перетворень | Утиліти UNIX, компілятори |
| • Замкнений цикл управління | Потрібно забезпечити постійне управління в умовах погано передбачуваних впливів оточення, особливо, якщо система повинна реагувати на зовнішні фізичні події | Системи управління рухом |
| **«Виклик-повернення» (call-return)** | **Порядок виконання дій досить визначено, компонентам нема чого витрачати час на очікування звернення від інших** | |
| • Процедурна декомпозиція | Дані незмінні, процедури роботи з ними можуть трохи змінюватися, можуть виникати нові | Основна схема побудови програм для мов |

| | | |
|---|---|---|
| | | C, Pascal, Ada |
| • Абстракні типи даних | Важливі можливості внесення змін та інтеграції з іншими системами, в системі багато даних, структура яких може змінюватися | Бібліотеки компонентів |
| • Багаторівнева система | Важливі переносимість і можливість багаторазового використання, є природне розшарування системи на специфічні тільки для неї функції та функції загального характеру, специфічні для платформи | Протоколи (модель OSI и реальні) |
| **Незалежні компоненти** | **Можливо розпаралелювання роботи і використання декількох машин, система природно розбивається на слабо пов'язані невеликі компоненти, робота яких може бути організована майже незалежно** | |
| • Клієнт-сервер | Завдання, які вирішуються природно розподіляються між ініціаторами і оброблювачами запитів, можлива зміна зовнішнього представлення даних і способів їх обробки | Основна модель бізнес-застосунків |
| • Розподілені об’єкти | Можливість використання розподіленої архітектури і численні дані з мінливою структурою | |
| **Інтерактивні системи** | **Необхідність досить швидко реагувати на дії користувача, мінливість призначеного для користувача інтерфейсу** | |
| • Дані-представлення-обробник | Зміни в зовнішньому поданні досить вірогідні, одна і та ж інформація може надаватися по-різному в декількох місцях, система повинна швидко реагувати на зміни даних | Document-View в MFC (Microsoft Foundation Classes) |
| • Представлення-абстракція-управління | Інтерактивна система на основі агентів, що мають власні стану і призначений для користувача інтерфейс, можливо додавання нових агентів | |
| **Системы на основе хранилища данных** | **Основні функції системи пов'язані зі зберіганням, обробкою і представленням великої кількості даних** | |
| • Репозиторій | Порядок роботи визначається потоком зовнішніх подій і цілком визначений | Середовища розробки і CASE-системи |
| • Дошка оголошень | Спосіб вирішення завдання в цілому невідомий або занадто трудомісткий, але відомі методи, частково вирішують завдання, композиція яких здатна видавати прийнятні результати, можливо додавання нових споживачів даних або обробників | Системи розпізнавання тексту |

## *Література*

[1] И. Соммервилл. Инженерия программного обеспечения. Вильямс, 2002.
[2] Э. Дж. Брауде. Технология разработки программного обеспечения. Питер, 2004.

[3] М. Фаулер и др. Архитектура корпоративных программных приложений. Вильямс, 2004.
[4] М. Фаулер, К. Скотт. UML в кратком изложении. М., Мир, 1999.
[5] Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. М., ДМК, 2000.
[6] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Питер-ДМК, 2001.
[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. Wiley, 2002.
[8] L. Bass, P. Clements, R. Kazman. Software Architecture in Practice. 2-nd edition, Addison-Wesley, 2003.

# Лекція 3 Принципи встановлення вимог розробки системи

## 3.1 Принципи встановлення вимог

Встановлення вимог - перший етап життєвого циклу розробки системи. Система, що підлягає розробці, визначається за допомогою виду діяльності, котрий отримав назву системного планування. Мета встановлення охоронних вимог полягає в тому, щоб дати розгорнуте визначення функціональних - а також не функціональних - вимог, які учасники проекту очікують затвердити в реалізованої і розгортається системі.

Вимоги визначають послуги, очікувані від системи (формулювання сервісів) і обмеження, яким система повинна підкорятися (формулювання обмежень). Ці формулювання сервісів можна об'єднати в кілька груп: одна з груп що описує кість кордону системи, друга - необхідні бізнесфункції (функціональні вимоги), а третя - необхідні структури даних (вимоги до даних). Формулювання обмежень можна класифікувати відповідно до різними категоріями обмежень, що накладаються на систему, такі як необхідну "враження і відчуття від використання програми", продуктивність, безпеку і т.д.

Вимоги необхідно отримати від замовників (користувачів і власників сис теми). Цей вид діяльності, званий виявленням вимог (requirements elicita tion), здійснюється аналітиком бізнес-процеси (або системним аналітиком). Для цього підходять різні методи, починаючи з традиційних інтерв'ю з замов кому і закінчуючи (при необхідності) побудовою програмного прототипу, з по міццю якого розкриваються додаткові вимоги.

Зібрані вимоги повинні бути піддані ретельному аналізу для виявила лення в них повторів і суперечностей. Це незмінно призводить до перегляду охоронних вимог і їх повторному погодженню з замовниками.

Вимоги, задовільні з точки зору замовника, документуються. При цьому вимогам дають визначення, класифікують, нумерують і привласнюють їм пріоритети. Структура документа, що описує вимоги, відповідає шаблона (template), заданої в організації для цієї мети.

Хоча документ, що фіксує вимоги, носить в значній мірі описатель ний характер, цілком можливо включити в нього високорівневу схематичну біз несмодель. Як правило, бізнесмодель складається з моделі кордонів системи, моделі бізнес неспрецедентов і моделі бізнескласса.

Вимоги користувача подібні рухомій цілі. Щоб впоратися з зрад стійкими проти вимог, необхідно вміти управляти змінами. Управління вимогами включає такі види діяльності як оцінка впливу змін одних вимог на інші, а також на незмінну частину системи.

## 3.2. Виявлення вимог

Бізнес Аналітик виявляє вимоги до системи за допомогою консультацій. До участі в консультаціях залучаються замовники і експерти в проблемній області. У деяких випадках БІЗНЕСАНАЛІТИКА володіє достатнім досвідом у проблемної області, і допомога експерта може бути зайвими. В цьому випадку БІЗНЕСАНАЛІТИКА є різновидом Експерта проблемної області, що відображено в моделі, показаної на рис. 3.1, за допомогою відносини узагальнення. (Слід, однак, мати на увазі, що рис. 3.1 - це не модель прецедентів: нотація для прецедентів використовується тут тільки для зручності.)

Вимоги, виявлені за допомогою експерта проблемної області, становлять основу знань про проблемою області. Вони фіксують широко визнані, чи не залежать від часу бізнес-правила, застосовні до більшості організацій і систем. Вимоги, виявлені в ході консультацій з замовниками, виражаються в сценаріях прецедентів. Вони виходять за рамки базових знань про проблемну область і фіксують унікальні риси організації - спосіб ведення бізнесу "тут і зараз" Чи бо побажання щодо того, як слід вести бізнес.

Завдання БІЗНЕСАНАЛІТИКА полягає в тому, щоб об'єднати два набору вимог в бізнес-моделі. Як показано на рис. 3.1, Бізнес-модель містить Модель бізнес-класів та Модель бізнес-прецедентів. Модель бізнес-класів перед ставлять собою діаграму класів верхнього рівня, яка ідентифікує і зв'язує між собою бізнесоб'екти. Модель бізнес-прецедентів - це діаграма прецедентів верхнього рівня, яка ідентифікує основні функціональні будівельні блоки системи.

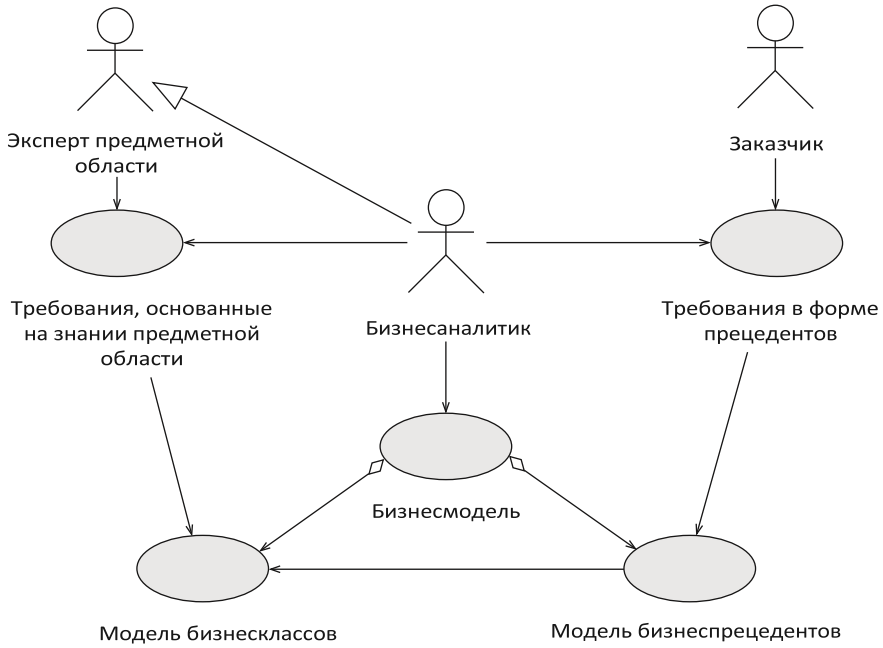



Рис. 3.1. Взаємні впливу моделей, характерні для процесу визначення вимог

У загальному випадку, класи проблемної області (бізнесоб'екти) не повинні виводи дивіться з прецедентів [75]. Однак, на практиці правильність моделі бізнес класів повинна підтверджуватися за допомогою порівняння з

Моделлю бізнеспрецедентов. Це порівняння, як правило, призводить до деякої налаштуванні або розширенню Моделі бізнес-класів.

Дотримуючись Хоффер (Hoffer) [37] ми розрізняємо традиційні і сучасні мето ди виявлення фактів і збору інформації.

### 3.2.1. Традиційні методи виявлення вимог

До традиційних методів виявлення вимог відносяться використання інтерв'ю і анкети, спостереження і вивчення ділових документів. Це прості і економічних методи.

Ефективність традиційних методів обернено пропорційна ризику проекту. Високий ризик означає, що систему важко реалізувати - не цілком ясні навіть узагальнені вимоги. Для таких проектів традиційних методів навряд чи буде достатньо.

#### *3.2.1.1. Інтерв'ювання замовників і експертів в проблемної області*

Використання інтерв'ю являє собою основний метод виявлення фактів і збору інформації. Більшість інтерв'ю проводяться з замовниками. Інтерв'ю з замовниками дозволяють виявити здебільшого "прецедентні" вимоги, тобто вимоги випливають з "прецедентів" (рис. 3.1). Якщо БІЗНЕСАНАЛІТИКА не володіючи достатнім досвідом в проблемній області, можна також взяти інтерв'ю у відповідних експертів.

Інтерв'ю з експертами в прикладної області найчастіше є про сто процес передачі знань - заняття з навчання БІЗНЕСАНАЛІТИКА. Інтерв'ю з замовниками відрізняє велика складність [46], [81].

Замовники можуть мати дуже туманне уявлення про свої вимоги. Вони можуть не бажати співпрацювати або не вміють висловлювати свої вимоги в зрозумілій формі. Вони також можуть запитувати реалізацію вимог, які перевершують бюджет проекту або реалізовуються. І нарешті, досить імовірно, що вимоги, ис ходять від різних груп користувачів, можуть виявитися суперечливими.

Існують два основних типи інтерв'ю: структуроване (формальне) і не структуроване (неформальне). Структуроване інтерв'ю готується заздалегідь, відрізняється ясністю постановки питань, а багато питань можуть бути задані заздалегідь. Заздалегідь сформульовані питання можна розділити на дві категорії: питання с від критим безліччю відповідей (openended question) (відповіді на цю категорію питань зарані не готуються) і питання с замкнутим безліччю відповідей (closedended question) (відповіді на цю категорію питань можна вибрати зі списку підготовлених відповідей).

Структурованого інтерв'ю повинно супроводжувати неструктуроване інтерв'ю. Неструктуроване інтерв'ю більше нагадує неформальну зустріч, до торою не властиві заготовлені про запас питання або заздалегідь поставлені це чи. Мета неструктурованого інтерв'ю - спонукати замовника до того, щоб він

поділився своїми думками і в процесі бесіди підійшов до вимог, яких бізнес-аналітик міг і не чекати і, отже, не міг поставити потрібні питання.

Як структуроване, так і неструктуроване інтерв'ю потребують до деяких законів рій відправної точки і контексті для обговорення. Це може бути невеликий документ або записка, відправлена по електронній пошті беруть інтерв'ю перед зустріччю, мета яких - пояснити цілі інтерв'юера або поставити деякі запитання.

Існують три категорії питань, яких, в загальному випадку, необхідно мати за [91].

1. Небеспристрасні питання, в яких інтерв'юер висловлює (прямо чи опосередковано) свою думку з питання ( "Чи повинні ми працювати так, як ми працюємо?").
2. Предвзяті питання, аналогічні небезстороннім, але відрізняються від останніх тим, що думка інтерв'юера є, очевидно, тенденційним ( "Ви ж не станете цього робити, чи не так?").
3. Наводящі питання, які припускають відповідь в самому питанні ( "Ви ж зробите саме так, чи не так?").

Успіх інтерв'ю залежить від багатьох факторів, але чи не головними серед них є навички інтерв'юера в області комунікації і міжособистісного спілкування. Хоча основне завдання інтерв'юера - задавати питання і володіти ситуацією, не менше важлива але в ході бесіди уважно слухати і проявляти терпіння до беруть інтерв'ю так, щоб він відчував себе невимушено. Для збереження хороших особистих від відносин і в розрахунку на позитивний відгук з боку інтерв'юйованого необхідно відправити йому протягом одного-двох днів після інтерв'ю записку, утримуючи короткі підсумки інтерв'ю.

*3.2.1.2. Анкетування*

Використання анкет або анкетування (questionnaires) - ефективний спосіб збору інформації від багатьох замовників. Зазвичай анкети використовуються в доповнення ня до інтерв'ю, а не замість них. Виняток можуть становити проекти з низьким ризиком, цілі яких ясно окреслені. Для таких проектів зазвичай досить іс користуватися анкети з питаннями, які носять пасивний характер і не відрізняються великою глибиною.

У загальному випадку, анкетування менш продуктивно, ніж використання інтерв'ю, оскільки в питання або можливі відповіді не можна внести додаткову ясність. Анкетування пасивно - це можна розцінювати і як гідність, і як недостатньо струм. Це гідність, оскільки у респондента є час подумати над відповіддю, а анкета може залишитися анонімною. Це недолік, оскільки респонденту нелегко прояснити для себе питання.

Анкета (або опитувальний лист) повинна бути розроблена таким чином, щоб, по можли вості, полегшити відповіді на питання. Зокрема, слід уникати питань з невизначеним ленним безліччю відповідей - більшість питань повинні

ставитися до питань із замкнутим списком відповідей. Подібні питання можуть набувати трьох форм [91].

1. Багатоальтернативні питання (multiplechoice questions). При відповіді на ці питання респондент повинен вказати один або більше відповідей, вибравши їх із прикладеного списку. Крім того, іноді допускаються додаткові коментарі до питань з боку респондентів.
2. Рейтінгові питання (rating questions). При відповіді на цей тип питань респондент повинен висловити свою думку щодо висловленого твердження. Для цього можуть використовуватися такі рейтингові значення, як "абсолютно згоден", "згоден", "ставлюся нейтрально", "не згоден", "абсолютно не згоден" і "не знаю".
3. питання з ранжуванням (ranking questions). Цей тип питань передбачає ранжування відповідей за допомогою привласнення ним послідовних номерів, процентних значень і використання інших засобів упорядкування.

Ретельно продумана, що полегшує відповіді анкета заохочує респондента для того, щоб відразу повернути заповнений документ. Однак, при оцінці результатів анкетування БІЗНЕСАНАЛІТИКА повинен передбачити можливість спотворення інформації через те, що ті люди, які не відповідали на питання, могли б відповісти на них інакше, ніж брали участь в опитуванні [37].

*3.2.1.3. Спостереження*

У деяких ситуаціях БІЗНЕСАНАЛІТИКА знаходить важким отримати повну ін формацію з використанням інтерв'ю і анкет. У замовника з тих чи інших причи нам може бути відсутнім можливість (або вміння) надати необхідну ін формацію, або він може не мати повного уявлення про бізнес-процеси в ціло. У подібних випадках в якості ефективного методу виявлення фактів може виступати спостереження (observation). Какнікак, кращий спосіб навчитися зав'язувати гал стукіт - поспостерігати за процесом.

Спостереження може виступати в двох формах.

1. Пасивне спостереження (passive observation), в ході якого БІЗНЕСАНАЛІТИКА спостерігає за різними видами ділової діяльності без втручання або прямої участі в них. У деяких випадках для того, щоб спостереження було якомога менше нав'язливим, можна навіть використовувати відеокамери.
2. Активне спостереження (active observation), в ході якого БІЗНЕСАНАЛІТИКА бере участь в діяльності і стає фактично частиною команди.

Щоб результати спостережень були представницькими, спостереження необхідно проводити протягом тривалого періоду часу, в різні часові інтервали і при різній робочому навантаженні (вибіркові періоди). Основна праця пов'язана з наглядом, полягає в тому, що люди, за якими спостерігають, схильні поводитися інакше, ніж у звичайній обстановці. Зокрема, вони прагнуть

працювати відповідно до формальних правил і процедур. Це призводить до спотворення реального стану справ за рахунок приховування "раціональних" прийомів рабо ти - як позитивних, так і негативних. Слід пам'ятати, що "робота по правилам" - це одна з ефективних форм страйкового руху.

*3.2.1.4. Вивчення документів і програмних систем*

Вивчення документів і програмних систем є неоціненним методом виявлення як вимог типу прецедентів, так і вимог, пов'язаних зі знанням про проблемної області *(див. Розд. 3.2)*. Цей метод використовується завжди, хоча він може стосується тільки окремих сторін системи.

Вимоги, що формулюються у вигляді прецедентів, або прецедентні вимоги (use case requirements) виявляються за допомогою вивчення існуючих організаційних до ментів і системних форм і звітів (якщо, звичайно, для поточної системи існують автоматизовані рішення, що типово для великих організацій). Одним з найбільш корисних способів осягнення суті вимог на основі прецедентів є фіксація дефектів (природно, при їх наявності) та формування запитів на зміну для існуючої системи.

До організаційних документів, що підлягають вивченню, належать: форми ділових документів (по можливості - заповнені), опису робочих процедур, має обов'язки, методичні керівництва, бізнес-план, схеми організаційних них структур, внутріофісну кореспонденція, протоколи нарад, облікові записи, зовнішня кореспонденція, скарги клієнтів і т.д.

Системні форми і звіти, що підлягають вивченню, включають: копії екранів, від подружжя разом з відповідною документацією - системні керівництва по експлутації, призначена для користувача документація, технічна документація, системні моді чи аналізу і проектування і т.д.

Вимоги, засновані на знанні проблемної області, з'ясовуються за допомогою вивчення журналів і підручників, які відносяться до даної сфери. Вивчення патентованих програмних пакетів на зразок ERPсістем (Enterprise Resource Planning Systems - системи планування ресурсів підприємства) також може стати багатим джерелом знань про проблемну область. Отже, відвідування бібліотек і постачальників ПО є частиною процесу виявлення вимог (звичайно, Inter net дозволяє здійснити багато такі "візити", не залишаючи офісу).

### 3.2.2. Сучасні методи виявлення вимог

До сучасних методів виявлення вимог відноситься використання про програмних прототипів, а також такі методи, як JAD (Joint Application Develop ment - спільна розробка додатків) і RAD (Rapid Application Development - швидка розробка додатків). Ці підходи пропонують більш глибоке проник новение в суть вимог, але за рахунок більшої ціни і зусиль. Однак, долговремен ні вкладення в ці методи можуть окупитися з лишком.

Застосування сучасних методів зазвичай пов'язане з високим проектним ризиком. Існує безліч факторів, що обумовлюють високий ризик проекту. До таких факторів належать неясні цілі, недокументовані процедури, нестабільні вимоги, слабке знання справи користувачами, недосвідчені розробники, недостатньо точна прихильність користувачів розробці і т.д.

*3.2.2.1. Прототипування*

Прототипування (prototyping) - це найбільш часто використовуваний сучасний метод виявлення вимог. Програмні прототипи конструюються для візуалізації системи або її частини для замовників з метою отримання їх відгуків.

Прототип являє собою демонстраційну систему - "нашвидку і грубо" зроблену робочу модель рішення, яка представляє користувальницький GUI інтерфейс і моделює поведінку системи при ініціюванні користувачем раз особистих подій. Інформаційне наповнення екранів частіше жорстко запрограмованих в програмі прототипу, ніж виходить автоматично з бази даних.

Складність (і зростаючі "апетити" замовників) сучасних GUIінтерфейсов роблять прототипування обов'язковим елементом розробки ПО. Прототипи дозволяють досить непогано оцінити достовірність, і корисність системи до початку її реалізації.

У загальному випадку, прототип - це дуже ефективний спосіб виявлення охоронних вимог, які важко отримати від замовника за допомогою інших засобів. Найчастіше така ситуація зустрічається для систем, які повинні надати в розпорядження користувачів нові бізнес-функції. Подібна ситуація також характерна для випадків суперечливих вимог і наявності проблем в кооперації між замовниками і розробниками.

Існують дві основні різновиди прототипів [46].

1. "Одноразовий" прототип ( "throwaway" prototype), який після того, як виявлення вимог завершено, просто відкидається. Розробка "одноразового" прототипу націлена тільки на етап встановлення вимог ЖЦ ПО. Як правило, цей прототип концентрується на найменш зрозумілих вимогах.
2. Еволюційний прототип (evolutionary prototype), який зберігається після виявлення вимог і використовується для створення кінцевого програмного продукту. Еволюційний прототип націлений на прискорення поставок товарів. Як правило, він концентрується на ясно викладених вимогах, так що першу версію продукту можна надати замовнику досить швидко (хоча її функціональні можливості, як правило, неповні).

Додатковим аргументом на користь використання саме "одноразового" прототипа може служити прагнення уникнути ризику "консервації" швидких і

грубих і, як наслідок, неефективних рішень в кінцевому продукті. Однак міць і гнучкість з тимчасових засобів створення ПО послаблюють цей аргумент. Якщо управління проектом здійснюється належним чином, то причини, по якій можна було б позбутися неефективних запропонованих для прототипу рішень, не існує.

*3.2.2.2. Спільна розробка додатків (JAD метод)*

JAD метод повністю відповідає своїй назві - це спільна розробка додатків (Joint Application Development), що здійснюється в ході одного або декількох нарад із залученням всіх учасників проекту (замовників і розробників) [95]. Хоча ми відносимо JAD-підхід до сучасних методів виявлення вимог, цей метод був вперше введений в кінці 1970х років компанією IBM.

Існує багато різновидів JAD-метода і багато фірм, що пропонують послуги по організації і проведенню JAD-нарад. Проведення JAD-нарад може займати кілька годин, кілька днів або навіть пару тижнів. Кількість учасників не повинно перевищувати 2530 осіб. У нараді бере участь певне коло осіб [37], [91].

1. Ведучий - людина, яка проводить і модерує зустріч (тому іноді його називають модератором). Ця людина повинна володіти винятковими здібностями в області комунікації, не повинен ставитися до числа учасників проекту (крім того, що він є лідером JADсессіі), володіє ґрунтовним знанням проблемної області (проте не обов'язково добре володіє проблемами розробки ПО).
2. Секретар - людина, яка фіксує хід JAD-сесії з використанням комп'ютера. Ця людина повинна вміти швидко вводити текст в комп'ютер і добре володіти питаннями розробки ПО. Для документальної фіксації ходу сесії і розробки первинних моделей рішень секретар може використовувати CASEсредства.
3. Замовник (користувачі або керівники) - це основні учасники, які викладають і обговорюють вимоги, приймають рішення, стверджують проектні цілі і т.д.
4. Разробник - бізнесаналітик і інші члени проектної бригади. Ці учасники більше слухають, що говорять - вони присутні на нараді для того, щоб усвідомити фактичний бік справи і зібрати інформацію, а не позичати цілком увагу інших учасників в процесі наради.

JAD-метод ґрунтується на груповій динаміці. Групові зусилля більш перспективні з точки зору отримання найкращих рішень проблем. Групи сприяють підвищенню продуктивності, швидше навчаються, схильні до більш кваліфікованим висновками, дозволяють виключити багато помилок, приймають ризиковані рівня (іноді це може носити негативний характер!), Концентрують увагу навчаючи на найбільш важливих питаннях, об'єднують людей і т.д.

Якщо JAD сесія проводиться відповідно до правил, можна домогтися хороших результатів. Однак, не слід забувати про попередження: "... компанія Ford Motor в 1950х роках зазнала невдачі, намагаючись вивести на ринок модель Edsel - автомобіль, розроблений комітетом" [95].

*3.2.2.3. Швидка розробка додатків (RAD-метод)*

Метод швидкої розробки додатків (Rapid Application Development - RAD) - це не що більше, ніж метод виявлення вимог - це цілісний підхід до розробки ПЗ [37]. Як зрозуміло з назви методу, він передбачає швидку поставку системних рішень. Технічна перевага відступає на друге місце в порівнянні з бо ростью поставки.

*3.3. Узгодження і перевірка обґрунтованості вимог*

Згідно Вуду (Wood) і Сильверу (Silver) [95] технологія RAD поєднує в собі п'ять методів, перерахованих нижче.

1. Еволюційне прототипування.
2. CASE-засоби з можливостями генерації програм і циклічної розробкою з переходом від проектних моделей до програми і назад.
3. Спеціалісти, які володіють розвиненими інструментальними засобами (Specialists with Ad vanced Tools - SWAT) – RAD-бригада розробників. Кращі аналітики, проектувальники і програмісти, яких тільки може залучити організація. Бригада працює в рамках суворого тимчасового режиму і розміщується разом з користувачами.
4. Інтерактивний JAD-метод – JAD-сесія, під час якої секретар замінюється бригадою SWAT, оснащеної CASEсредствамі.
5. Жорсткі тимчасові рамки (timeboxing) - метод управління проектом, який відводить бригаді SWAT фіксований період часу (timebox) для завершення проекту. Цей метод перешкоджає "розповзання рамок проекту"; якщо проект затягується, то рамки рішення звужуються, щоб дати можливість завершити проект вчасно.

Використання RAD-підходу може виявитися привабливим варіантом для багатьох проектів, особливо, для невеликих проектів, які не зачіпають сферу ключових бізнес-процеси організації, і які, таким чином, не ставлять план рішення для інших проектів з розробки ПЗ. Малоймовірно, щоб швидкі рішення були оптимальними або довготривалими для ключових сфер діяльності організації. З використанням RAD-метода пов'язаний ряд проблем; деякі з них перераховані нижче.

1. Несумісний проект GU-Iінтерфейса.

2. Замість загальних рішень, що сприяють багаторазовому використанню ПО, спеціалізовані рішення.

3. Неповна документація.

4. Важке для підтримки і масштабування ПО і т.д.

### 3.3. Узгодження і перевірка обґрунтованості вимог

Вимоги, отримані від користувачів, можуть дублюватися або суперечать один одному. Деякі вимоги можуть бути неясними або нереальними Інші вимоги можуть залишитися нез'ясованими. З цієї причини перш, ніж вимоги потраплять в документ опису вимог, їх необхідно узгодити.

Насправді узгодження і перевірка обґрунтованості вимог здійснює ся паралельно з виявленням вимог. Після того як вимоги виявлені, вони піддаються певному рівню перевірки. Для всіх сучасних методів ви явища вимог, які пов'язані з так званої "груповою динамікою", це цілком природно. Як би там не було, після того як виявлені вимоги зібрані разом, вони в будь-якому випадку повинні бути піддані ретельному обговоренню і перевірці.

Узгодження і перевірка вимог не можуть бути відокремлені від процесу підготовки документа опису вимог. Зазвичай погодження розбіжностей між вимогами засноване на чорновому варіанті документа. Вимоги, перераховані в чорновому варіанті до документу, у якому обговорюються і при необхідності модифікуються. Побічні вимоги видаляються. Нововиявлені вимоги додаються.

Для перевірки обґрунтованості вимог (requirements validation) необхідна більш повна версія документа, в якому всі вимоги чітко ідентифіковані і класифіковані. Учасники проекту вивчають документ і проводять наради з їх формальному перегляду. Перегляди (reviews) часто структуровані у вигляді так званих процедур наскрізного контролю (walkthroughs) або інспекцій (inspections). Пере огляди є різновидом тестування (testing) (розд. 10.1.1).

#### *3.3.1. Вимоги, що виходять за рамки проекту*

Вибір проекту в сфері IT і, отже, підлягає реалізації системи (і їх чітких меж) визначається в рамках виду діяльності, який отримав назву системного планування. Однак, деталізовані взаємозалежності між системами можуть бути розкриті лише на етапі аналізу вимог. Установлення меж системи (системних рамок) - завдання етапу аналізу вимог, так що проблема "розтягування рамок" може вирішуватися на ранніх етапах процесу розроблення як складова частина цього завдання.

Щоб мати можливість вирішити, чи лежить конкретну вимогу в рамках системи або виходить за рамки, необхідна еталонна модель, по відношенню до якої і повинно прийматися рішення. Історично роль подібної еталонної моделі сиг рала контекстна діаграма (context diagram) - головна міжнародна діаграма популярно го методу структурного моделювання під назвою діаграма потоків даних (Data Flow Diagrams - DFD). Хоча в мові UML місце цієї діаграми зайняла діаграма прецедентів, контекстна діаграма, як і раніше залишається чудовим методом встановлення меж системи.

Існують, втім, і інші причини, за якими вимога може бути розцінено, як виходить за рамки проекту [81]. Наприклад, вимога може становити велику трудність для реалізації в комп'ютеризованої системи і залишається

прерогативою людини, що бере участь в процесі. Або вимога може мати низький пріоритет і виключається з першої версії системи. Вимога може бути також реалізовано за допомогою апаратного забезпечення або зовнішніх пристроїв і, таким чином, може виявитися поза контролем з боку програмного забезпечення.

### *3.3.2. Матриця залежності вимог*

Вважаючи, що всі вимоги чітко ідентифіковані і пронумеровані, можна сконструювати матрицю залежностей вимог (requirements dependency matrix) (або матрицю взаємодії (interaction matrix вимог)) [81], [46]. У стовпці і рядку заголовка перераховані впорядковані ідентифікатори вимог, як показано на рис. 3.2.

## 3.3. Узгодження і перевірка обґрунтованості вимог

| **вимоги** | **Т1** | **Т2** | **Т3** | **Т4** |
|---|---|---|---|---|
| **Т1** | X | X | X | X |
| **Т2** | Конфлікт | X | X | X |
| **Т3** | | | X | X |
| **Т4** | | Перекриття | Перекриття | X |

Рис. 3.2. Матриця залежності вимог

Верхня права частина матриці (над діагоналлю включно) не використовується. Решта осередки вказують на те, перекриваються чи два будь-яких вимоги, протиречивими один одному або незалежні один від одного (порожні клітинки). Суперечливі вимоги необхідно обговорити з замовниками і при можливості Переформулювати для пом'якшення протиріч (фіксацію протиріччя, видиму для наступної розробки, необхідно зберегти). Перекриваються вимоги також повинні бути сформульовані заново, щоб виключити збіги.

Матриця залежності вимог являє собою простий, але ефективний метод виявлення протиріч перекриттів, коли кількість вимог відносно невелика. В іншому випадку описаний метод все ж можна застосувати, якщо вдається згрупувати вимоги по категоріям (розд. 3.4.1), а потім порівняти їх окремо в межах кожної категорії.

### *3.3.3. Ризики і пріоритети вимог*

Після того, як в результаті зняття протиріч і усунення повторив вироблений переглянутий набір вимог, їх необхідно піддати аналізу ризиків і призначити їм пріоритети. Аналіз ризиків (risk analysis) спрямований на ідентифікацію вимог, які є потенційними джерелами працю ностей в розробці. Призначення пріоритетів (prioritization) необхідно для того, що б забезпечити можливість без праці змінити рамки проекту в разі виникнення непередбачених затримок.

Вимоги можуть бути "ризикованими" внаслідок впливу різних чинників.

Вимогам властиві такі типові види ризиків [81].

1. Технічний ризик, коли вимога технічно важко реалізувати.
2. Ризик, пов'язаний зі зниженням продуктивності, коли вимога - будучи реалізовано - може несприятливо позначитися на часі реакції системи.
3. Ризик, пов'язаний з порушенням безпеки, коли вимога - будучи реалізовано - може створити проломи в захисті системи.
4. Ризик, пов'язаний з процесом розробки, коли для реалізації вимоги необхідно використання незвичайних методів розробки, незнайомих розробникам (наприклад, методів формальної специфікації).
5. Ризик, пов'язаний з порушенням цілісності баз даних, коли вимога не може бути легко перевірено і може привести до суперечливості даних.
6. Політичний ризик, коли вимога може виявитися важким виконати через внутрішньополітичні причини.
7. Ризик, пов'язаний з порушенням законності, коли вимога може привести до порушення чинних законів або очікуваної зміни закону.
8. Ризик пов'язаний з мінливістю, коли вимога може потенційно змінюватися або еволюціонувати протягом процесу розробки.

В ідеалі пріоритети вимогам призначають окремі замовники в процесі ви явища вимог. Потім вони узгоджуються на нарадах і знову змінюються після додатки до них факторів ризику.

Для виключення невизначеності і полегшення призначення пріоритетів коли чество груп пріоритетів не повинно бути занадто велике. Зазвичай використовується від трьох до п'яти пріоритетів. Їх можна позначити як "високий", "середній", "низький" і "невизначений". Альтернативний перелік пріоритетів може виглядати, на приклад, так: "істотне", "корисне", "важко визначити" і "відкладене".

### 3.4. Управління вимогами

Вимогами необхідно управляти. Управління вимогами в дійсності ності є частиною загального управління проектом. Воно пов'язане з трьома основними питаннями.

1. Ідентифікація, класифікація, організація і документування вимог.

2. Зміна вимог (за допомогою процесів, які встановлюють спосо б висунення, узгодження, перевірки достовірності та документування неминучих змін до вимог).

3. Відстеження вимог (за допомогою процесів, які підтримують відносини взаємозалежності між вимогами та іншими системними ар тефактамі, а також, власне, між вимогами) (зверніться до глави 10).

*3.4.1. Ідентифікація та класифікація вимог*

Вимоги описуються природною мовою, наприклад, наступним чином.

1."Система повинна запланувати наступний телефонний дзвінок клієнту за запитом Телемаркетер".

2."Система повинна автоматично набирати запланований телефонний номер і одночасно відображати на екрані Телемаркетер інформацію, що включає телефонний номер, номер клієнта, ім'я клієнта".

3."У разі успішного з'єднання система повинна відобразити вступний текст, з яким телемаркетер може звернутися до клієнта для того, щоб зав'язати розмову".

Типова система може складатися з сотень або тисяч формулювань вимог, на зразок тих, що наведені вище. Для належного управління такою величезною кількістю вимог їх необхідно пронумерувати за допомогою деякої схеми ідентифікації (identification scheme) Схема може включати класифікацію вимог у вигляді груп, які легше піддаються управлінню.

Існує кілька методів ідентифікації та класифікації вимог [46].

Унікальний ідентифікатор - зазвичай послідовний номер, присвоєний вручну або згенерований з використанням бази даних CASE-засоби (тобто бази даних (або сховища), в якій CASE-засоби зберігають артефакти, вироблені в результаті аналізу або проектування).

*3.4.2. Управління вимогами*

1. Послідовний номер всередині ієрархії документа - присвоюється з урахуванням положення вимог в межах документа опису вимог (наприклад, сьоме вимога в третьому розділі другого розділу може бути пронумеровано як 2.3.7).
2. Послідовний номер в межах категорії вимог - присвоюється на додаток до мнемонічному імені, яке позначає категорію вимог (де під категорією вимог увазі функціональне вимога, вимога до даних, вимоги до продуктивності, вимоги до безпеки і т.д.)

Кожен з методів ідентифікації має свої за і проти. Найбільшою гнучкістю і захищеністю від помилок відрізняються унікальні ідентифікатори, генеруємі за допомогою бази даних. Бази даних мають вбудовані можливості ми генерації

унікальних ідентифікаторів для кожного нового запису даних в умовах одночасного доступу до даних з боку багатьох користувачів.

Деякі бази здатні додатково підтримувати супровід не скількох версій однієї і тієї ж записи (за рахунок розширення значення унікального ідентифікатора за допомогою номера версії). Нарешті, бази даних можуть мати можливості підтримки посилальної цілісності зв'язків між артефактами моделювання, включаючи вимоги, і можуть, таким чином, забезпечити необхідну підтримку управління та простежуваності вимог.

*3.4.3. Ієрархії вимог*

Вимоги можна впорядкувати у вигляді ієрархічно впорядкованої структури, що представляє відношення батько-нащадок. Ставлення родитель-потомок подібно відношенню композиції (розд. 2.1.4). Батьківське вимога складено з дочірніх вимог. Дочірнє вимога - це фактично "підвимоги" батьківського вимоги.

Ієрархічні відносини дозволяють ввести додатковий рівень класифікації вимог. Це може безпосередньо відображатися (або не відбиватися) в ідентифікаційнім номері (за допомогою використання точки в запису складеного име ні). Тому вимога, пронумерована як 4.9, може бути дев'ятим "нащадком" "батька" з ідентифікаційним номером, рівним 4.

Наведений нижче набір формулювань є ієрархічно упорядковані вимоги.

1. "Система повинна запланувати наступний телефонний дзвінок клієнту за запитом Телемаркетер".

1.1. "Система повинна активізувати клавішу Next Call (Наступний дзвінок) при відкритті форми Telemarketing Control (Управління Телемаркена Тінг) або при завершенні попереднього виклику".

1.2. "Система повинна видалити дзвінок з початку черги запланованих дзвінків і надати йому статус поточного дзвінка".

1.3. Інші вимоги ...

Ієрархії вимог дозволяють визначати вимоги на різних рівнях абстракції. Це узгоджується із загальним принципом моделювання, відповідно документу, котрим при переході до більш низького рівня абстракції моделі систематично збагачуються все новими деталями. В результаті для батьківських вимог можна сконструювати узагальнені вимоги, а деталізовані моделі можна замовити з дочірніми вимогами.

*3.4.4. Управління змінами*

Вимоги змінюються. Вимоги можуть змінюватися, вони можуть бути видалені, а ти нові вимоги можуть бути додані на будь-якому етапі розробки. Зміни можна рас розглядати як "удар", а ось некеровані зміни - це справжній удар по проекту.

Чим далі просунулася розробка, тим дорожче обходиться внесення змін. Фактично, мінімальні витрати, необхідні для виправлення проекту після внесення змін, завжди ростуть і часто ростуть експоненціально. Зміна щойно сформульованих вимог, які не пов'язані з іншими вимогами, - це проста вправа в редагуванні. Зміна тих же вимог, після того як вони реалізовані, може обійтися надзвичайно дорого.

Зміни можуть бути пов'язані з суб'єктивними помилками, але часто вони вимушені звертатися покликані змінами внутрішньої політики або зовнішніми факторами, такими як конкурентні сили, глобальні ринки або технологічні досягнення. Поза залежності від причин, для документування запитів на зміни (change request) оцінки впливу, що чиниться змінами (change impact), і здійснення змін необхідна сильна стратегія управління змінами.

Оскільки зміна вимог обходиться дорого, для кожного запиту на зміни необхідно створювати формальний бізнес-прецедент. Обґрунтована зміна, яке не зустрічалося раніше, оцінюється з точки зору його технічної реалізації, впливу на решті проект і витрат. Після узгодження вимога включає в відповідні моделі і реалізується в програмному забезпеченні.

Управління змінами, крім іншого, включає відстеження великих обсягів по мов взаємозалежної інформації протягом тривалого періоду часу. Без інструментів підтримки управління змінами приречене. В ідеалі, зміни охоронних вимог повинні зберігатися і відслідковуватися за допомогою засобів конфігураційного управління ПЗ, яке використовується розробниками для контролю версій моделей і програм протягом ЖЦ розробки. Відповідні CASE-засоби повинні або мати власні можливості конфігураційного управління, або можливості взаємодії з автономними засобами конфігураційного управління.

*3.4.5. Відстеження вимог*

Відстеження вимог (requirements traceability) - це всього лише частина, хоча і вкрай важлива частина, управління змінами (change management). Блок простежується мости вимог технології управління змінами підтримує відносини простежуваності, щоб фіксувати зміни, які виходять з або вносяться до вимог протягом ЖЦ розробки.

Розглянемо вимога: "Система повинна запланувати наступний телефонний дзвінок клієнту за запитом Телемаркетер". Ця вимога можна згодом смо деліровать за допомогою діаграми послідовностей, яка активізується з GUIінтерфейса за допомогою кнопки запуску дії, позначеної як Next Call, і тригера, запрограмованого в базі даних. Якщо між усіма цими елементами існує ставлення простежуваності, зміна будь-якого елементу знову відкритому кість відношення для обговорення - траєкторія системи "підпадає під підозру" (кажучи мовою одного з інструментальних засобів).

Ставлення простежуваності може охоплювати багато моделей послідовних при етапів ЖЦ. Безпосередньою модифікації підлягають тільки суміжні зв'язку з простежування. Наприклад, якщо елемент А сходить до

елементу B, а елемент B сходить до елементу C, зміна, внесена в будь-який з кінцевих точок відносини, має здійснюватися в два етапи: спочатку за допомогою модифікації зв'язку A-B, а потім B-C. (Більш докладно простежуваності і управління вимогами розглядають Ріва в главі 10).

### 3.5. Бізнес-модель вимог

На етапі встановлення вимог здійснюється виявлення вимог і їх визначення, переважно, у вигляді формулювань природною мовою. Формальне моделювання вимог з використанням мови UML проводиться пізніше на етапі специфікації вимог. Проте, під час встановлення вимог постійно ведеться діяльність по узагальненому візуального уявлення зібраних вимог, звана бізнес-моделюванням вимог.

Для встановлення рамок системи потрібно, щонайменше, Високорівнева візуальна модель, що дозволяє позначити ключові прецеденти і ввести найбільш істотні бізнес-класа. На рис. 3.3 показані залежності між цими трьома моделями етапу встановлення вимог і моделі інших етапів ЖЦ розробки.

Провідна роль діаграм прецедентів в ЖЦ розробки знайшла своє відображення на рис. 3.3, з якого ясно, що джерелом тестових прецедентів, призначеної для користувача документації і проектних планів виступають моделі прецедентів. Більш того, діа грами прецедентів і моделі класів використовуються паралельно і по черзі грають роль "лідера гонки" в рамках послідовних ітерацій розробки. Проектування і реалізація також тісно переплетені і можуть ініціювати зворотний зв'язок з моделями специфікації вимог.

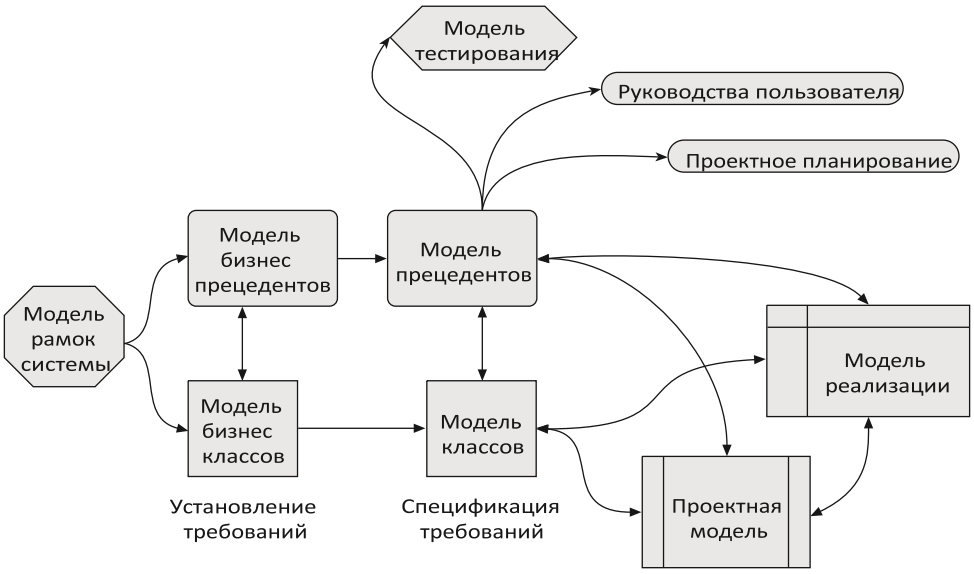



Рис. 3.3. Використання бізнес моделей на різних етапах вироблення вимог

## Лекція 4. Принципи специфікації вимог

Специфікація вимог пов'язана з докладним моделюванням вимог за замовниками, визначених у процесі встановлення вимог. При цьому рассматрива ються тільки послуги, які прагнуть отримати від системи замовники (формулювання сервісів) (разд.3.1). На етапі специфікації вимог формулювання обмежень не підлягають подальшого опрацювання, хоча і можуть зазнати змін як результат звичайного циклу ітерації.

В якості вхідної інформації процесу специфікації вимог виступають неформальні вимоги замовників, а результатом цього процесу є моді чи специфікації проектних конструкцій. Ці моделі (розд. 2.2) дають більш фор мальное визначення різних сторін (подань) системи. Зазвичай требова ня користувачів в процесі специфікації поділяються на дві основні кате горії: функціональні вимоги і вимоги до даних.

Як результат етапу специфікації виступає розширений ( "детально пророблений") документ опису вимог (розд. 3.6). Новий документ часто називають документом специфікації вимог (або просто "специфікацією" на жар гонах розробників). Структура вихідного документа не змінюється, однак содер жание значно розширюється за рахунок глав, які визначають вимоги замовлення чиков. Поступово для цілей проектування і реалізації документ специфікації вимог замінює документ опису вимог (на практиці, розширений до мент може як і раніше називатися документом опису вимог).

Моделі специфікацій можна розділити на три групи.

1. Моделі станів.

2. Моделі поведінки.

3. Моделі Зміни станів.

Моделі станів "деталізують" вимоги до даних. Моделі поведінки забезпе чивают деталізовані специфікації для функціональних вимог. Моделі зміни станів охоплюють два види вимог. Вони покликані пояснити, ка ким чином дію функцій призводить до зміни даних.

Моделі представляються у вигляді діаграм на мові візуального моделювання (візуальна модель ІНГ Language) - в нашому випадку це мова UML. Зазвичай діаграма служить цілям моделиро вання однієї зі сторін системи - станів, поведінки або зміни станів. За Метн виняток становить діаграма класів, яка визначає всі три аспекти - стан і поведінку об'єктів, і, побічно, зміни станів об'єктів.

Кожна діаграма дає уявлення про певну стороні системи. Взяті разом діаграми дають можливість розробникам і користувачам поглянути на пропоноване рішення з різних точок зору, виділяючи одні його сторони і ігнорувати руя інші. Жодна з діаграм окремо не дає повного визначення систе ми. Систему можна зрозуміти тільки через взаємопов'язаний набір діаграм.

Аналогічно нагоди інтерпретації завершених моделей конструювання діа грам - це не послідовний процес побудови однієї діаграми за одною. Діаграми розробляються в паралель, і в результаті кожної наступної Ітера ції до них додаються нові деталі. У той час, як розробники повинні следо вать строго певному процесу розробки, рішення про те, яка з моделей має відігравати роль "рушійної сили" розробки, значною мірою залежить від особистих переваг аналітика. Зазвичай діаграми прецедентів і моделі клас сов - як найбільш важливі типи моделей - конструюються паралельно, взаємно "збагачуючи" один одного ідеями.

З кожною новою итерацией розробки глибина і ступінь деталізації специфи кации зростає. Багато глибші властивості об'єктів моделі виражаються ско реї в текстовому, ніж графічному вигляді. Деякі властивості визначають задум об'єкта моделі, а не результат аналізу. Деякі інші властивості можуть відображати особливості CASEсредств.

## 4.2. специфікації станів

Стан об'єкта визначається значеннями його атрибутів і асоціацій. Напри заходів, BankAccount (об'єкт Банківський рахунок) може перебувати в стані "перевищення кредитного ліміту", якщо значення атрибута балансу (БАЛАНС) негативно кові. Оскільки стану об'єкта визначаються структурам даних, моделі структур даних називаються специфікаціями станів.

Специфікація станів дає статичний погляд на систему (тому моделиро вання станів часто називають статичним моделюванням). Тут основний за дачею є визначення класів проблемної області, їх атрибутів і відносин з іншими класами. Спочатку операції класів зазвичай не розглядаються. Вони виводи водяться з моделей специфікації поведінки.

У типовій ситуації спочатку визначаються классисущності, тобто класи, які визначають проблемну область і характеризуються постійною присутністю в базі даних системи. Подібні класи іноді називаються "бізнескласса". Класи, вича обслуговують системні події (керуючі класи) і класи, які перед становлять GUIінтерфейс (класи уявлення або прикордонні класи) НЕ встановлюється вають до тих пір, поки не стануть відомі поведінкові характеристики системи.

### 4.2.1. моделювання класів

Модель класів - це наріжний камінь розробки об'єктно ної системи. Класи лежать в основі спостережливості властивостей і поведінки систе ми. На жаль, класи завжди важко піддаються визначенню, а властивості класів не завжди очевидні. Дуже малоймовірно, щоб два аналітика прийшли до одного і того ж безлічі класів і їх властивостей для однієї і тієї ж нетривіальною проблем ної області. Хоча моделі класів можуть відрізнятися, кінцевий результат і ступінь задоволеності користувача можуть бути в рівній мірі достатніми (або в рав іншої мері недостатніми).

Моделювання класів - це не детермінований процес. Не існує про ного рецепта відшукання і визначення найкращих класів. Цей процес в значній котельної мірі носить ітеративний і покроковий нарощуваний характер. До факторів, що визначають успішне проектування класів, відноситься рівень кваліфікації і досвіду аналітика, зокрема, такі можливості.

1. Знання в області моделювання класів.

2. РОЗУМІННЯ області проблемною.

3. Досвід у сфері аналогічних і успішних проектів.

4. Здатність дивитися вперед і передбачати наслідки рішень.

5. Готовність до перегляду моделі з метою усунення недоліків і т.д.

Останній момент пов'язаний з використанням CASEсредств. Широкомасштабне застосування CASEтехнологіі може стати перешкодою на шляху розробки системи в технологічно незрілих організаціях (розд. 1.1.4.2). Однак використання ДЕЛО коштів для підвищення особистої продуктивності завжди виправдано.

4.2.1.1. виявлення класів

Два різних аналітика, як правило, не можуть прийти до ідентичним моделям клас сов для однієї і тієї ж проблемної області, і точно так само два різних аналітика не користуються одним і тим же розумовим процесом при виділенні класів. Літі ратура рясніє підходами, пропонованими для виявлення класів. Аналітики мо гут спочатку навіть слідувати одному з цих підходів, проте подальші ітерації, як правило, обов'язково призводять до використання нешаблонних і в чомусь навіть випадкових механізмів.

Барами (Бахрамі) [4] детально вивчив головні особливості чотирьох основних підходів до виявлення класів (відкриття класу). Нижче перераховані ці підходи.

1. Підхід на основі використання іменних груп.

2. Підхід на основі використання загальних шаблонів для класів.

3. Підхід на основі використання прецедентів.

4. Підхід CRC (клас-відповідальність-колабораціоністи - Клас-обязанності- "співробітники").

Барами [4] поставив у відповідність кожному з підходів опубліковані роботи, проте - на нашу мненію- тільки останній підхід має незаперечну ис джерелом. Тепер ми узагальнимо ці підходи, а потім приведемо приклади, в яких використовується комплексний підхід (змішаний підхід).

4.2.1.1.1. Підхід на основі використання іменних груп

Підхід на основі використання іменних груп (тобто іменників в предло жениях) передбачає, що аналітик читає формулювання документа опису тре мог в пошуках іменних груп. Кожне іменник розглядається як потенційний клас (кандидат клас). Потім список всіх класів поділяється на сле дмуть три групи.

1. Релевантні або відповідні класи.

2. Нечіткі або сумнівні класи.

3. Нерелевантні або невідповідні класи.

До нерелевантних (не має значення) відносяться класи, які виходять за рамки проблем ної області. Для них не вдається дати формулювання їх призначення. Досвідчені анали тікіпрактікі найімовірніше не включають невідповідні класи в початковий список потенційних класів. Таким чином вдається уникнути формальних кроків по ідентифікації і виключенню нерелевантних класів.

До релевантним (відношення) відносяться класи, які безумовно належать про проблемної області. Іменники, що представляють імена цих класів, час то зустрічаються в документі опису вимог. Крім того, значення і призначенням ня цих класів можна обґрунтувати на основі загальних знань про прикладну області, а також на основі вивчення аналогічних систем, керівництв, документів і патенто ванних програмних пакетів.

До нечітким (нечітка) відносяться класи, які не можна впевнено і беззастережно визнати придатними. Вони складають найбільшу проблему. Їх необхідно про аналізувати глибше, а потім або включити в список релевантних класів, або виключити зі списку нерелевантних. Остаточне віднесення цих класів до тієї чи іншої групи, власне, і проводить відмінність між хорошою і поганою моделлю класів.

Підхід на основі використання іменних груп передбачає наявність повного та коректного документа опису вимог. На практиці це припущення рідко відповідає дійсності. Але навіть якщо воно обгрунтовано, копітка изуче ня великих обсягів тексту не обов'язково має привести до отримання исчерп вающего і точного результату.

4.2.1.1.2. Підхід на основі використання загальних шаблонів для класів

Підхід на основі використання загальних шаблонів для класів дозволяє вивести потен соціальні класи на основі теорії родової класифікації об'єктів. Теорія класси фікації стосується поділу світу об'єктів на зручні групи, що дозволяє більш ефективно будувати міркування про них.

Барами [4] наводить такий перелік груп (шаблонів) для виявлення по потенційних класів.

- Понятійний (або концептуальний) клас (поняття клас) являє собою ідею, яку розділяє або з якої згодна значна спільність людей. За відсутності понять люди не здатні ефективно спілкуватися або навіть спілкуватися на якомусь задовільному рівні. Наприклад, Бронювання (Резервування) - це понятійний клас, що відноситься до системи резервування місць в авіакомпаніях.
- Событійний клас (клас подій). Подія - це щось, що не вимагає часу стосовно нашої часовій шкалі. Наприклад, Прибуття (Прибуття) - це подієвий клас, що відноситься до системи резервування місць в авіакомпаніях.
- Організаціонний клас (організація класу). Організація - це будь-який вид цілеспрямованого об'єднання або сукупності сутностей. Наприклад, турагент (Бюро подорожей) - це клас, що відноситься до системи резервування місць в авіакомпаніях.
- Класс "людей" (люди, клас). Під "людьми" тут розуміється скоріше роль, яку людина відіграє в тій чи іншій системі, а не фізична особа. Наприклад, Пасажир (Пасажир) - це клас, що відноситься до системи резервування місць в авіакомпаніях.
- Класс розташування (місця класу). Місцезнаходження визначає фізичне розташування об'єктів, пов'язаних з інформаційною системою. Наприклад, TravelOffice (Офіс бюро подорожей) - подібний клас, що відноситься до системи резервування місць в авіакомпаніях.
- Дж. Рамбана (J. Рамбо), А. Джекобсон (I. Jacobson) і Г. Буч (Г. Буч) [76] пропонують іншу схему класифікації.
- Фізіческій клас (фізичний клас) (наприклад, літак (Літак)).
- Бізнескласс (бізнес-клас) (наприклад, резервування).
- Логіческій клас (логічний клас) (наприклад, FlightTimetable (Розклад рейсів)).
- Прікладной клас (клас додатки) (наприклад, ReservationTransaction (Операція резервування)).
- Компьютерний клас (комп'ютерний клас) (наприклад, Index (Індекс)).
- Поведенческій клас (поведінковий клас) (наприклад, ReservationCancellation (Скасування резервування)).

Підхід на основі використання загальних шаблонів класів служить в якості поліз ного керівництва, але не визначає систематичного процесу, за допомогою якого можна було б виділити надійне і повне безліч класів. Цей підхід можна з успіхом використовувати для визначення початкового безлічі класів або для перев ки того, чи повинні деякі класи (отримані іншими способами) бути присутнім в нашому безлічі або, навпаки, бути відсутніми в ньому. Однак, підхід на основі ис користування загальних шаблонів класів занадто слабо пов'язаний зі специфічними требо ваниями користувачів, щоб претендувати на вичерпне рішення.

Особлива небезпека, пов'язана з підходом на основі використання загальних шаблонів класів, полягає в неправильному тлумаченні імен класів.

Наприклад, що означає Прибуття (Прибуття)? Чи означає це прибуття на взлетнопосадочную смугу (час приземлення), прибуття до терміналу (час висадки), прибуття в зал повернення багажу (час митного огляду) і т.д.? Аналогічно, слово Бронювання (в даному слу чаї резервація. Прим. Ред.) У середовищі північноамериканських індіанців має зовсім інше значення в порівнянні з тим, що малося на увазі досі.

4.2.1.1.3. Підхід на основі використання прецедентів

Підходу на основі використання прецедентів надається особливе значення в мові UML. Можна навіть сказати, що цей підхід рекомендується використовувати в рамках UML (якщо бути точним - то в рамках методології RUP (Rational Unified Process)). Графічна модель прецедентів супроводжується неформальними описами, а також діаграма ми послідовностей і кооперації для окремих прецедентів (розд. 2.2 і далі в цьому розділі). Ці додаткові описи і кроки визначення діаграм (і об'єктів) потрібно виконати для кожного прецеденту. На основі цієї інформації можна прийти до узагальнень, необхідним для виявлення потенційних класів.

Підхід, що направляється прецедентами, володіє особливостями, притаманними під ходу снізувверх. Після того, як прецеденти стають відомі, а уявлення про систему з точки зору взаємодії, щонайменше, частково визначено з по міццю діаграм послідовностей, об'єкти, що використовуються в цих діаграмах, призводять до виявлення класів.

Насправді цей підхід в чомусь схожий на підхід, який використовує іменні групи. Їх об'єднує те, що прецеденти специфікують вимоги. Обидва підходи спрямовані на вивчення формулювань, викладених в документі опису вимог, щоб виявити в результаті потенційні класи. Те, що ці формулювання викладаються в оповідної формі або представлені графічно, має второ статечне значення. У будь-якому випадку на цьому етапі ЖЦ розробки ПЗ більшу частину прецедентів можна описати тільки в текстовій формі без діаграм взаємодії.

Підхід, заснований на прецедентах, має ті ж вади, що й під хід, який використовує іменні групи. Будучи по суті підходом снізувверх в сенсі точності, він спирається на повноту і коректність моделей прецедентів. У результа ті, він навіть може привести до небажаного розбалансування ітеративного і нарощуємо процесу розробки ПО, при якому моделі прецедентів повинні бути завершені ще до побудови моделей класів. Загалом, хоч би якими були цілі і засоби, це призводить до функціонального підходу (functiondriven підхід) (прихильники об'єктно підходу вважають за краще називати його про блемноорієнтірованним (problemdriven)).

4.2.1.1.4. CRC Підхід

CRC Підхід (Клас-Відповідальність-Співавтори - клас-відповідальність-"співро ники") являє собою щось більше, ніж метод виявлення класів, - це при привабливості спосіб інтерпретації та вивчення об'єктів (а також і навчання об'єк проектних підходу). Найбільшу популярність підхід CRC отримав завдяки роботам Ребекки ВірфсБрок (Ребекка WirfsBrock) і її колег Б. Вилкерсон (Б. Вилкерсон) і Л. Вінер (L. Wiener) [94], [93].

CRC Підхід включає в себе сеанси "мозкового штурму", проведення яких обліг чає за рахунок використання спеціально підготовлених карток. Картки складаються з трьох відділень: ім'я класу записується в верхньому відділенні, "обов'язки" класу пере чисельні в лівому відділенні, а "співробітники" перераховані в правому відділенні. Зобов'язане сти - це послуги (операції), які клас готовий виконати в інтересах інших класів. Для виконання багатьох обов'язків необхідна участь (обслуговування) з боку інших класів. Такі класи перераховуються як "співробітники".

CRC Процес - це живий процес, під час якого розробники "грають в карти", вони заповнюють картки іменами класів і призначають їм "обов'язки" і "співробітників" в ході виконання сценарію обробки інформації (наприклад, сце Нарія прецеденту). У тих випадках, коли виникає потреба в якійсь послузі, а су Існуючі класи не покривають її, створюється новий клас, якому призначаються відповідні "обов'язки" і "співробітники". Якщо клас стає "занадто за прийнятим", він поділяється на кілька менших класів.

CRC Підхід відрізняється від інших підходів тим, що при його використанні виокрем лення класів є результатом аналізу повідомлень, переданих між об'єктивним тами для виконання завдань обробки інформації. Акцент робиться на уніфікує ванном методі розподілу "інтелекту" в системі, і деякі класи можуть бути скоріше отримані, виходячи з подібної технічної потреби, ніж виявлені в якості "бізнесоб'ектов" як таких. У цьому сенсі метод CRC може бути по над прийнятним для перевірки правильності вибору класів уже виявлених з по міццю інших методів. CRC Підхід також корисний при встановленні властивостей класів (які логічно випливають з "обов'язків" і типів "співробітників" Класу).

4.2.1.1.5. комплексний підхід

На практиці процес виявлення класів в різний час, найімовірніше, слід різним підходам. Найчастіше, він включає елементи всіх чотирьох підходів, рассмот корінних вище. Важливими чинниками при цьому виступають загальна ерудиція екс перта, його досвід і інтуїція. Процес в чистому вигляді не слід ні методу сверхувніз, ні методу снізувверх - він весь час йде "з середини". Подібний підхід до виявле ню класів ми називаємо комплексним підходом (змішаний підхід).

Ось один з можливих сценаріїв. Початкове безліч класів можна сформи вати на основі загальних знань і досвіду експерта. При цьому додатково можна керуватися підходом на основі загальних шаблонів класів. Решта класи можна

додати, грунтуючись на аналізі узагальненого опису проблемної облас ти з використанням підходу на основі іменних груп. Якщо в розпорядженні анали тика є прецеденти, можна скористатися підходом, які направляються Прец дентами, щоб додати нові і перевірити спроможність існуючих класів. Нарешті, CRC підхід дозволяє застосувати "мозковий штурм" для перевірки пра ності вибору виділеного до цього часу безлічі класів.

4.2.1.1.6. Деякі правила виявлення класів

Нижче наведено далеко не повний перелік керівних принципів або правил менту, котрим повинен слідувати аналітик при виборі потенційних класів. Знову напір міна про те, що тут ми маємо справу лише з классамісущностямі.

1. Для кожного класу має бути ясно сформульовано його призначення в системі.

2. Кожен клас - це шаблон опису безлічі об'єктів. Поодинокі класи, для яких можна уявити існування тільки одного об'єкта, досить малоймовірні серед "бізнесоб'ектов". Подібні класи зазвичай складають в додатку "загальне знання" і як правило жорстко запрограмовані в програмах додатки. Наприклад, якщо система спроектована для единст кої організації, існування класу Організація (Організація) мо же бути не виправдано.

3. Кожен клас (тобто класссущность) повинен містити набір атрибутів. Хоро шим прийомом є встановлення ідентифікують атрибутів (ключів), щоб допомогти нам судити про потужність (потужність) класу (тобто очікуваний коли честве об'єктів даного класу в базі даних). Слід, однак, пам'ятати про те, що клас не обов'язково повинен володіти призначеним для користувача ключем. Об'єкти класів ідентифікуються за допомогою ідентифікаторів об'єктів

(OID) (розд. 2.1.1.3).

4. Кожен клас повинен відрізнятися від атрибута. Чи подається поняття класом або атрибутом залежить від галузі застосування. Колір автомобіля зазвичай сприймається як атрибут класу Car (Автомобіль). Однак на фаб рике з виробництва фарб Color (Колір) - це безперечно клас зі своїми власними атрибутами (яскравістю, насиченістю, прозорістю і т.д.).

5. Кожен клас містить набір операцій. Однак на даному етапі ми не торкаємося питань ідентифікації операцій. Операції, що входять в інтерфейс класу (послуги, що надаються класом системі), є логічним наслідком їм формулювання призначення класу (пункт 1).

4.2.1.1.7. Приклади виявлення класів

Давайте проаналізуємо вимоги з метою виділення потенційних класів. У першому затвердження відповідними класами є класи Ступінь (Ступінь) і курс (Курс). Ці два класи задовольняють п'яти правилам, перерахованих раніше. Ми поки не впевнені, чи повинен і яким чином клас Курс бути звужений до класів CompulsoryCourse (Обов'язковий курс) і ElectiveCourse (Вибірковий курс). На

приклад, ясно, що курс є обов'язковим або вибірковим залежно від сте пені. Можливо, що відмінність між обов'язковими і вибірковими курсами може бути зафіксовано за допомогою асоціації або навіть атрибута класу. Таким чином,

CompulsoryCourse і ElectiveCourse розглядаються як нечіткі класи.

Друге твердження ідентифікує тільки атрибути класу для гольфу, а саме course_level (рівень курсу) і credit_point_value (кількість умовних оч ков). Третє твердження характеризує асоціацію між класами Курс і

Ступінь. Четверта формулювання вводить атрибут min_total_credit_points (мінімальна загальна кількість умовних очок) в якості атрибута класу ступеня.

Останнє твердження дозволяє нам виділити три нових класи: Студент (Студент), CourseOffering (Пропонований курс) і StudyProgram (Програма обу чення). Перші два, безумовно, є релевантними класами, ось StudyProgram а можна перетворити в асоціацію між класами Студент і CourseOffering. Тому StudyProgram класифікується як нечіткий клас. Наші міркування відображені в табл. 4.1.

Таблица 4.1. Потенциальные классы (Запись на университетские курсы)

| ***Релевантные классы*** | ***Нечеткие классы*** |
|---|---|
| Course (Курс) | CompulsoryCourse (Обязательный курс) |
| Degree (Степень) | и ElectiveCourse (Выборочный курс) |
| Student (Студент) | StudyProgram (Программа обучения) |
| CourseOffering (Предлагаемый курс) | |

Перше твердження містить кілька іменників, але тільки недо торие з них можна перетворити в потенційні класи. "Відеомагазинів" - це не клас, оскільки він становить всю систему (в базі даних може бути тільки один об'єкт цього класу - так званий поодинокий клас). Аналогічно, поняття "запасу" і "відеотеки" занадто загальні, щоб розглядати їх як класів, по край ній мері на цьому етапі. Релевантними класами представляються MovieTitle (Назва фільму), відеокасетах (Відеокасета) і відеодиск (Відеодиск).

Друге твердження вводить додаткову спеціалізацію відеопрокату як про ката відеокасет і дисків. Ми можемо запропонувати три нових класи: BetaTape, VHSTape і DVDDisk. Однак оскільки ми маємо справу лише з одним типом відео дисків, в кінцевій моделі можна не залишати обидва класи: відеодісковая і DVDDisk. Який з двох класів залишити, залежить від кінцевого рівня спеціалізації примі нительно до відеокасет і дисків. Зауважимо, що ми поки не впевнені в тому, чи будуть класи BetaTape і VHSTape володіти какимилибо відмітними атрибутами (за винятком того факту, що один з них належить типу Beta, а інший - VHS).

Третє твердження говорить про те, що кожне найменування фільму відрізняється умовами прокату, пов'язаними з ним. Однак не ясно, що мається на увазі під "фільмом" - найменування фільму або ж носій фільму (касета або

диск)? Нам необхідно прояснити цю вимогу з замовниками. Тим часом, ми можемо віддати перевагу оголосити клас RentalConditions (Умови прокату) як нечіткий, замість того, щоб зберігати інформацію про період прокату і плати за прокат в клас се для найменування фільму або для носія фільму.

Останнє формулювання переконує нас в тому, що класи BetaTape, VHSTape і DVDDisk (або відеодиск) - релевантні. Нам потрібно зберігати інформацію про ті кущіх умовах для кожної касети і диска. Однак атрибути на зразок video_condition (умови відеопрокату) або number_currently_available (кількість, яка є в наявності) можна в загальному оголосити в абстрактному класі верхнього рівня (назвемо його VideoMedium (відеоносіях)), після чого вони можуть бути успадковані конкретним подклассом (таким як VHSTape). Підсумки наших рас суджень відображені в табл. 4.2.

Таблиця 4.2. Потенциальные классы (Магазин видеопроката)

| ***Релевантные классы*** | ***Нечеткие классы*** |
|---|---|
| MovieTitle (Название фильма)<br>VideoMedium (Видеоноситель)<br>VideoTape (Видеокассета)<br>VideoDisk (или DVDDisk)<br>BetaTape<br>VHSTape | RentalConditions (Условия проката) |

Перше твердження містить поняття "клієнт", "контракт" і "товар". Наша про щая ерудиція і досвід підказують нам, що це типові класи. Однак поняття контракту і товару не входять в рамки системи Управління контактами з клієнтами і повинні бути відкинуті.

Замовник (Клієнт) - це релевантний клас, проте ми можемо віддати перевагу на кликати його Контакт (Контакт), маючи на увазі, що не всі контакти здійснюються з на шими справжніми клієнтами. Різниця між готівковим і потенційним клієнтом може виправдати або не виправдати введення таких класів як CurrentCustomer (Готівковий клієнт) і ProspectiveCustomer (Потенційний клієнт). Оскільки впевненості у нас немає, ми оголошуємо ці класи нечіткими.

Друге твердження проливає нове світло на наведені вище міркування. Нам необхідно провести відмінність між контактної організацією і контактною особою. Ім'я клієнта не здається занадто зручним для назви класу. Крім іншого, Замовник поняття має на увазі тільки готівкового клієнта і, крім того, це може містити в собі як поняття організації, так і контактної особи. Наша нова пропозиція полягає в тому, щоб назвати класи наступним чином: Організація (Організація), Contact (Контакт) (мається на увазі контактна ліцо), CurrentOrg (Готівкова організація) (тобто організація, яка є нашим на особистим клієнтом) і ProspectiveOrg ( потенційна організація) (тобто організує ція, що є нашим потенційним клієнтом).

У другому затвердження згадується кілька атрибутів класів. Однак звичайні ний і кур'єрський поштові адреси являють собою складові атрибути і при Меним до обох класів: Організація і контакт. Тому PostalAddress і CourierAddress - законні нечіткі класи.

Таблица 4.3. Потенциальные классы (Управление контактами с клиентами)

| ***Релевантные классы*** | ***Нечеткие классы*** |
|---|---|
| Organization (Организация) | CurrentOrg |
| Contact (Контакт) | ProspectiveOrg |
| Employee (Сотрудник) | PostalAddress |
| Task (Задание) | CourierAddress |
| Event (Мероприятие) | |

У третій формулюванні вводиться три релевантних класу Співробітник (Співробітник), Task (завдання) і Event (Захід). Ця пропозиція пояснює суть планової діяльності.

Останнє твердження присвячено подальшому проясненню сенсу і взаимоот відносин між класами, однак тут не вводиться жодного нового класу.

4.2.1.2. специфікація класів

Після того, як перелік потенційних класів сформований, необхідна їх подальша специфікація: класи потрібно включити в діаграму класів і визна ділити їх властивості. Деякі властивості можна ввести і відобразити всередині графічні ських піктограм, що представляють класи на діаграмі класів. Багато інші властивості, включені в специфікацію класу, мають тільки текстове представле ня. CASEсредства, як правило, володіють можливостями редагування, позво ляющая легко вводити або модифікувати подібну інформацію за допомогою діалогових вікон, забезпечених вкладками, або за допомогою аналогічних способів.

Як пояснювалося на початку цієї глави, класи задаються на певному рівні абстракції. Більш розвинені можливості моделювання мови UML тут не ис користуються - вони розглядаються в розділі 5 і наступних розділах.

4.2.1.2.1. іменування класів

Кожному класу необхідно присвоїти ім'я. При роботі з деякими ВИПАДОК засобами крім імені класу можна також привласнити код, можливо, відмінний від імені. Код може задовольняти угодам по іменування, необхідним целе вим мовою програмування або СУБД. Для генерації програмного коду з проектної моделі використовується саме код класу, а не ім'я.

Мимохідь, ми прийняли певна угода щодо імен класів. Ця угода укладається в тому, що ім'я класу починається з великої літери. Для складу них імен в якості першої літери кожного слова також використовується велика літера (замість того, щоб відокремлювати слова знаком підкреслення або

дефісом). Це всього лише реко вані угоду, проте серед розробників воно знайшло досить багато при верженцев. (В розд. 6.1.3.2 до імені кожного класу рекомендується додати однобуквен ний префікс для позначення програмного шару, до якого належить клас).

Ім'я класу має бути іменником в однині (наприклад, курс) або, при можливості, поєднанням прикметника і сущест вітельнимі в однині (наприклад, CompulsoryCourse). Ясно, що клас є шаблон для безлічі об'єктів і використання в якості іменників у множині не несе ніякої додаткової ін формації. Іноді іменник в однині не відображає справжнього призначення класу. У подібних ситуаціях допустимо використання існуючих тільних у множині (наприклад, RentalConditions (Умови прокату) в прикладі 4.2).

Ім'я класу має бути осмисленим. Вона повинна відбивати справжню природу класу. Воно повинно запозичувати з словника користувачів (а не жаргону розробників).

Краще використовувати довші імена, ніж приховувати їх сенс за шифром. Можна з упевненістю сказати, що імена довжиною понад тридцять символів занадто громіздкі (а деякі програмні середовища їх просто не сприймають, якщо ДЕЛО кошти працюють з іменами класів, а не кодами класів). Можливо також використання довших описових імен на додаток до імен та кодами класів.

4.2.1.2.2. Виявлення та специфікація атрибутів класів

Графічна піктограма, що представляє клас, складається з трьох відділень (ім'я класу, атрибути, операції) (розд. 2.1.2). Специфікація атрибутів класів належить до специфікації станів і розглядається в цьому розділі. Специфічні кация операцій розглядається пізніше в цьому розділі в підрозділі, присвяченому специфи кации поведінки (розд. 4.3).

Виділення атрибутів здійснюється паралельно з виділенням класів. Ідентифікації атрибутів свого роду "побічний ефект" встановлення класів. Це не означає, що виявлення атрибутів - просте завдання. Навпаки, це процес, требующій значних зусиль і багаторазових ітерацій.

Вихідні моделі специфікації визначають тільки атрибути, які є существ веннимі для розуміння станів, в яких можуть знаходитися об'єкти класу. Ос ментальні атрибути можна до пори до часу ігнорувати (проте аналітик повинен бути впевнений в тому, що встановлена, але проігнорована на певному етапі інформація не буде помилково загублена і буде зафіксована згодом). Мало ймовірно, щоб всі атрибути класу були приведені в документі опису охоронних вимог, однак важливо не включати в специфікацію ті атрибути, які не випливають з вимог. У наступних ітераціях можна додати більше атрибутів.

Для імен атрибутів ми рекомендуємо дотримуватися простого угоди: в іменах атрибутів використовувати тільки малі літери, а слова в складових іменах від делять підкресленням.

4.2.1.2.3. Приклади специфікації класів

У першому затвердження згадуються конфлікти розкладу, однак ми не знаємо дос товерно, як слід моделювати цю проблему. Можливо, що мова тут йде про Прец денте, який процедурно визначає конфлікти розкладу. Другу частину цієї ж формулювання можна змоделювати за рахунок ведення атрибута enrolment_quota (квота набору) в клас CourseOffering. Тепер також ясно, що клас повинен мати Атрібі тами рік (рік) і семестр (семестр).

Друге формулювання зміцнює нас на думці про необхідність введення класу StudyProgram. Можна бачити, що клас StudyProgram поєднує в собі ряд дисцип лін, пропонованих до вивчення в поточний момент. Тому клас StudyProgram як і повинен мати атрибутами рік і семестр.

Найближчий розгляд нечітких класів CompulsoryCourse і

ElectiveCourse призводить нас до висновку, що навчальний курс є обов'язковим або вибірковим щодо певної наукового ступеня. Один і той же курс мо же бути обов'язковим по відношенню до одного ступеня, вибірковим, що стосується іншого, і взагалі неприпустимим стосовно деяким іншим ступенями. Раз так, то CompulsoryCourse і ElectiveCourse не є класами в повному смислі ле слова. (Зауважимо, що тут ми не торкаємося область моделювання класів з ис користуванням узагальнення (2.1.5 розд.) - Моделювання узагальнення розглядається в розділі 4.2.4.)

На рис. 4.1 представлена модель класів, відповідна проведеним нами міркувань. Крім того, на малюнку використовуються символи (стереотипи (стереотипу)) << PK >> і << >> CK для позначення первинних ключів і потенційних ключів, з відповідально (розд. 8.4.1.2). Це унікальні ідентифікатори об'єктів для рассмот корінних класів. Тут же задані типи даних для атрибутів.

Класи StudyProgram і CourseOffering не мають поки що ідентифікують атрибутів. Вони будуть введені в ці класи після встановлення асоціативних зв'язків між класами (розд. 2.1.3 і 4.2.2).

Перше твердження роз'яснює, що умови прокату відрізняються для касети (відеокасетах) і диска (відеодиск), що містять один і той же фільм. Тому має сенс ввести окремий клас RentalConditions (який повинен асоціювати ся з класами відеоплівку і відеодиск).

З формулювання другого вимоги випливає необхідність співіснування обох класів: відеодісковая і DVDDisk. Ієрархія узагальнення з коренем VideoMedium тепер стає досить очевидною, однак ми відкладаємо обговорення ставлення ня узагальнення до розділу 4.2.4.

На підставі аналізу третього пропозиції ми вводимо в клас MovieTitle ат рібути коду фільму movie_code (як ключового атрибута) і режисера - режисер. Решта атрибути були розглянуті в прикладі 4.2.

На рис. 4.2 показана модель класів для додатка Магазин відеопрокату, по строєна в результаті обговорення вимог прикладів 4.2 і 4.3. VideoMedium.number_currently_available Атрибут є атрибутом з областю дейст віякласс (статичним атрибутом) (розд. 2.1.6). MovieTitle.is_in_stock Атрибут, що відображає наявність в запасі фільму з даними назвою, є похідним (отримані) атрибутом, тобто він може бути обчислений на основі наявного в наявності ті кущего кількості фільмів - VideoMedium.number_currently_available.

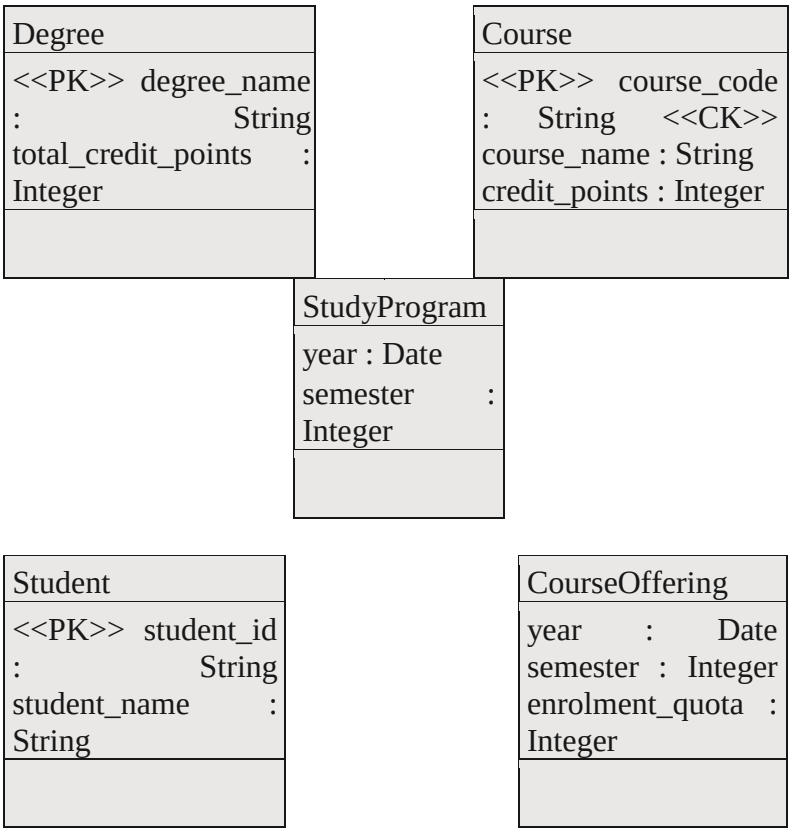
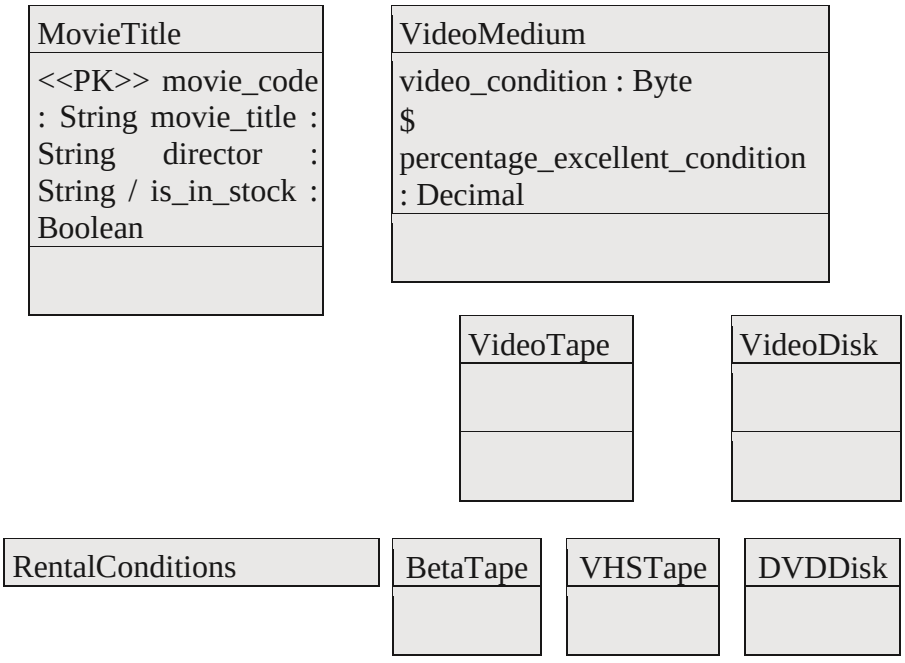



*Рис. 4.1. Спецификация классов (Запись на университетские курсы)*



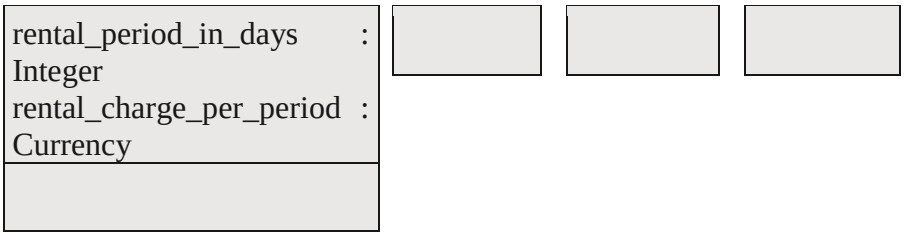



*Рис. 4.2. Спецификация классов (Магазин видеопроката)*

Аналіз першого твердження говорить нам про те, що поняття готівкового клієнта ви водиться на основі асоціації між класами організації і контракту. Ця асоціацію -ціативності зв'язок може бути досить динамічною. Отже, потреба в класах CurrentOrg і ProspectiveOrg відпадає. Більш того, наша система не стосується управ ління контрактами і не відповідає за підтримку класу контракту. Кращий варіант, до торий проглядається в даному випадку - змоделювати рішення за допомогою вироб водного атрибута Organization.is_current, який позначає готівкову органи зацию і при необхідності може змінюватися підсистемою управління контрактами.

Друге твердження наводить на думку про необхідність запровадження двох класів для адрес: PostalAddress і CourierAddress.

Решта формулювання вимог дають додаткову інформацію про зі триманні атрибутів класів. Крім того, тут можна висловити кілька сообра жень, пов'язаних з асоціативними відносинами та обмеженнями цілісності (вони будуть розглянуті пізніше).

Специфікація класів для додатка Управління контактами з клієнтами перед ставлена на рис. 4.3. Як видно з малюнка, відносини між класами в моделі від сутні. Тому, наприклад, восьме твердження, яке пов'язує співробітників із завданнями і заходами, не знайшло відображення в моделі.

Перша формулювання дозволяє нам встановити кілька атрибутів класу

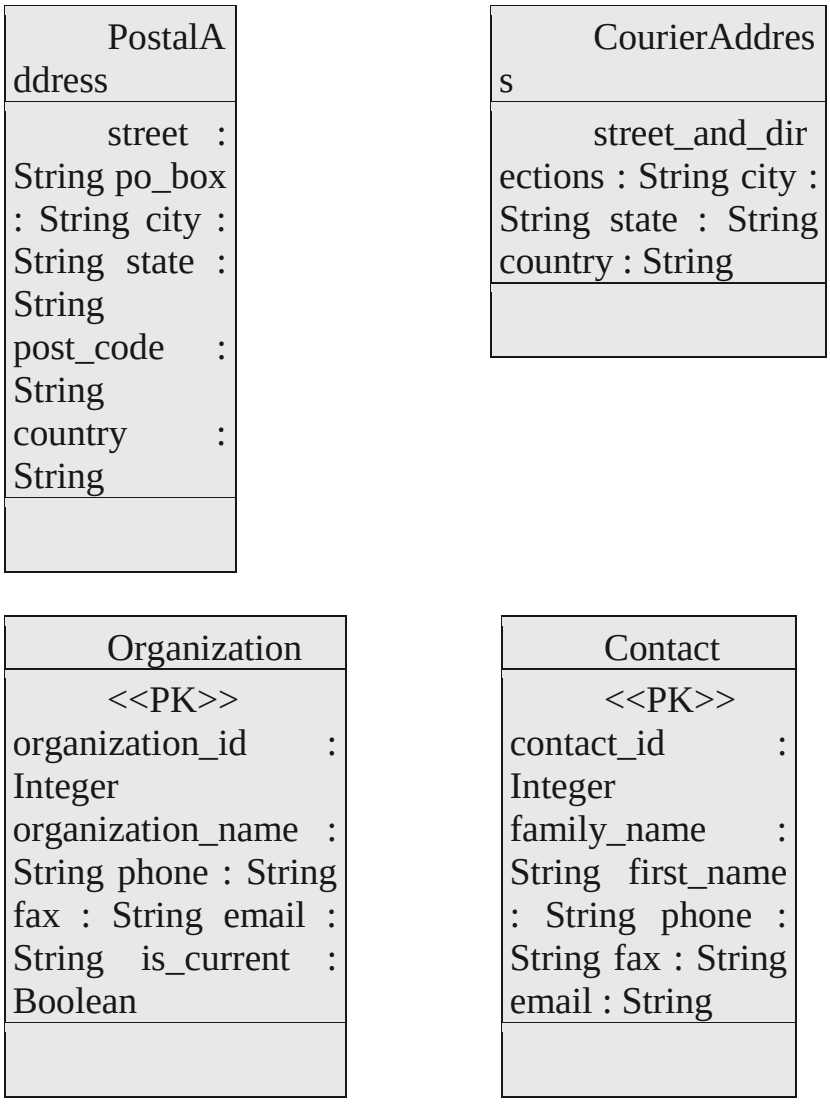
Кампанія. Клас кампанії містить наступні атрибути CAMPAIGN_CODE (код кам панії) (первинний ключ), campaign_title (назва кампанії), DATE_START (дата початку) і date_closed (дата закриття).

Остання пропозиція затвердження 1 стосується призів кампанії. При його бли жайшее розгляді можна прийти до висновку про те, що премія (Приз) - самостоя вальний клас: приз розігрується, йому притаманна така властивість як переможець, і він повинен володіти іншими явно не вираженими властивостями, такими як опис, цінність і місце в ряду інших призів кампанії.

Ми вводимо клас Приз в модель класів разом з атрибутами, про які говорилося вище: prize_descr, prize_value і prize_ranking. Ми помічаємо, що дата троянди Гриша призів збігається для всіх призів кампанії. Ми вводимо атрибут дати розигри ша date_drawn в клас кампанії. Приз виграє благодійник. Ми можемо зафік сіровать цей факт пізніше в зв'язку з введенням асоціації класів і премії Supporter.

2 Формулювання говорить, що у кожного квитка є номер, однак цей номер не унікальний серед усіх квитків (він унікальний тільки в рамках кампанії). Ми вводимо ат рібут ticket_number, однак, не надаємо йому статус первинного ключа для класу CampaignTicket. Два інших атрибута CampaignTicket представляють ціну квитка (ticket_value) і статус квитка (ticket_status). Загальна кількість квитків і до лічество проданих квитків - КЛАСУ кампанії Атрибути (num_tickets і num_tickets_sold).

3 Затвердження виявляє кілька відкритих питань. Які структури дан них потрібні нам для розрахунку продуктивності Телемаркетер? Щоб відповісти на це питання нам потрібно визначити деякі показники, за допомогою яких можна висловити продуктивність. Один з можливих варіантів полягає в тому, щоб підраховувати середня кількість дзвінків на годину і середня кількість успішних дзвін ков на годину. Потім можна обчислити показник продуктивності, розділивши кількість вус пешню дзвінків на загальну кількість дзвінків. Тепер введемо в клас Телепродавец відповідні атрибути: average_per_hour і success_per_hour.



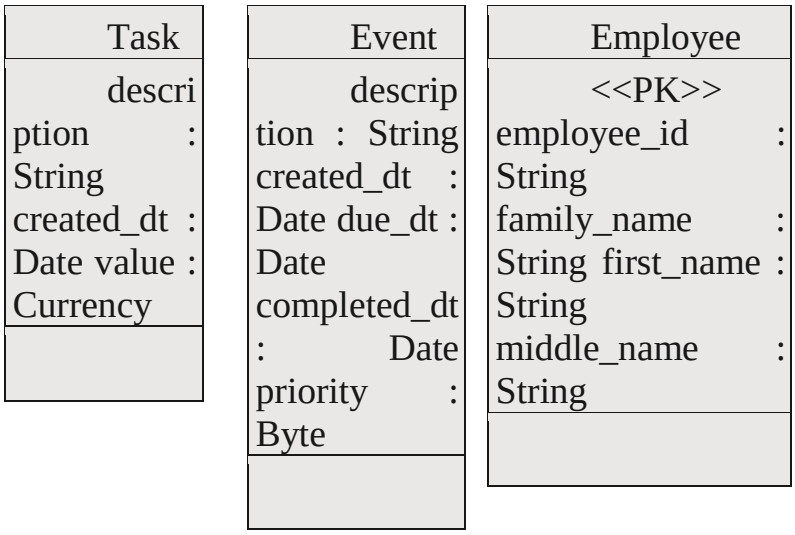



*Рис. 4.3. Спецификация классов (Управление контактами с клиентами)*

Для обчислення показників продуктивності Телемаркетер необхідно запоми нать тривалість кожного дзвінка. Сховищем для цієї інформації служить клас CallOutcome (Результат дзвінка). У цей клас ми вводимо атрибути для початку і закриття кампанії: start_time і END_TIME. Ми припускаємо, що результат каж дого дзвінка пов'язаний Телемаркетер допомогою асоціації.

Результатом аналізу затвердження 4 є включення ряду атрибутів в клас Прихильник. Ось ці атрибути: supporter_id (первинний ключ),. supporter_name, phone_number, mailing_address, date_first, date_last, campaign_count, preferred_hours і credit_card_attributes. Деякі з цих атрибутів (mailing_address, preferred_hours) досить складні для того, щоб превра тить їх в додаткові класи пізніше в процесі розробки. Поки ж ми зберігаємо їх як атрибути.

5 Затвердження стосується класу CallScheduled (Запланований дзвінок). Ми вводимо в нього атрибути phone_number, пріоритет і attempt_number. У нас немає підлогу ного розуміння того, як підтримувати вимогу про те, що наступні дзвінки повинні проводиться в різний час дня. Очевидно, що це повинно бути покладено на алгоритм планування, однак нам потрібна підтримка структур даних. До сча стю, певне світло на цю проблему проливає наступне формулювання.

Результатом аналізу затвердження 6 є введення класу CallType (Тип дзвін ка). Цей клас містить наступні атрибути: type_descr, call_attempt_limit, а також alternate_hours. Останній атрибут - це складна структура даних, анало гічно атрибуту preferred_hours Прихильник класу, яка врешті-решт стане окремим класом.

Останнє твердження класифікує результати дзвінків. Це пряма вказівка на необхідність введення відповідного класу - OutcomeType. Можливі типи результату можуть зберігатися в атрибуті outcome_type_descr. Не ясно, які ще атри пляшок можна включити в OutcomeType, проте ми переконані, що у міру вивчення де талізірованних вимог, ці атрибути будуть встановлені. Одним з них

може бути атрибут follow_up_action, призначення якого - зберігати інформацію про ти пічних наступні кроки стосовно кожного типу результату.

На рис. 4.4 представлена модель класів, завершальна наведені вище рассу ждения. Асоціативні відносини, вже встановлені в моделі бізнескласса (рис. 3.6), збережені. Нові асоціацію не введені.

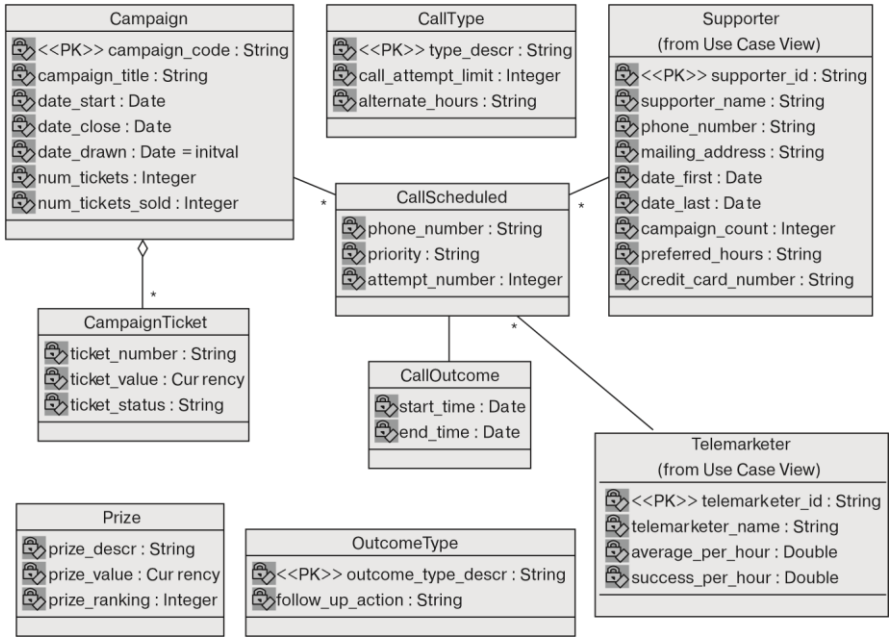



***Рис. 4.4. Спецификация классов (Телемаркетинг)***

4.2.2. моделювання асоціацій

Асоціації служать об'єднанню об'єктів в системі. Вони сприяють взаємо дії між об'єктами. Без асоціацій об'єкти можуть встановлювати зв'язок тіль ки під час прогону програми, якщо вони спільно використовують одні і ті ж Атрібі ти або вони мають доступ (за допомогою інших засобів, таких як глобальні змін ні) до ідентифікує об'єкти значеннями інших об'єктів.

Асоціації представляють собою найбільш істотний вид відносин моделей, зокрема, моделей постійних "бізнесоб'єктов". Асоціації підтримують ви конання прецедентів і, таким чином, забезпечують поєднання специфікації станів і поведінки.

4.2.2.1. виявлення асоціацій

Знаходження основних асоціацій є побічний ефект процес са виявлення класів. При визначенні класів аналітик приймає рішення про ат рібутах класів, і деякі з цих атрибутів є асоціаціями з іншими класами. Атрибути можуть ставитися до елементарних типів даних або можуть вво диться в якості інших класів, встановлюючи таким чином відносини з іншими класами. По суті, будь-який атрибут, що відноситься до Неелементарні типам даних, повинен моделюватися як асоціація (або агрегація) по класу, який представляє цей тип даних.

Виконання пробного прогону прецедентів дозволяє виявити залишаються асоціації цій. Встановлюються шляхи взаємодії між класами, необхідні для прогону прецедентів. Зазвичай асоціації повинні підтримувати ці шляхи взаємодії.

Кожна тернарного асоціація повинна бути замінена циклом або бінарної асоціації цією. Тернарние асоціації привносять ризик невірного семантичного тлумачення.

Іноді для того, щоб повністю виразити базову семантику, цикли, утворені асоціаціями, не повинні комутувати (бути замкнутими) [51]. Це означає, що щонайменше одна з асоціацій в циклі може бути похідною (похідний). Подібна асоціація є надлишкової в семантичному сенсі і повинна бути виключена (хороша семантична модель повинна бути позбавлена надмірності). Цілком допус Тімо, що багато похідні асоціації, тим не менш, все ж увійдуть в проектну модель (наприклад, з міркувань ефективності). 4.2.2.2. специфікація асоціацій

Специфікація асоціацій має на увазі виконання наступних дій.

1. Привласнення Імен асоціаціям.

2. Привласнення імен асоціативним ролям.

3. Встановлення кратності асоціації (розд. 2.1.3.2).

Правила іменування асоціацій повинні відповідати угодам по име нованіе атрибутів - імена асоціацій складаються з малих літер, окремі слова в імені асоціації розділяються підкресленням (розд 4.2.1.2.2.).

Якщо два класи пов'язані лише одним асоціативним відношенням, задавати ім'я асоціації та асоціативні рольові імена між цими класами необов'язково (розд. 2.1.2.1.1). CASEсредства можуть внутрішньо розрізняти кожну асоціацію через системні ідентифікаційні імена.

Рольові імена можна використовувати для розкриття більш складних асоціацій, в ча стности самоассоціативних відносин (самостійні об'єднання) (рекурсивних асоціацій, кото риє пов'язують об'єкти одного і того ж класу). При завданні рольових імен їх наслідком дме вибирати з урахуванням того, що в проектній моделі вони стануть атрибутами класів, розташованих на протилежних кінцях асоціативного зв'язку.

Кратність повинна бути задана для обох кінців (ролей) асоціації. Якщо питання кратності на цьому етапі не ясний, нижня і верхня межі кратності можна опустити.

### *4.2.2.3. Пример спецификации ассоциации*

Модель ассоциаций для приложения *Управление контактами с клиентами* показана на рис. 4.5. Чтобы продемонстрировать гибкость ассоциативного моделирования, имена ассоциаций и ролевые имена используются бессистемно.

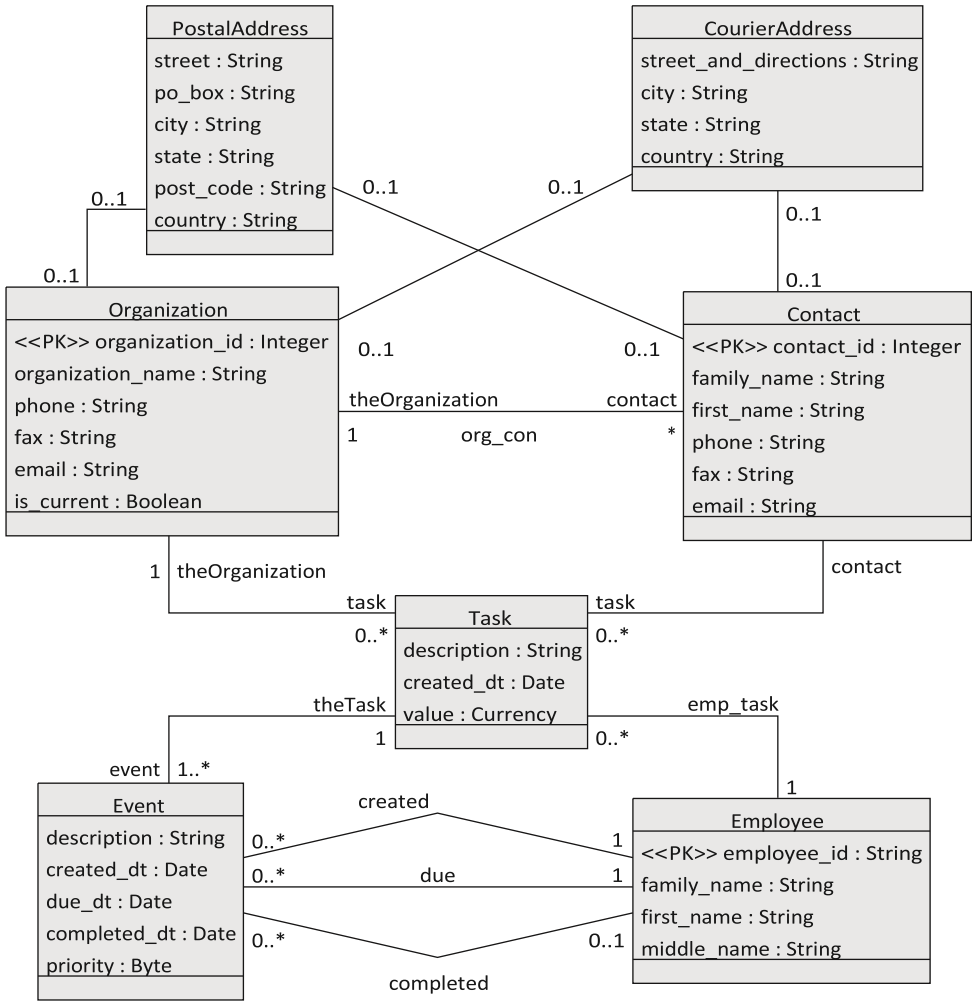



***Рис. 4.5. Спецификация ассоциаций (Управление контактами с клиентами)***

Кратність всіх асоціацій між класами і PostalAddress CourierAddress з одного боку і класами Організація і зв'язатися з з іншого дорівнює "нуль або один". Дамо пояснення до використання асоціацію між класами і організацією

Поштова адреса).

На одному кінці асоціації об'єкт Організація пов'язаний максимум з одним об'єктах тому PostalAddress, але тільки в тому випадку, якщо поштова адреса організації извес тен. На протилежному кінці асоціації конкретний об'єкт PostalAddress по в'я з об'єктом організації або об'єктом контакту. Отже, щоб удовле творять цим обмеженням, кратність повинна дорівнювати "нуль або

один". (Навіть за цих умов, саме обмеження необхідно окремо зафіксувати документально за допомогою засобів моделювання обмежень мови UML (розд. 5.1.2)).

Асоціація між класами і організації Зв'язатися демонструє використання у який спосіб асоціацій, так і рольових імен. Рольові імена перетворюються в імена атрибутів моделі реалізації програми Управління контактами з клієнтами. Модель реалізації містить наступні атрибути: Organization.contact і Contact.theOrganization. Префікс "The" означає, що роль theOrganization має кратність точно рівну "один". (Артикль "The" вживається в англійській мові для вказівки на певний, конкретний об'єкт. Прим. Ред.). Кратність ро чи контакт дорівнює "багато" (нижня і верхня межі не встановлені).

Кратність ролі контактного Між класами завдань і зв'язатися з не задана. Вимоги не пояснюють, чи повинно завдання бути безпосередньо пов'язано з контактом. Тому у нас немає впевненості в тому, чи може воно бути пов'язано більше, ніж з одним контактом.

Нарешті, існує три асоціації між класами і події співробітників. Ці ас асоціації встановлюють, хто із співробітників створює захід, хто відповідає за його виконання і хто, врешті-решт, завершує його. Під час створення мероприя ку співробітник, який працює над його завершенням, невідомий (тому крат ність на кінці асоціації завершена з боку працівника дорівнює "нуль або один").

# Обзорная лекция по курсу «Архитектура вычислительных систем» для специальности Т10.02

### 1. Архитектура как набор взаимодействующих компонентов

Ранее область применения вычислительных систем определялась ее быстродействием. Однако существует достаточно большое количество ВС, обладающих равным быстродействием, но имеющих совершенно разные способы представления данных, методы организации памяти, режимы работы, системы команд, набор ВнУ и т. д. Таким образом, ВС имеет, кроме быстродействия, ряд других характеристик, необычайно важных в той или иной области применения. Это стало особенно заметно при переходе к ВС четвертого и пятого поколений. Совокупность таких характеристик и легла в основу понятия архитектуры ВС.

Архитектура ВС определяет основные функциональные возможности системы, сферу применения (научно-техническая, экономическая, управление и т. д.), режим работы (пакетный, мультипрограммный, разделения времени, диалоговый и т. д.), характеризует параметры ВС (быстродействие, набор и объем памяти, набор периферийных устройств н т. д.), особенности структуры (одно-, многопроцессорная) и т. д. Составные части понятия "архитектура" можно определить следующей схемой (рис. 1.1).

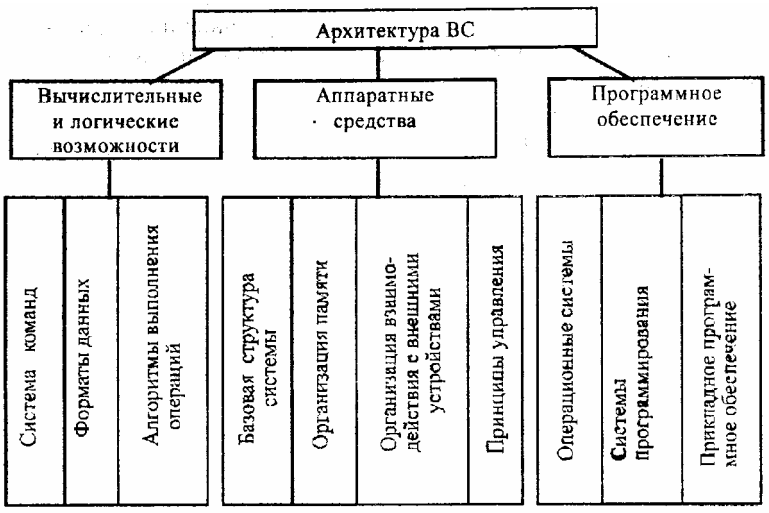



Рис. 1.1. Функциональные возможности ВС

Вычислительные и логические возможности ВС. Они обусловливаются системой команд (СК), характеризующей гибкость программирования, форматами данных и скоростью выполнения операций, определяющих класс задач, наиболее эффективно решаемых на ВС. Система команд ВС, базирующихся на архитектуре фон Неймана, сегодня мало чем отличается от СК ЭВМ 50-х годов. Большинство достижений в этой области остались незамеченными проектировщиками и соответственно не нашли адекватного воплощения в архитектуре современных компьютеров.

Анализ показывает, что в различных программах чаще всего встречаются достаточно простые команды: команды пересылки и команды процессора с использованием, регистров и простых режимов адресации. Не нашли широкого применения и нетрадиционные способы кодирования данных, несмотря на значительные возможности их в плане разработки быстродействующих алгоритмов арифметических операций. Среди них знакоразрядные системы, системы в коде вычетов и др.

Рассмотрим структуру системы команд в зависимости от класса решаемых задач (рис. 1.2).

К командам управления мы относим команды ввода-вывода данных и команды управления состоянием процессора, памяти и каналов.

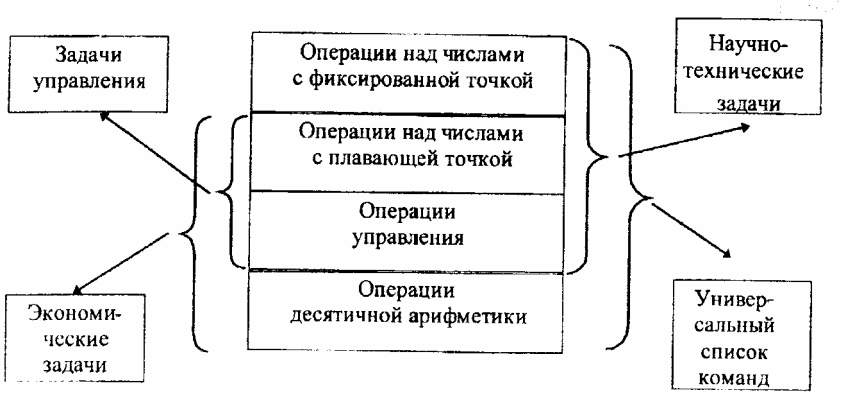



Рис. 1.2. Классификация СК по назначению

Как видно из рис. 1.2, для решения задач любого класса необходимы команды типов 2 и 3. Следовательно, эти типы команд должны присутствовать в любом компьютере.

Большое влияние на точность выполнения операций оказывают форматы данных. Современные компьютеры имеют развитую систему форматов.

Алгоритмы выполнения операций достаточно полно отражают производительность только однопроцессорных ВС.

Аппаратные средства. Простейшая ВС включает модули пяти типов:

центральный процессор, основная память, каналы, контроллеры и внешние устройства.

Процессор (УУ + АЛУ + намять) управляет работой системы и обеспечивает вычисления непосредственно по программе. Выполнение машинных команд, команд ввода-вывода (I/O), обращение к памяти, управление состоянием устройств инициализируются или выполняются с помощью процессора.

Основная память предназначена для хранения команд и данных и обеспечивает адресный доступ к ним от процессора. Современная память работает со скоростью, близкой к скорости работы процессора.

Каналы - спецустройства, управляющие обменом данных с внешними устройствами. Каналы инициируют свою работу с помощью процессора и затем переходят в автономный режим работы. Это, по сути, спецпроцессор ввода - вывода, обеспечивающий работу внешних устройств, контроль информации и т. д.

Контроллеры ввода-вывода служат для подсоединения внешних устройств (ВнУ) к каналам и обеспечивают обмен управляющей информацией с внешними устройствами, присвоение приоритетов и выдачу информации о состоянии ВнУ для канала, т. е. это устройства управления ВнУ.

ВнУ служат для ввода-вывода информации с различных носителей. Память может быть организована как многоуровневая с различным объемом и временем доступа к ней - сверхоперативная (СОЗУ), оперативная (ОП), внешняя (ВнП) (рис. 1 3), так и одноуровневая, виртуальная. Почти всегда виртуальная память есть переупорядоченное подмножество реальной памяти.

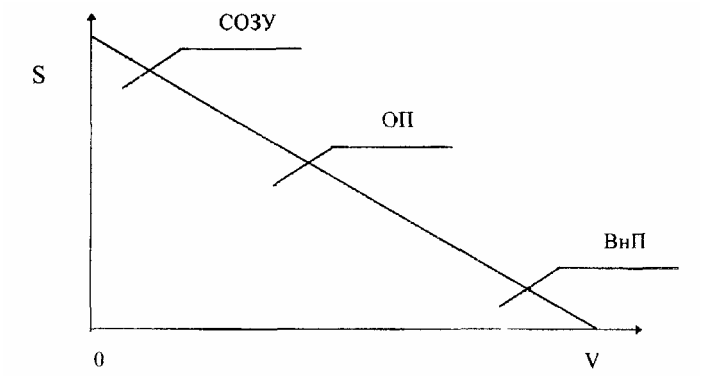



Рис 1 3 Типы памяти (V - объем, S - быстродействие)

Уровни иерархии памяти взаимосвязаны между собой, все данные одного уровня могут быть найдены на более низком уровне.

Успешное или неуспешное обращение к уровню памяти называют соответственно

попаданием (hit) или промахом (miss), а соответствующее время - временем обращения (hit time или miss penalty).

Существенное влияние на производительность ВС оказывают каналы ввода-вывода. Мультиплексный канал обеспечивает работу группы медленных устройств, блок-мультиплексный - группы быстрых устройств, селекторный - монополизирует информационную магистраль только одним быстродействующим устройством.

Для повышения пропускной способности каналов используют некоторые дополнительные меры, например буферизацию ВнУ путем введения памяти в состав самого устройства или контроллера.

Аппаратные средства защиты памяти служат для управления доступом к различным областям памяти в соответствии с имеющимися у пользователя полномочиями.

Программное обеспечение. Оно является составной частью архитектуры компьютера и существенно влияет на весь вычислительный процесс, в частности позволяет эффективно эксплуатировать аппаратные средства системы.

Операционная система (ОС) управляет ресурсами, разрешает конфликтные ситуации" оптимизирует функционирование системы в целом.

Широкий спектр языков программирования позволяет описывать практически любые задачи, а разнообразие компиляторов - их эффективно реализовывать.

Роль прикладного программного обеспечения (ПО) необычайно велика для решения тематических задач.

## 2. Архитектура как интерфейс между уровнями физической системы

Применительно к ВС термин "архитектура" может быть определен как распределение функций, выполняемых системой, по различным уровням и установление интерфейса между этими уровнями. На рис. 2.1 представлен набор уровней абстракции как специфического свойства архитектуры ВС Остановимся лишь на ключевых уровнях.

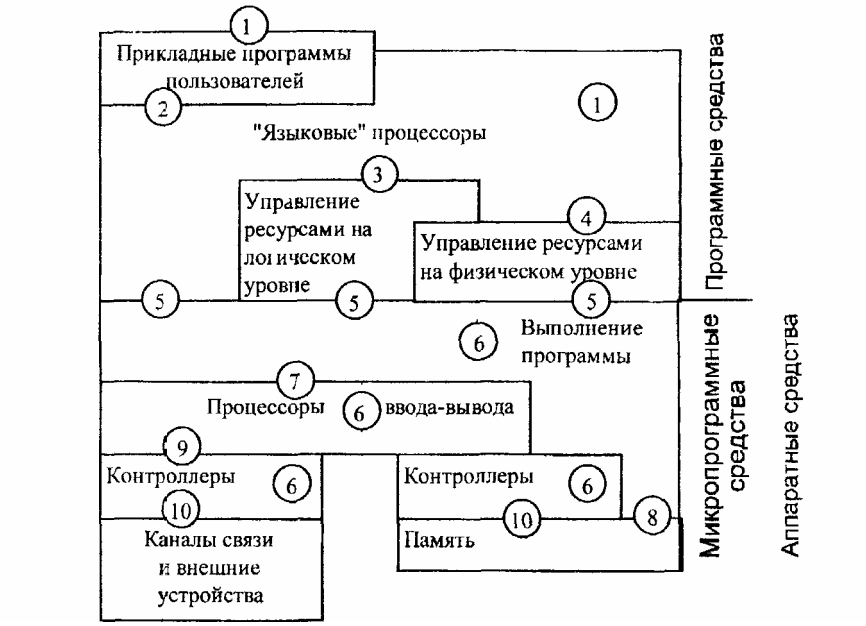



Рис 2.1 Многоуровневая организация архитектуры вычислительной системы

Архитектура первого уровня определяет, какие функции по обработке данных решаются системой, а какие передаются внешнему миру: пользователю, оператору ЭВМ, администратору баз данных и т.д. Система взаимодействует с внешним миром через два набора интерфейсов: языки (язык программирования, язык оператора терминала, язык управления заданиями, язык общения с базой данных, язык оператора ЭВМ) и системные программы (программы редактирования, связи, оптимизации, восстановления и обновления информации, интерпретации, управления и т.д., т.е. программы, созданные разработчиком системы). Оба интерфейса должны быть созданы при разработке архитектуры системы.

Уровни, определяемые интерфейсами внутри программного обеспечения, могут быть представлены как архитектура программного обеспечения. К примеру, если прикладные задачи реализованы на языках программирования, которые не входят в набор языков, предоставляемых системой пользователю, то здесь речь может идти об архитектуре уровня, позволяющего определить указанные языки.

Уровень 5 является одним из центральных уровней архитектуры и проводит разграничение между системным программным и аппаратным (т.е. схемным и микропрограммным) обеспечением. Он позволяет представить физическую структуру системы независимо от способа реализации.

Остальные уровни отражают интерфейсы и распределяют функции между отдельными частями физической системы.

Таким образом, можно сказать, что архитектура компьютера это абстрактное представление физической системы с точки зрения программиста.

**3 Концепция виртуальной памяти**

Каждая часть среды компьютера имеет собственное обозначение: ячейка - адрес, периферийное устройство - номер и т. д. В простейших компьютерах собственные обозначения указываются непосредственно в программе. В более сложных компьютерах программа отделена от среды "аппаратом преобразования собственных обозначений". Рассмотрим один элемент среды - память. Аппарат преобразования адреса (АПА) не находится под прямым управлением программы и связь с ним осуществляется только через процедуры, работающие в управляющем режиме.

Если программисту безразлично существование АПА, то он работает с набором ячеек и периферийных устройств, образующих "виртуальную (математическую, мнимую) среду". Почти всегда виртуальная среда есть переупорядоченное подмножество реальной среды. Каждому виртуальному элементу соответствует реальный элемент, но обратное не всегда верно.

Рассмотрим один из элементов виртуальной среды - виртуальную память (ВП).

**Задачи, решаемые виртуальной памятью**

Виртуальный адрес - адрес, по которому ссылаются на ячейку виртуальной памяти. Область виртуальных адресов - это множество всех виртуальных адресов.

Использование виртуальной адресации обусловливается следующими обстоятельствами.

1. Однородность области адресов. Представим себе реальный компьютер без виртуальной памяти. Пусть на нем выполняется параллельно несколько процессов. У каждого процесса будет отдельная локальная среда и каким-то образом распределяемые элементы общей среды. Программисту требуется заранее знать, к каким конкретно частям общей среды его процедура может обращаться. Это затруднительно для пользователей ЭВМ, составляющих свои собственные программы. Отвести наперед фиксированную область среды для каждого процесса невозможно, ибо положение каждой конкретной программы определяется положением всех других программ.

При виртуальной адресации каждый процесс может выполняться в памяти начиная с фиксированной (обычно нулевой) ячейки, имеющей необходимы размеры области ЗУ. Автору безразлично, в каком участке памяти выполняется его программа, так как каждое обращение к виртуальной памяти во время выполнения посредством АПА преобразуется в реальное обращение.

Замечание. АПА работает не во время ассемблирования, а непосредственно во время выполнения обращения,

2. Защита памяти. Общеизвестно, что основная цель защиты памяти состоит в том, чтобы не дать возможности некорректному процессу испортит часть среды, относящуюся к другому процессу. Особенно это важно при защите сред управляющих процедур. Виртуальная адресация здесь используется следующим образом: при каждой ссылке процессом на память проверяется, принадлежит ли она к области виртуальных адресов, отведенных для данного процесса.

3. Изменение структуры памяти. При проектировании больших программ структура памяти машины с малой ОП явно усложняет проектируемую программу. Применение виртуальной адресации позволяет преобразовать память на разных ступенях иерархии в "одноуровневую память" с одинаковым доступом ко всем элементам и отобразить ее на реальную память.

Для удовлетворения пунктов 1-3 требуется аппарат "страничной" организации памяти, для пунктов 1,2 достаточно иметь регистры "настройки" регистры "базы" и "границы".

1.6.2. Страничная организация памяти

Отображение виртуальной памяти в реальную обычно осуществляется с помощью страничной организации памяти.

Виртуальную память в системе со страничной организацией памяти делят на ряд "блоков" фиксированной длины, равной 2k где k - целое натуральное число. Так как первая ячейка блока N + 1 примыкает к последней ячейке блока N, то программисту факт разбиения ВП на блоки учитывать не требуется.

Оперативная память компьютера делится на "страницы", а вспомогательная - на "сегменты" такого же размера.

Виртуальную память пользователя можно разделить на три типа:

•"активные" блоки, которые содержат программу и данные, используемые в текущий момент;

•"пассивные" блоки, содержащие программу и данные, которые будут использоваться при выполнении программы;

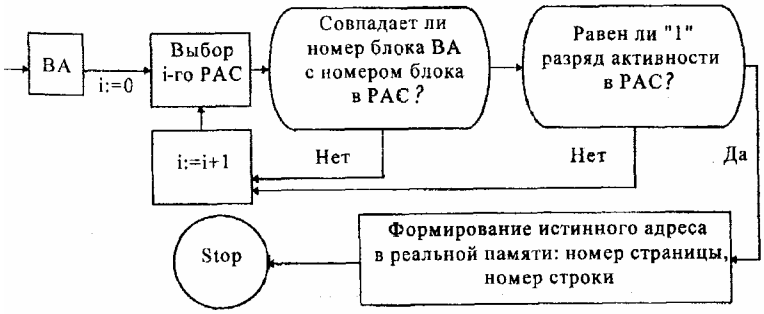
•"мнимые" блоки, к которым не обращаются на протяжении выполнения программы.

Соотношения между первым и вторым типами блоков ВП в процессе выполнения программы изменяются. Третий тип блоков возможен лишь при наличии очень большой области ВП,

Аппарат виртуальной адресации должен отображать виртуальную среду на реальную, причем так, чтобы "активные" блоки находились в оперативной памяти, "пассивные" - по возможности в оперативной или вспомогательной, а мнимые - нигде.

Преобразование виртуального адреса в реальный происходит с помощью регистров адресов страниц (РАС). Аппарат виртуальной адресации отображает виртуальный адрес в реальный следующим образом; виртуальный адрес сравнивается одновременно с содержимым всех РАС. Единственным сравнимым с ним РАС будет тот, который содержит гот же номер блока и "1" в разряде активности. РАС определяет номер страницы, с которой он связан. Для получения реального адреса памяти к номеру страницы данного РАС присоединяется номер строки из виртуального адреса (ВА). В последовательной интерпретации процесс отображения ВА в реальный можно представить следующей схемой.

Разряд записи в РАС служит для экономии времени перезаписи "страницы" в ВнП. Когда блок переносится из ВнП в оперативную память, в разряд записи пишут "О". Если какая-то строка данной страницы ОП изменяется в процессе обращения к ней, то пишут "1". И пока в разряде записи "0", эта страница является точной копией соответствующего блока в ВнП.




В разряд использования "1" посылается при очередном обращения к данной странице. Через равные промежутки времени содержимое регистров использования сканируется и записывается в определенную ячейку памяти тем самым создавая статистику использования данной страницы.

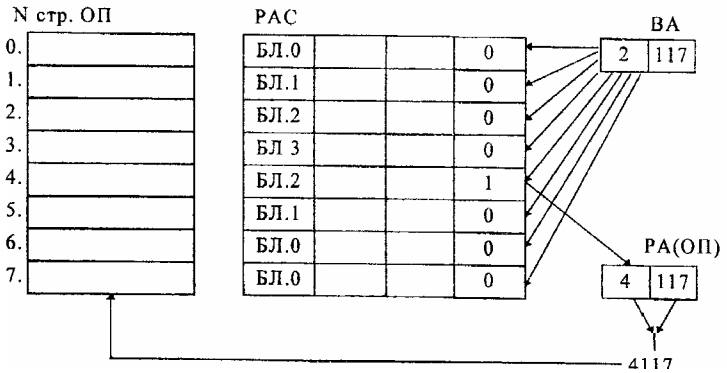
Полный процесс отображения ВП в реальную выполняется за три этапа. Д

1. Если происходит обращение к блоку ВП, который должен отобрав жаться на страницу ОП, РАС для которой является одним из ассоциативных регистров, то процесс отображения выполняется согласно выше приведённой схеме.

2. Если обращение происходит к блоку ВП, для которого условия п. 1 не выполняются, то номер виртуального блока служит указателем для обращения к таблице блоков (ТБ) в оперативной памяти. Если из ТБ видно, что искомый блок находится в оперативной памяти, то номер страницы также может быть выбран из ТБ.

Если же нужный блок находится во вспомогательной памяти или отсутствует вообще, то происходит прерывание, и тогда переходим к управляющей процедуре (см. п. 3)

3. Обработка прерывания. Для эффективной работы системы необходимо, чтобы ассоциативные регистры содержали РАС для наиболее часто используемых страниц. Основная цель методов оптимизации распределения реальной памяти состоит в минимизации числа обменов между оперативной и вспомогательной памятью. Отсюда следует, что необходимо корректно выбирать стратегию замещения страниц.




Основные стратегии замещения страниц, наиболее часто используемые на практике: циклическое изгнание, случайная выборка, наименьшее число обращений с момента последнего прерывания. Результаты экспериментов показали, что ни в одном случае разница в числе обменов, требуемых конкретной задачей при использовании указанных стратегий, не превышала 10 %.

**4. Взаимодействие и управление процессами**

Понятие "программа" - недостаточно мощное понятие для описания «рационных систем, однако детальное исследование программ позволяет выделить ряд важных концепций, которые проясняют принципы построения операционных систем.

При выполнении программ явно выделяются три объекта:

• последовательность команд или процедура, которая определяет программу;

• процессор, который выполняет процедуру;

• среда, т.е. та часть окружающею мира, которую процессор может непосредственно воспринимать или изменять.

К среде относятся, например, память, универсальные и управляющие регистры ЭВМ, поскольку они могут и восприниматься, и изменяться программой.

Можно выделить следующие свойства программы:

• операции, заданные процедурой, выполняются строго последовательно, т. е. следующий шаг не начинается, пока предыдущий полностью не выполнится (из определения программы мы исключаем любой процесс, содержащий совмещение операций);

• среда полностью управляется программой, а следовательно, и изменяется только в результате шагов, выполненных программой;

• время выполнения операций, а также временной интервал между выполнением операций не имеют отношения к выполнению программы. Естественно, здесь мы не учитываем, что вся программа должна быть выполнена за разумный интервал времени;

• совершенно не имеет значения, выполняется ли программа целиком на одном процессоре, лишь бы не изменялась среда программы.

Программа в силу указанных свойств предполагает отсутствие внешнего воздействия на её выполнение. Хорошим приближением такого рода программ являются программы, написанные на языках высокого уровня. Важное достоинство таких программ - возможность точного повторения их работы, если только она не имеет доступа к часам реального времени.

Программы, удовлетворяющие указанным свойствам, не могут быть операционными системами, ибо:

• предполагается, что операционная система эффективно использует ресурсы компьютера, а это требует совмещения операций различных компонентов;

• ОС отвечает на поступающие запросы за определенное время, а так как они поступают произвольным образом, то последовательность операций определяется не только самой системой.

**5. Понятие процесса и состояния**

Создавать современные операционные системы без теоретической базы стало затруднительно, так как невозможно повторить условия, приведшие к ошибке во время работы, а следовательно, выявить ее источник. Вес это привело к появлению понятия процесса. Понятие "процесс" в последнее время встречается в литературе по операционным системам довольно часто, однако вкладываемый в это понятие смысл иногда сильно различается. Мы будем пользоваться следующим определением.

Процесс есть тройка (Q, f, g), где Q - множество состояний процесса, f - функция действия f: Q -> Q; g - начальное состояние процесса.

Действия, реализуемые процессом, будем рассматривать как результат выполнения некоторой программы на реальном (виртуальном) процессоре.

Перечислим некоторые свойства процесса:

• процесс не является закрытой системой и может взаимодействовать с другими процессами, воспринимая или изменяя часть среды, которую он с ними разделяет;

• каждый процесс живет лишь временно. Имеется и "главный" процесс выполняемый вручную при включении компьютера, который начинает всю цепочку процессов;

• в любой момент процесс может быть описан его состоянием. Все параметры (переменные), характеризующие текущее состояние процесса, объединяются в "вектор состояний" или "слово состояний", которые и позволяют возобновить процесс после его прерывания, в том числе и локальную среду.

Цикл жизни процесса может быть представлен как переход его из одного состояния в другое.

Пример. Пусть заданы три процесса (задания пользователя), одновременно присутствующие в системе с мультипрограммированием. И пусть каждый из них может находиться в трех состояниях: ожидание, готовность, выполнение.

Опишем эти состояния.

ОЖИДАНИЕ. Процесс ожидает выполнения какого-либо события (например, завершения операции I/O).

ГОТОВНОСТЬ. Процесс готов к выполнению, но процессов больше чем процессоров, и он должен ждать своей очереди.

ВЫПОЛНЕНИЕ. Процессору выделены все ресурсы, в том числе и процессор, и его программы выполняются в настоящее время.

Схема на рисунке предполагает, что процессы уже существовали в памяти компьютера и будут существовать всегда. На практике пользователь должен эти процессы (задания)

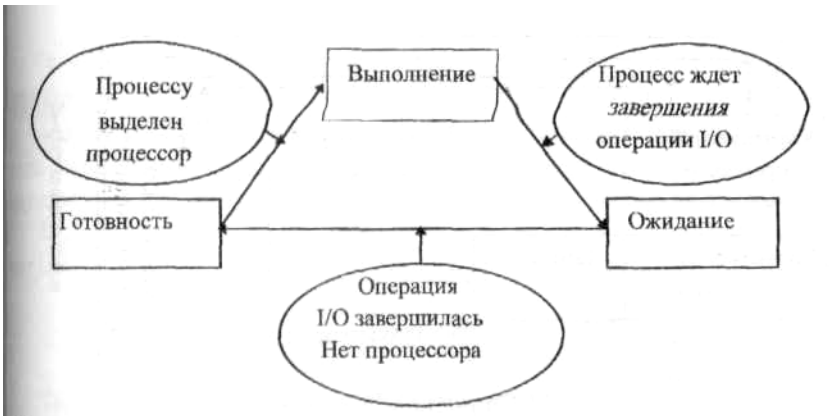



*Рис. 7.1.* Схема перехода процессов из одного состояния в другое


*Рис. 7.2.* Полная схема обработки процесса

предоставить системе. Тогда реальная схема будет выглядеть так.

Здесь дополнительно введены состояния: т доставление, хранение и завершение. Поясним их.

ПРЕДОСТАВЛЕНИЕ. Пользователь предоставляет задание системе, на которое она должна отреагировать.

ХРАНЕНИЕ. Задание преобразовано во внутреннюю форму, понятную лишь компьютеру, но ресурсы еще не выделены (например, задание записано на диск). Процесс, в случае выделения ресурсов, переходит в состояние готовности.

ЗАВЕРШЕНИЕ. Процесс завершил свои вычисления и все выделенные процессу ресурсы могут быть освобождены и возвращены системе.

**6. Организация системы прерывания.**

Появление системы прерывания в компьютерах конца 50-х годов позволило существенно продвинуться в разработках ЭВМ, по новому переосмыслив их возможности и среды применения. Действительно, уже на первых ЭВМ, обладающих этим свойством, появилась возможность автономной работы периферийных устройств после их запуска центральным устройством правления. По окончании работы или в связи с другой причиной периферийное устройство смогло через прерывание заставить центральный процессор обратить на себя внимание. Это открыло путь к широкому совмещению операций, что повлекло за собой естественное усложнение математического обеспечения. Современные операционные системы полностью разрабатываются и базируются на развитой системе прерывания.

**Основные определения и характеристики**

Работу вычислительной системы можно представить как последовательность программно-определяемых (порождаемых программой и возможные моменты появления которых известны) и программно-независимых (вызванных посторонними от программы источниками или моменты возникновения которых неизвестны) событий.

События, происходящие вне процессора, как правило, программно-независимые (выход параметров объекта за дозволенные пределы, запросы оператора и т. д.) и происходят асинхронно. То же относится и к периферийным устройствам, работающим одновременно с выполнением программы в процессоре, хотя начало их работы и задает процессор, однако её окончание и последовательность операций неизвестны.

События, происходящие внутри процессора, могут быть двух типов. Последовательность арифметических операций определяется программой, в то время как особые ситуации (переполнение, попытка деления на нуль и т.д.) зависят от сочетания операндов и предусмотреть их во время программирования практически невозможно. ВС, вообще говоря, должна реагировать на любые события, которые могут повлиять на процесс вычислений. В случае программно-определяемых событий для этого достаточно иметь специальный набор команд (переход по нулю, по знаку и т. д.). Если же события программно-независимые, то, как правило, неэффективно использовать обычные программные методы для их опознания.

Чтобы ВС могла реагировать на программно-независимые события при минимальных усилиях программиста и максимально возможном быстродействии, ее надо снабдить дополнительными аппаратно-логическими средствами, совокупность которых называют системой прерывания программ (СПП)

*Определение*. Прерывание программы это свойство ВС при возникновении особых событий временно прекратить выполнение текущей программы и передать управление программе, специально предусмотренной для обработки данного события.

В системе с прерыванием каждое программно-независимое событие (источник прерывания) должно, если оно может повлиять на ход обработки сопровождаться сигналом, говорящим о его возникновении. Назовем эти сигналы запросами прерывания. Программы, затребованные запросами прерывания, назовем прерывающими программами, в отличие от прерванных грамм, выполнявшихся компьютером до появления запросов прерывания.

Так как функции по сохранению и восстановлению состояния прерванной программы возлагаются на саму прерывающую программу, то последняя должна состоять из трех частей: подготовительной и восстановительной, обеспечивающих переход к нужной программе, и собственно прерывающей программы.

По окончании работы прерывающей программы переход может быть осуществлён либо к прерванной программе, либо к другой прерывающей программе.

Так как всевозможные запросы на прерывание вырабатываются независимо и асинхронно, то возможны такие ситуации:

• приход запросов последовательный;

• одновременный приход нескольких запросов;

• приход запроса во время выполнения прерывающей программы. Следовательно, должен быть организован порядок, в котором поступающие запросы удовлетворяются. Если в ВС имеются средства для обслуживания запросов в порядке присвоенного им приоритета, то такие системы срывания называются приоритетными.

СПП, как правило, выполняют следующие основные функции:

• организуют вход в прерывающую программу;

• осуществляют приоритетный выбор между запросами прерывания;

• обеспечивают возврат к прерванной программе и программное изменение приоритетов программ.

**Вход в прерывающую программу**

Система прерывания программ должна определить допустимый момент прерывания текущей программы и начальный адрес прерывающей программы. Наиболее простыми являются три следующих способа определения допустимого момента прерывания.

*Метод помеченного оператора (опорных точек)*. Суть метода состоит в следующем. В специальные разряды команд, после которых допускается прерывание, записывается определенный знак, разрешающий или запрещающий прерывание. Тогда вдоль программы можно расставить её опорные точки. Здесь желательно так расставить точки прерывания, чтобы информация, находящаяся в регистрах процессора после выполнения данной команды, дальше не использовалась. Это уменьшает время обслуживания и увеличивает время реакции.

*Покомандный способ*. Здесь прерывание допускается после выполнения любой команды. Способ прост в реализации.

*Метод быстрого реагирования*. Прерывание допускается во время выполнения любой команды, т.е. после выполнения её очередного такта.

Из-за простоты и удачного сочетания характеристик прерывания наибольшее распространение получил второй способ, хотя в последних ВС используется и третий. Так, если обнаружено, что адрес операнда сформирован неверно, то целесообразно сразу же прервать выполнение операции, чтобы ошибка не распространилась на другие такты. Это необходимо при мультипрограммной работе, если адресованный операнд в команде принадлежит внешней памяти. Здесь текущая операция не может быть продолжена, пока итеративная память не получит данные из ВнП.

Распознавание начального адреса прерывающей программы можно осуществлять как программным, так и аппаратным способом.

Суть программного распознавания: все линии связи, по которым приходят запросы прерывания, объединяются в схему ИЛИ, формирующую на выходе один и тот же сигнал, который в допустимый момент прерывания поступает в прерывающую программу. Последняя распознает запросы и разветвляется для их выполнения.

Вход системы прерывания, обладающей способностью формировать собственный адрес начала прерывающей программы, принято называть уровнем прерывания.

Таким образом, система с аппаратным распознаванием причин прерывания может быть названа многоуровневой системой прерывания.

Простейший способ указания начальных адресов состоит в следующем. Каждому уровню присваивается номер и в памяти отводится ячейка, адрес которой, к примеру, равен номеру уровня. В такой ячейке памяти может храниться команда перехода к остальной части прерывающей программы, которая может находиться в любом месте памяти. Набор фиксированных по отношению к своим уровням ячеек памяти образует таблицу входов в прерывающие программы, содержание которых может менять программист.

**Приоритетное обслуживание прерываний**

Аппарат приоритетов предназначен для повышения эффективности использования всех ресурсов ВС. Так, неэффективно одновременное выполнение двух заданий, каждое из которых требует большой загрузки устройств ввода-вывода и незначительно использует центральный процессор. Приоритет задания может назначаться исходя из: времени его выполнения ("короткие" задания имеют более высокий приоритет по сравнению с "длинными" заданиями); объема используемой оперативной памяти (задания, требующие большого объема памяти, не должны иметь одинаковый приоритет); интенсивности и объема использования других ресурсов ВС; срочности выполнения и т.д.

При программном распознавании причин прерывания прерывающая программа начинает анализ с тех запросов на прерывание, которые имеют колее высокий приоритет. При этом в процессе работы мы можем программным путем изменить приоритет запросов. При аппаратном распознавании причин прерывания указанные функции возлагаются на специальное оборудование.

Понятие приоритета в прерывании программ имеет два смысла. Предположим, что никаких ограничений на время поступления запросов прерывания не накладывается. При одновременном поступлении нескольких запросов для немедленного удовлетворения может быть принят только один. Этот тип приоритета, определяющий очередность рассмотрения запросов прерывания, называют приоритетом между запросами прерываний. Прерывающие программы, однако, могут иметь относительно текущей программы различную степень важности, и не любым запросом может быть прервана выполняющаяся программа. Чтобы иметь возможность прервать текущую программу, принятый СПП запрос должен соответствовать программе более важной, нежели выполняемая в данный момент. Этот тип приоритета определяет старшинство программ и обычно его называют приоритетом между прерывающими программами.

В случае простейшей аппаратной реализации приоритета между запросами прерывания может быть использован метод "последовательного поиска".

Суть метода состоит в последовательном изменении содержимого счётчика от 0 до $2^n$ - 1 и просмотре всех $2^n$ уровней прерывания до совпадения содержимого счетчика с номером уровня.

При одновременном появлении нескольких запросов жестко закрепляйся запрос с уровня с меньшим номером. Метод "последовательного поиска" прост в реализации, но время реакции велико, так как в общем случае необходимо прохождение счетчиком всех $2^n$ позиций, что при большом n может выходить за допустимые временные пределы. Особенно это важно при работе

в режиме реального времени.

Приоритет между прерывающими программами определяет, какие программы могут прервать данную программу, а какие нет. Этот вид приоритета для многоуровневых систем с достаточной глубиной прерывания имеет гораздо большее значение, чем приоритет между запросами прерывания.

Таким образом, приоритет между запросами прерывания нужен лишь для выбора одного запроса из многих, а приоритет между прерывающими программами определяет фактический порядок выполнения программ.

Так как степень важности программ, их объем, требуемые ресурсы т. д. могут изменяться в ходе вычислительного процесса, то только приоритеты между запросами прерывания могут быть строго зафиксированы. Приоритеты же между прерывающими программами должны быть программно управляемыми.

*Маска прерывания*. Маска - шаблонная последовательность знаков, управляющая сохранением или исключением отдельных частей другой последовательности знаков. В простейшем виде маска - это двоичное число, каждый разряд которого соответствует одному из уровней прерывания и разрешает (например, состояние "1") или запрещает (состояние "0") прерывание запросов, относящихся к данному уровню. Управление приоритетом находятся полностью в распоряжении программы. Для каждой прерывающей программы может быть установлена своя маска, указывающая, какие программы способны ее прерывать. Каждый разряд маски соответствует отдельной программе. Маски всех программ хранятся в памяти. Если какая-нибудь программа вызывается для выполнения, то её маска засылается в регистр маски. Физически маска реализуется обычно в виде триггерного регистра, состояние которого можно изменить программным путем. При формировании его состояние "1" получают лишь те триггеры, которые соответствуют программам с более высоким, чем у данной программы, приоритетом.

**Организация возврата к прерванной программе**

Для осуществления возврата к прерванной программе необходимо полностью восстановить её начальное состояние. Информацию, которую следует сохранять при прерывании программы, можно разделить на основную (которая запоминается всегда) и дополнительную (необходимость запоминания второй зависит от содержания прерывающей программы).

В основную информацию можно включить:

• содержимое счетчика адреса команд, т. е. адрес первой невыполненной команды прерванной программы;

• триггер состояния системы: "рабочее" или состояние "ожидания";

• маска прерывания, устанавливаемая каждой новой программой;

• код прерывания - двоичное число, отдельное для каждого уровня объединяющего прерывание от нескольких источников, по которому прерывающая программа опознает конкретный источник прерывания.

Так как код прерывания обычно находится в общем для всех регистре, то, если предыдущий код полностью себя не исчерпал (а это почти всегда так), с приходом новой прерывающей программы его надо запомнить.

Указанная информация образует так называемое "слово состояния программы" (ССП)

(иногда его называют вектором состояния программы), которое хранится в некотором поле памяти компьютера. В момент прерывания старое ССП, относящееся к прерванной программе, заменяется ССП прерывающей программы, а в конце прерывания старое ССП восстанавливается прерывающей программой. Для ускорения процесса замены ССП эту процедуру выполняют обычно аппаратным путем.

Замечание. В период сохранения и восстановления ССП прерывания любого уровня запрещены.

К дополнительной информации относят содержимое:

- арифметических регистров;
- индексных регистров;
- прочих программно-доступных регистров, общих для всех программ, и т.п.

Сохранение дополнительной информации увеличивает время обслуживания. Поэтому программисту надо тщательно продумать, что из дополнительной информации следует запоминать в каждом конкретном случае. Более того, момент прерывания следует выбирать так (если это возможно), чтоб дополнительной информации для сохранения было как можно меньше.