# Архітектура та проектування програмного забезпечення

Огляд патернів
Патерни проектування класів / об'єктів
Архітектурні системні патерни
Патерни управління
Патерни, призначені для представлення даних у Web
Патерни інтеграції корпоративних інформаційних систем
Model-Viewer-Controller
Схема архітектури MVC

## ПОНЯТТЯ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

**Архітектура програмного забезпечення** (англ. software architecture) - це структура програми або обчислювальної системи, яка включає програмні компоненти, видимі зовні властивості цих компонентів, а також відносини між ними. Цей термін стосується також документування архітектури програмного забезпечення. Документування архітектури ПЗ спрощує процес комунікації між зацікавленими особами (англ. stakeholders), дозволяє зафіксувати прийняті на ранніх етапах проектування рішення про високорівневий дизайн системи і дозволяє використовувати компоненти цього дизайну і шаблони повторно в інших проектах.

**Огляд**

Область комп'ютерних наук з моменту свого утворення зіткнулася з проблемами, пов'язаними зі складністю програмних систем. Раніше проблеми складності вирішувалися розробниками шляхом правильного вибору структур даних, розробки алгоритмів та застосування концепції розмежування повноважень. Хоча термін "архітектура програмного забезпечення" є відносно новим для індустрії розробки ПЗ, фундаментальні принципи цієї області невпорядковано застосовувалися піонерами розробки ПЗ починаючи з середини 1980-х. Перші спроби усвідомити і пояснити програмну архітектуру системи були повні неточностей і страждали від нестачі організованості, часто це була просто діаграма з блоків, сполучених лініями. У 1990-ті роки спостерігається спроба визначити і систематизувати основні аспекти даної дисципліни. Початковий набір шаблонів проектування, стилів дизайну, передового досвіду (best-practices), мов опису та формальна логіка були розроблені протягом цього часу.

Основною ідеєю дисципліни програмної архітектури є ідея зниження складності системи шляхом абстракції і розмежування повноважень. На сьогоднішній день до цих пір немає згоди щодо чіткого визначення терміна "архітектура програмного забезпечення".

Будучи в справжній момент свого розвитку дисципліною без чітких правил про "правильний" шлях створення системи, проектування архітектури ПЗ все ще є сумішшю науки і мистецтва. Аспект "мистецтва" полягає в тому, що будь-яка комерційна система має на увазі наявність застосування або місії. Те, які ключові цілі має система, описується за допомогою сценаріїв як нефункціональні вимоги до системи, також відомі як атрибути якості, що визначають, як буде вести себе система. Атрибути якості системи включають в себе **відмовостійкість**, збереження зворотної сумісності, розширюваність, надійність, придатність до сервісного обслуговування (maintainability), доступність, безпека, зручність використання, а також інші якості. З точки зору користувача програмної архітектури, програмна архітектура дає напрям для руху і вирішення завдань, пов'язаних зі спеціальністю кожного такого користувача, наприклад, зацікавленої особи, розробника ПЗ, групи підтримки ПЗ, фахівця із супроводу ПЗ, фахівця з розгортання ПЗ, тестера, а також кінцевих користувачів. У цьому сенсі архітектура програмного забезпечення насправді об'єднує різні точки зору на систему. Той факт, що ці декілька різних точок зору можуть бути об'єднані в архітектурі програмного забезпечення є аргументом на захист необхідності та доцільності створення архітектури ПЗ ще до етапу розробки ПЗ.

**Підсумок**

**Архітектура** - це принцип організації компонентів усередині системи: їх кількість, якість, інтерфейси і протоколи взаємодії.

Що залежить від архітектури? Від неї залежить ціна на підтримку і розробку нових фіч, трудовитрати на побудову цілої системи з використанням даної архітектури. Тобто формально від архітектури залежить найважливіший параметр розробки - собівартість. А побічно ще і можливість повторного використання коду, а разом з ним і зменшення трудовитрат на кожну подальшу розробку.

Добре, ми з'ясували що архітектура це дуже важливий аспект розробки. Але що ж це таке? У контексті PHP5 додатки з упором в **парадигму** - це ієрархія класів, інтерфейси та схеми їх взаємодії.

Вибір або створення архітектури залежить від конкретних завдань. Наприклад, наскільки універсальним планується додаток, які модулі повинні бути присутніми, яка запланована навантаження на ресурс.

## ПРИНЦИПИ ПРОЕКТУВАННЯ

## ЯКІСТЬ АРХІТЕКТУРИ

**Ознаки загниваючого проекту**

Отже, будь-який програмний код має взаємозалежності одних частин від інших. Класи вимагають наявності інших класів, одні функції викликають інші і т.д. У міру зростання будь-якого проекту взаємозалежностей стає все більше і більше. Вимоги до проекту змінюються, розробники іноді вносять швидкі й не завжди вдалі рішення. Якщо залежностями грамотно не управляти, то проект неминуче почне загнивати. Код стає складніше розуміти, він частіше ламається, стає менш гнучким і важким для повторного використання. У результаті швидкість розробки падає, проект чинить опір змінам, і ось вже серед розробників звучать заклики «Давайте все переробимо! Наступного разу ми зробимо супер-архітектуру ». Ось найбільш поширені ознаки поганого або загниваючого в плані коду проекту:

**Закритість** (rigid) - система відчайдушно чинить опір змінам, неможливо сказати, скільки займе реалізація тієї або іншої функціональності, тому що зміни, швидше за все, торкнуться багатьох компонентів системи. Через це вносити зміни стає занадто дорого, тому що вони вимагають багато часу.

Нестійкість, **крихкість** (fragile) - система ламається в непередбачених місцях, хоча зміни, були проведені до цього, зламані компоненти явно не зачіпали.

**Нерухомість** або **монолітність** (not reusable) - система побудована таким чином і характер залежностей такий, що використовувати будь-які компоненти окремо від інших не представляється можливим.

**В'язкість** (high viscosity) - код проекту такий, що зробити що-небудь неправильно набагато простіше, ніж зробити щось правильно.

**Невиправдані повторення** (high code duplication) - розмір проекту набагато більший, ніж він міг би бути, якби абстракції застосовувалися частіше.

**Надмірна складність** (overcomplicated design) - проект містить рішення, користь від яких неочевидна, вони приховують реальну суть системи, ускладнюючи її розуміння і розвиток.

Майже будь-який більш-менш досвідчений розробник може пригадати приклад коду, який відповідав хоча б одному за цією ознакою.

## Як зробити кращу архітектуру

За довгі роки розумні люди виробили деякі основоположні принципи ООП, дотримання яких дозволяє створювати кращу архітектуру:

**Висока зчепленість коду** (High Cohesion) - код, відповідальний за яку-небудь одну функціональність, повинен бути зосереджений в одному місці.

**Низька зв'язаність коду** (Low Coupling) - класи повинні мати мінімальну залежність від інших класів.

**Вказуй, а не питай** (Tell, Don't Ask) - класи містять дані і методи для оперування цими даними. Класи не повинні цікавитися даними з інших класів.

**Не розмовляй з незнайомцями** (Don't talk to strangers) - класи повинні знати тільки про своїх безпосередніх сусідів. Чим менше знає клас про існування інших класів або інтерфейсів - тим більш стійкий код.

Всі ці рекомендації спрямовані на те, щоб постаратися розвести класи по сторонах, зосередити сильні взаємозв'язки в одному місці і провести чіткі розмежувальні лінії в коді.

Але ці принципи надто розпливчасті, тому з'явився якийсь набір більш чітких правил, якими слід керуватися при формуванні архітектури.

**Принцип персональної відповідальності** (Single Responsibility Principle) - клас володіє тільки 1 відповідальністю, тому існує тільки 1 причина, що приводить до його зміни.

**Принцип відкриття-закриття** (Open-Closed Principle) - класи повинні бути відкриті для розширень, але закриті для модифікацій. Здається, що це неможливо, однак варто згадати шаблон проектування Strategy і стає більш-менш зрозуміло.

**Принцип підстановки** Ліскоу (Liskov Substitution Principle) - дочірні класи можна використовувати через інтерфейси базових класів без знання про

те, що це саме дочірній клас. Інакше - дочірній клас не повинен заперечувати поведінку батьківського класу і повинна бути можливість використовувати дочірній клас скрізь, де використовувався батьківський клас.

**Принцип інверсії залежностей** (Dependency Inversion Principle) - всередині системи стоять на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не залежать від подробиць.

**Принцип відділення інтерфейсу** (Interface Segregation Principle) - клієнти не повинні потрапляти в залежність від методів, якими вони не користуються. Клієнти визначають, які інтерфейси їм потрібні.

Тепер докладніше.

## Принципи гарної ахрітектури

### Принцип відкриття-закриття (Open Close Principle або OCP)

Програмні суті такі як класи, модулі та функції повинні бути відкриті для розширення, але закриті для змін.

Ви можете обговорювати його, коли пишете ваші класи, щоб бути впевненими в тому, що коли вам буде потрібно розширити поведінку, ви не повинні будете змінювати клас, але можете розширювати його. Подібний же принцип застосуємо для модулів, пакетів і бібліотек. Якщо у вас є бібліотека, що складається з множини класів, то є багато причин для того, щоб ви вважали за краще розширення замість зміни коду, який вже написаний (заради зворотної сумісності, повернення до попереднього тестування і т.д.). Це причина, по якій ми повинні бути впевнені, що наші модулі слідують Принципу відкриття-закриття. По відношенню до класів Принцип відкриття-закриття може бути гарантовано корисний за рахунок використання Абстрактних Класів і конкретних класів для реалізації їх поведінки. Це буде змушувати мати Конкретні Класи, що розширюють Абстрактні Класи замість їх зміни. Деякі приватні випадки цього принципу є Шаблонний Патерн і Стратегічний Патерн (Template Pattern and Strategy Pattern).

### Принцип Заміщення Ліскоу (Liskov's Substitution Principle)

Похідні типи повинні бути здатні повністю заміщатися їх базовими типами.

Цей принцип є всього лише розширенням Принципу відкриття-закриття в умовах поведінки, що означає, що ми повинні бути впевнені, що нові похідні класи є розширенням базових класів без зміни їх поведінки. Нові похідні класи повинні бути здатні замінювати базові класи без будь-яких змін у коді. Принцип Заміщення Ліскоу був введений на 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy.

**Принцип Єдиної Відповідальності (Single Responsibility Principle)**

Клас повинен мати тільки одну причину для зміни.

У цьому контексті відповідальність розглядається як єдина причина для зміни. Цей принцип стверджує, що якщо ми маємо 2 причини для зміни класу, то ми повинні розділити функціональність на 2 класи. Кожен клас повинен мати тільки одну відповідальність, і в майбутньому, якщо нам буде потрібно зробити одну зміну ми будемо робити це в класі, який утримує цю одну відповідальність. Коли нам потрібно робити зміни в класі, який має більше відповідальностей, зміна може вплинути на іншу функціональність класів.

Принцип Єдиної Відповідальності був введений Tom DeMarco в його книзі «Structured Analysis and Systems Specification, 1979». Роберт Мартін переробив цю концепцію і визначив, що відповідальність є причиною для зміни.

**Принцип Відділення Інтерфейсу (Interface Segregation Principle)**

Клієнти не повинні бути залежними від інтерфейсів, які вони не використовують.

Цей принцип вчить нас піклуватися про те, як ми пишемо наші інтерфейси. Коли ми пишемо інтерфейси, ми повинні подбати про додавання тільки тих методів, які там повинні бути. Якщо ми додаємо методи, які не повинні бути там, тоді класи, що реалізують інтерфейс будуть повинні реалізовувати зайві методи так само як і інші методи. Наприклад, якщо ми створюємо інтерфейс, званий Worker (Робочий) і додаємо метод lunch break (обідня перерва), тоді всі workers (робітники) будуть мати реалізацію цього зайвого методу. А що якщо робітник виявився роботом?

Інтерфейси містять методи, які не є специфічними для них, такі методи призводять до того, що інтерфейси називають забрудненими або жирними. Ми повинні уникати створення таких інтерфейсів.

**Принцип інверсії залежностей**

(Dependency Inversion Principle) - залежності всередині системи будуються на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не залежать від подробиць.

Даний принцип дуже важливий і гідний докладного розгляду.

**Принцип інверсії залежностей в деталях**

У протиставленні поганому дизайну, гарний дизайн архітектури повинен бути гнучким, стійким, і пристосованим до повторного використання. Чим

нижче взаємозв'язок компонентів додатка один з одним, тим вища гнучкість і мобільність всієї програми в цілому. Програми, що характеризуються високим коефіцієнтом мобільності, дозволяють застосовувати свої компоненти знову і знову для вирішення однотипних задач. Це веде до зниження дублювання коду. Такі програми складаються з великого набору досить дрібних компонентів, кожен з яких виконує малу частину роботи, але виконує її якісно. Дрібні компоненти набагато простіше тестувати, реалізовувати і супроводжувати.

Якщо ви дотримуєтеся принцип інверсії залежностей, то ваш код більш пристосований до змін і менше залежить від контексту виконання. Вірно і зворотне твердження. Якщо ваш додаток є хорошим прикладом вдалого дизайну архітектури, то він, в тій чи іншій мірі, дотримується принципу інверсії залежностей.

**Суть принципу**

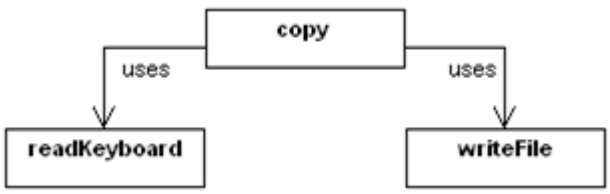
1. Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва типи модулів повинні залежати від абстракцій;

2. Абстракція не повинна залежати від реалізації. Реалізація повинна залежати від абстракції.

Традиційні методи розробки (наприклад, процедурне програмування) мають тенденцію до створення коду, в якому високорівневі модулі, як раз, залежать від низькорівневих. Це відбувається через те, що одна з цілей цих методів розробки - визначення ієрархії підпрограм, а отже і ієрархії викликів усередині модулів (високорівневі модулі викликають низькорівневі). Саме це є причиною низької гнучкості і закостенілості дизайну. При правильному використанні, ГО методики дозволяють обійти це обмеження.

Розглянемо приклад програми, яка копіює в файл дані, введені з клавіатури.

У нас є три модулі (у даному випадку це функції). Один модуль (іноді його називають сервіс) відповідає за читання з клавіатури. Другий - за виведення у файл. А третій високорівневий модуль об'єднує два низькорівневих модуля з метою організації їх роботи.



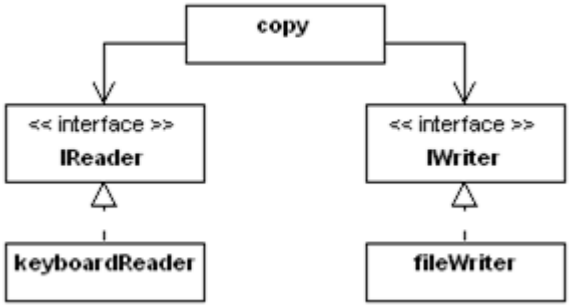

**Наш модуль copy може виглядати приблизно так.**

```
while (($ data = readKeyboard ())! == false)
{
writeFile (". / filename", $ data);
```

}

Низькорівневі модулі readKeyboard і writeFile володіють високою гнучкістю. Ми легко можемо використовувати їх у контексті відмінному від функції copy. Проте сама функція «copy» не може бути повторно використана в іншому контексті. Наприклад, для відправки даних з файлу системного оброблювача логів.

Використовуючи принцип інверсії залежностей, можна зробити модуль copy незалежним від об'єктів джерела і призначення даних. Для цього необхідно виробити абстракції для цих об'єктів, і зробити модулі залежними від цих абстракцій, а не один від одного.




*interface IReader*
*{*
*public function read ();*
*}*

*interface IWriter*
*{*
*public function write ($ data);*
*}*

Модуль **copy** повинен покладатися тільки на вироблені абстракції і не робити ніяких припущень з приводу індивідуальних особливостей об'єктів вводу / виводу.

*while (($ data = $ reader-> read ())! == false)*
*{*
*$ Writer-> write ($ data);*
*}*

Приблизно таким чином виглядає використання нашого модуля користувачем.

$ *Copier = new copier ();*

*/ / Копіювання даних з клавіатури в файл*
*$ Copier-> run (new keyboardReader (), new fileWriter ('. / Filename'));*

/ / Надсилання даних з файлу системного оброблювача логів

*$ Copier-> run (new fileReader ('. / Filename'), new syslogWriter ());*

Тепер модуль сору можна використовувати в різних контекстах копіювання. Зміна поведінки модуля-копіювальника досягається шляхом асоціації його з об'єктами інших класів (але які залежать від тих же абстракцій).

Незважаючи на простоту виконаних нами дій ми отримали дуже важливий результат. Тепер наш код володіє наступними якостями:

• модуль може бути використаний для копіювання даних в контексті відмінному від даного;

• ми можемо додавати нові пристрої введення / виведення не змінюючи при цьому модуль сору.

Таким чином, знизилася крихкість коду, підвищилася його мобільність і гнучкість.

# ПОВТОРНЕ ВИКОРИСТАННЯ КОДУ

**Повторне використання коду** (англ. code reuse) - методологія проектування комп'ютерних та інших систем, що полягає в тому, що система (комп'ютерна програма, програмний модуль) частково або повністю повинна складатися з частин, написаних раніше компонентів і / або частин іншої системи. Повторне використання - основна методологія, яка застосовується для скорочення трудовитрат при розробці складних систем.

Найпоширеніший випадок повторного використання коду - бібліотеки програм. Бібліотеки надають загальну досить універсальну функціональність, яка покриває обрану предметну область. Приклади: Бібліотека функцій для роботи з комплексними числами, бібліотека функцій для роботи з 3D-графікою, бібліотека для використання протоколу TCP / IP, бібліотека для роботи з базами даних. Розробники нової програми можуть використовувати існуючі бібліотеки для вирішення своїх завдань.

Повторне використання коду за межами одного проекту практично неможливо, якщо у вас немає розробленого проектного каркаса [framework]. У різних проектах різні набори сервісів, що і ускладнює повторне використання об'єкта.

Розробка проектного каркаса віднімає багато сил і часу. Але навіть якщо з якихось причин ви не створили собі подібної системи, існує декілька прийомів заохочення повторного використання коду.

# FRAMEWORK-СИСТЕМИ

## Вступ

Що таке framework-система? Навіщо вона Вам потрібна? У чому вона Вам може допомогти? А в чому ні? У цьому документі я спробую дати відповіді на ці питання.

Інтернет технології за останні десять років зробили дуже великий крок вперед, ставши полігоном для ведення бізнесу та електронної комерції. Розвиток глобальної мережі спричинило за собою розвиток інтернет-додатків. Якщо раніше сайти були не більше ніж «оголошеннями на заборі», то тепер це повноцінні програми, здатні виконувати завдання по автоматизації збору даних, обробки даних та надання інформації.

Перед сучасними web-розробниками постає дуже широкий спектр завдань. Це ефективна робота з реляційними базами даних, зберігання і обробка даних у форматі XML, побудова гнучких систем відображення інформації та багато іншого. Така множина завдань робить старі методи розробки додатків вкрай неефективними. Це призводить до думки про необхідність наявності спеціального інструментарію для web-розробника, який допоможе йому у вирішенні часто виникаючих проблем і завдань.

*Отже, що таке framework?*

Коли людина вирішує якесь завдання багато разів поспіль, вона вчиться вирішувати її швидко і ефективно! З точки зору web-програмування, framework-система (CMF-система) це платформа, що дозволяє вирішувати завдання, які постійно виникають при створенні інтернет-додатків. Не варто думати, що CMF-система - це просто набір модулів для вирішення різнотипних завдань, яких в Інтернеті безліч. Framework-система це щось більше. Це:

• Термінологія, яка дозволяє розробникам говорити лаконічно про дуже складні речі;

• Набір архітектурних стандартів, які система накладає на інтернет-додатки. Це знімає з розробників необхідність придумувати все з нуля і дозволяє більш ефективно використовувати код повторно;

• Модулі для вирішення завдань «першої необхідності», що дозволяють почати розробку з порожнього місця, не винаходячи нічого свого.

Framework-система для web-розробника відіграє таку ж роль, як саквояж з інструментами для монтажника. Навіть якщо монтажник зможе виконати свою роботу без свого саквояжа, він витратить більше часу, а якість виконаної роботи буде на порядок нижче. Аналогічна ситуація спостерігається в процесі створення інтернет-додатків.

### CMF і CMS

Розглядаючи поняття framework-системи, не можна обійти стороною, поняття системи управління контентом. Дуже часто поняття CMF (Content Management Framework) плутають з поняттям CMS (Content Management System). Це невірно, тому що це принципово різні речі.

**CMF-системи** не можна порівнювати з CMS-системами! Це головне правило, яке дуже часто порушують розробники при обговоренні питань пов'язаних з розробкою та використанням CMF-систем. CMF і CMS різні поняття, незважаючи на їх схожість.

**CMS-система** - це набір модулів для швидкого створення сайтів. На відміну від CMF, CMS-система - це є завершений продукт, який орієнтований, в першу чергу, не на програмістів, а на користувачів не знайомих з премудростями створення інтернет-додатків. CMS-система (дуже часто її називають «движок сайту») дозволяє за лічені години створити сайт або портал який складається з обмеженого набору готових модулів (новини, гостьова книга, форум). В більшості своїй, CMS-системи створюються без урахування їх подальшого зростання. Підсумок - відсутність жорсткої внутрішньої архітектури системи. Це істотно ускладнює процес супроводження проекту.

Якщо вам достатньо можливостей CMS-системи, то, швидше за все, Ви будете задоволені. Однак якщо перед Вами постане питання про зміну дизайну або розширення можливостей програми, то, в більшості випадків, Вам доведеться вдатися за допомогою до кваліфікованих програмістів. І, можливо, навіть їм буде не просто розібратися в цій CMS-системі. Прочитавши наступний параграф, Ви зрозумієте, чому в цьому абзаці так багато «можливо» і «швидше за все».

Вищесказане відноситься до «чистих» CMS-систем. Тобто до CMS-систем, які написані з нуля на порожньому місці. На щастя, ніхто не заважає використовувати вигоди обох типів систем. Останнім часом в Інтернеті почали з'являтися CMF / CMS-системи. Ці системи являють собою CMS-систему, створену на фундаменті framework'а. Вигоди очевидні. Детермінована внутрішня архітектура, яка в більшості випадків документована і розвинені механізми абстракції, які не залежать від CMS-утворюючих модулів. Супроводжувати проект, створений на основі CMF / CMS-системи, на порядок простіше, ніж проект, створений на основі «чистої» CMS-системи. Це пояснюється тим, що в першому випадку, при створенні CMS-системи, програмістам доводиться виконувати ряд вимог, які диктує framework. Завдяки цьому CMS-система набуває яскраво виражену архітектуру, як і всі додатки створюються за допомогою CMF-системи.

Якщо CMS-система являє собою закінчений продукт, то CMF-система - це набір інструментів, за допомогою яких, можна створити абсолютно будь-який продукт, зокрема і CMS-систему. Так як framework-система - це набір інструментів, то для її використання потрібні програмісти, які можуть з цими інструментами працювати. З цим пов'язаний ще один момент, характерний для CMF - навчання персоналу для роботи з CMF-системою.

Продукти CMF-системи (додатки, написані на її основі) відрізняються індивідуальністю та високим рівнем адаптації до конкретної ситуації, тому що вони є програмними рішеннями, призначеними для вирішення конкретного кола завдань у конкретному контексті. За допомогою CMF можна створювати будь-які інтернет-додатки, починаючи гостьовими книгами, закінчуючи інтернет-магазинами і веб-сервісами.

Маючи фахівців, які знають архітектуру використовуваної CMF-системи, стає можливим, відносно легко, розширювати можливості системи, проводити аудити безпеки і т.д.

## Архітектура CMF-системи

У попередніх розділах вже було сказано дуже багато про архітектуру. Вам може здатися, що архітектурні стандарти абсолютно не потрібні. Але необхідно розуміти, що подібна «диктатура» не має на меті обмеження програміста у прийнятті рішень. Навпаки це зроблено для забезпечення максимальної гнучкості архітектури та можливості її безболісної зміни. Безумовно, в жертву таким якостям доводиться приносити простоту та прозорість системи.

Питання архітектури дуже складні за своєю суттю, і навіть багато фахівців не можуть сказати за п'ять хвилин нічого зрозумілого із приводу якихось конкретних рішень. Але, незважаючи на таку велику контекстну залежність архітектури від типу додатка, існують добре вивчені і перевірені варіанти вирішення архітектурних проблем. Ці варіанти рішення носять назву «шаблони проектування» (Design Patterns). У тому чи іншому вигляді шаблони проектування можуть бути застосовані у всіх додатках. Виявлення оптимальних реалізацій шаблонів становить невід'ємну частину роботи над framework-системою (я б сказав, що справжня framework-система просякнута духом патернів проектування).

Шаблони проектування існують для всіх основних типів завдань, що виконуються CMF-системою. Рішення цих завдань вимагають продуманої стандартизації (зрозуміло, в рамках проекту). Таких завдань декілька:

- Обробка запиту ІС;
- Організація предметної області ІС;
- Організація подання ІС;
- Організація допоміжних підсистем;

Завдання, які я виділив, занадто умовні, що б вважати їх формальним списком завдань framework-системи. Цей список наведений, що б Ви могли зрозуміти, в яких напрямках розробники концентрують свої зусилля.

**Обробка запиту**

Підсистема обробки запиту зіставляє запит клієнта з дією, що виконується системою. Запити до системи можуть бути досить «різношерстими». Вони відрізняються як по вигляду, так і за смисловим навантаженням. Це залежить від типу додатка. Самі механізми зіставлення і їх дії можуть змінюватися під час супроводження проекту. Ці вимоги диктують розробникам CMF-системи необхідність створення зручного механізму аналізу та обробки запитів. У разі якщо розробники справляються зі своїм завданням, то додатки, побудовані на базі їх framework-системи, будуть красивими і легко запам'ятовуються адресами кшталт «http://www.server.com/news/2005-02-03» замість «http : / / www.server.com/index.php?module=news&action=show&date=2005-02-03 ». Безумовно, краса запиту не єдине якість, якого домагаються розробники. Гнучкі механізми зіставлення запиту з дією грають дуже важливу роль, так це дуже зміна частина системи.

**Організація предметної області**

У кожної інформаційної системи є предметна область. Це набір термінів, об'єктів і правил, якими оперує додаток. Організація предметної області, одна з найскладніших завдань, яка сьогодні стоїть перед розробниками. У переважній більшості випадків функціонування предметної області забезпечують реляційні бази даних і об'єктно-орієнтовані технології відображення. Реляційний і об'єктно-орієнтований підхід геніальні окремо. Проте їх композиція, при невмілому поводженні, перетворює архітектуру інформаційної системи в купу мотлоху, розібратися в якій буде важко навіть досвідченому фахівцеві.

**Організація подання**

**Уявлення** - це підсистема відображення даних. З її допомогою логіка предметної області відокремлюється від логіки відображення даних. Уявлення - це найстабільніша частина інформаційної системи. Відображення даних може мінятися дуже часто, на відміну від самих даних і методів їх обробки. Тому framework-система повинна надати зручні та гнучкі механізми роботи з логікою відображення. Для вирішення цього завдання використовуються шаблонні системи , чиє завдання полягає у відділенні логіки відображення і укладання її в окремі файли (шаблони відображення), які можна редагувати окремо від усіх інших частин системи. Завдяки цьому, роботу над проектом можна ефективно

розпаралелити (Організація предметної області → програміст + адміністратор БД, Організація подання → верстальник + дизайнер).

**Організація допоміжних підсистем**

Під допоміжними підсистемами мається на увазі набір архітектурних рішень, покликаних полегшити працю програміста. Сюди можна віднести реалізації патернів загального призначення, які безпосередньо не відносяться до інших підсистем. Зокрема до допоміжних підсистем відносяться такі поняття як резолверів, хендла, різний реєстр (и), спостерігачі і т.д. Ці речі можуть бути використані в будь-якій іншій підсистемі для вирішення виникаючих проблем.

Наприклад, патерн singletone (одиночка) може бути використаний для підтримки декількох інстанцій об'єкта в одиничному екземплярі. Це завдання носить суто допоміжний характер і не може бути віднесено безпосередньо до рівня бізнес-логіки або будь-якого іншого.

Проте не варто недооцінювати важливість прийняття рішень по відношенню до цієї підсистемі. Від того наскільки зручними та ефективними будуть реалізації допоміжних патернів, залежить те, наскільки зручно буде програмувати інші підсистеми, і наскільки ефективно вони будуть працювати. Код, написаний у цій підсистемі, багато в чому визначає код, який буде писати програміст, що користується цим framework'ом.

**Маленькі бібліотеки**

Один з ворогів повторного використання коду - той факт, що люди не складають з свого коду бібліотеки. Клас багаторазового використання може бути похований у директорії однієї з програм і може ніколи не випробувати хвилюючого почуття реінкарнації в новому проекті. І тільки тому, що програміст не захотів винести цей клас (або класи) у бібліотеку.

Одна з причин трагедії: люди не люблять маленькі бібліотеки. Є в маленьких бібліотеках щось таке, що люди вважають неправильним. Придушіть в собі це почуття. Комп'ютеру абсолютно все одно, скільки у вас бібліотек.

Якщо ви написали код, який можна використовувати повторно, але він не вписується в вашу бібліотеку, створіть нову.

**Тримайте свою базу бібліотек [репозиторій]**

Більшість компаній не має ніякого поняття, який код у них є. І більшість програмістів до цих пір не повідомляють про те, що вони зробили і не цікавляться тим, що вже написано. Репозиторії покликані змінити ситуацію на краще.

В ідеальному світі програміст міг би зайти на сайт, подивитися по каталогу або пошуком знайти потрібний пакет бібліотек і закачати собі. Якщо

ви можете налагодити таку систему, в якій програмісти на добровільній основі будуть підтримувати базу вихідників - це прекрасно. Якщо ви заведете бібліотекаря, який відстежує коефіцієнт повторного використання, то це просто розкішно.

Інший спосіб - *автоматична генерація сховища з початкових кодів*. Досягається подібний ефект через використання стандартних заголовків для класів, методів, бібліотек і різних підсистем. Такі заголовки служать водночас технічним керівництвом і пунктами в списку репозиторія.

**Файли, що включаються**

Можливість повторного використання існуючого коду є дуже важливим, тому що може зберегти час і гроші і сприяти узгодженості. Припустимо, що сайт Web містить текстове меню, яке повторюється на кожній сторінці. Замість повторного кодування меню буде значно легше закодувати його один раз і динамічно включати вміст меню на кожну з окремих сторінок Web. Це можна зробити за допомогою так званого серверного включення файлу.

Файли, що включаються можуть містити будь-який код XHTML або PHP і зазвичай зберігаються з розширенням. Inc, хоча можна використовувати також розширення. Php,. Txt, або. Htm. Вміст файлу, що включається кодується один раз і включається в будь-яку необхідну кількість сторінок PHP. Якщо під файлом, що включається робиться зміна, то оновлення автоматично відбивається на всіх сторінках PHP, які посилаються на цей файл.

Нижче показаний приклад типового файлу, що включається, що містить інформацію про заголовок сторінки.

Header.inc

<h3> Welcome to WebBooks.Com </ h3>

Цей приклад показує файл, що включається з ім'ям header.inc. Файл містить текст "Welcome to WebBooks.Com", оточений тегом XHTML <h3>. Він створює заголовок третього рівня, який можна тепер включати на всі сторінки, які складають сайт WebBooks. Після створення файлу, що включається, його можна включити в сторінку PHP за допомогою однієї з наступних функцій:

require (ім'я_файлу) - включає і перевіряє вказаний файл

include (ім'я_файлу) - інший спосіб підключення файлів

У наступному прикладі файл header.inc включається в існуючу сторінку PHP:

home.php

<? Php

```
require ('header.inc');
echo "<p> This is the WebBooks site ...</ p>";
?>
```

Функція require () викликає файл header.inc і перевіряє вміст файлу. Вміст потім виводиться, так ніби воно було частиною сторінки home.php. У цьому прикладі функція require () кодується вгорі сторінки, так як вона містить інформацію заголовка. Оператор require () можна, однак, включити в будь-якому місці документа PHP. Розташування функції require () визначає, де буде виводитися вміст файлу в контексті сторінки PHP.

Welcome to WebBooks.Com

This is the WebBooks site ...

Важливо відзначити, що при використанні файлів, що включаються, які містять конфіденційну інформацію, таку, як паролі або інформацію про користувача, файли повинні зберігатися з використанням розширення. Php, а не. Inc або іншого нестандартного розширення. Файли, які застосовують нестандартні розширення файлів, можуть завантажуватися з сервера Web, а їх вміст можна переглядати як звичайний текст. Використання розширення. Php гарантує, що клієнт не зможе побачити вихідний код, сервер поверне тільки код XHTML.

**Використання функцій**

Функції використовуються для розбиття великих блоків коду на менші, більш керовані одиниці. Міститься всередині функції код виконує певне завдання і повертає значення. PHP містить два типи функцій - визначені користувачем (або створені програмістом) і внутрішні (вбудовані функції), які є частиною визначення мови PHP. Цей розділ присвячений створенню та застосуванню певних функцій користувача.

Певні функції користувача створюються за допомогою ключового слова function. Вони особливо корисні у великих програмах PHP, так як можуть містити блоки коду, які можуть викликатися чи використовуватися в програмі, що дозволяє уникнути повторного переписування коду. Далі представлений приклад простої визначеної користувачем функції PHP:

```
function AddNumbers ($ num1, $ num2)
{
echo "Це приклад функції PHP. Вона обчислює суму двох чисел і повертає
результат, що викликається у програмі ";

return $ num1 + $ num2;

}
```

Певні функції користувача можуть викликатися в будь-якому місці блоку коду PHP. У PHP функція виконується при використанні в коді її імені. Після виклику функція отримує всі передані їй значення у формі параметрів, виконує певні завдання і повертає значення, що викликає програма. Простий приклад показаний нижче.

```
<? Php
function AddNumbers ($ num1, $ num2)

{
return $ num1 + $ num2;
}
echo "Сума 5 і 2 дорівнює". AddNumbers (5,2);
?>
```

Проте певна на початку функція AddNumbers () викликається тільки пізніше в програмі. Виклик функції відбувається в операторі echo. Виводиться рядок "Сума 5 і 2 дорівнює". Ім'я функції з'єднується з рядком виведення, викликаючи тим самим функцію. Для функції передається два параметри - 5 і 2. Вони присвоюються параметрами функції $ num1 і $ num2. Параметри складаються, і викликається оператор return, щоб "повернути" значення або суму двох чисел в те місце в блоці коду PHP, який спочатку викликав функцію. Висновок результату показаний нижче:

Сума 5 і 2 дорівнює 7

Імена функцій слідують тим же правилам, що і змінні у PHP. Допустимі імена можуть починатися з букви або підкреслення, після чого може слідувати будь-які літери, цифри або підкреслення.

# ПАТЕРНИ ПРОЕКТУВАННЯ

**Загальні відомості**

В якості основи проектування інформаційних систем застосовуються "типові рішення" або "шаблони проектування" (Patterns).

**Шаблони проектування** (патерн, design pattern) - це багато разів застосовувана архітектурна конструкція, що надає рішення для загальної проблеми проектування в рамках конкретного контексту й описує значимість цього рішення.

Патерн не є закінченим зразком проекту, який може бути прямо перетворений в код, скоріше це опис або зразок для того, як вирішити завдання, таким чином, щоб це можна було використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони часто показують відносини і взаємодії між класами або об'єктами, без визначення того, які кінцеві класи чи об'єкти додатки будуть використовуватися. Алгоритми не розглядаються як шаблони, так як вони вирішують завдання обчислення, а не проектування.

У 1970-і роки архітектор Крістофер Олександр склав набір шаблонів проектування. В області архітектури ця ідея не отримала такого розвитку, як пізніше в області програмної розробки. Згідно з визначенням Крістофера Олександра: "Кожне типове рішення описує якусь повторювану проблему і ключ до її розгадки, причому таким чином, що ви можете користуватися цим ключем багаторазово, жодного разу не прийшовши до одного й того ж результату" .

У 1987 році Кент Бек і Вард Каннігем взяли ідеї Олександра та розробили шаблони відповідно до розробки програмного забезпечення для розробки графічних оболонок мовою Smalltalk.

У 1988 році Ерік Гамма почав писати докторську дисертацію при Цюріхському університеті про загальну переносимість цієї методики на розробку програм.

У 1989-1991 роках Джеймс Коплін трудився над розробкою ідіом для програмування на C + + та опублікував у 1991 році книгу Advanced C + + Idioms. У цьому ж році Ерік Гамма закінчує свою докторську дисертацію і переїжджає до США, де у співробітництві з Річардом Хелмом, Ральфом Джонсоном і Джоном Вліссідсом публікує книгу Design Patterns - Elements of Reusable Object-Oriented Software. У цій книзі описані 23 шаблона проектування. Також команда авторів цієї книги відома громадськості під назвою Банда чотирьох (Gang of Four, часто скорочується до GoF). Саме ця книга стала причиною зростання популярності шаблонів проектування.

Іншим видатним діячем у галузі проектування програмних систем, який підтримав використання патернів, є Мартін Фаулер, який написав книгу

"Архітектура корпоративних програмних додатків" (Patterns of Enterprise Application Architecture). Як зазначив Мартін Фаулер у своїй книзі "збираючись скористатися типовими рішеннями, не забувайте, що вони тільки відправна точка, а не пункт призначення".

У книзі Крейга Лармана "Застосування UML і шаблонів проектування" описано 9 шаблонів GRASP (General Responsibility Assignment Software Patterns, загальні зразки розподілу обов'язків) - патернів, використовуваних в об'єктно-орієнтованому проектуванні для вирішення спільних завдань за призначенням обов'язків класам і об'єктам. Кожен з них допомагає розв'язати певну проблему, що виникає при об'єктно-орієнтованому аналізі, і яка виникає практично в будь-якому проекті з розробки програмного забезпечення.

Головна користь кожного окремого шаблону полягає в тому, що він описує рішення цілого класу абстрактних проблем. Також той факт, що кожен шаблон має своє ім'я, полегшує дискусію про абстрактні структури даних (ADT) між розробниками, так як вони можуть посилатися на відомі шаблони. Таким чином, за рахунок шаблонів проводиться уніфікація термінології, назв модулів і елементів проекту.

Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову і знову.

Однак іноді шаблони консервують громіздку і малоефективну систему понять, розроблену вузькою групою. Коли кількість шаблонів зростає, перевищуючи критичну складність, виконавці починають ігнорувати шаблони і всю систему, з ними пов'язану. Нерідко шаблонами замінюється відсутність або недоліки документації, яка складна програмному середовищі.

Є думка, що сліпе застосування шаблонів з довідника, без осмислення причин і передумов виділення кожного окремого шаблону, уповільнює професійне зростання програміста. Люди, які дотримуються цієї думки, вважають, що знайомитися зі списками шаблонів треба тоді, коли "доріс" до них в професійному плані - і не раніше. Хороший критерій потрібного ступеня професіоналізму - виділення шаблонів самостійно, на підставі власного досвіду. При цьому, зрозуміло, знайомство з теорією, пов'язаної з шаблонами, корисно на будь-якому рівні професіоналізму і направляє розвиток програміста в правильну сторону. Сумніву піддається тільки використання шаблонів "за довідником".

Шаблони можуть пропагувати погані стилі розробки додатків, і часто сліпо застосовуються.

**Шаблони проектування класифікують наступним чином:**

**Патерни проектування класів / об'єктів**

- ✓ ***Структурні патерни проектування класів / об'єктів***

- Адаптер (Adapter) - GoF
- Декоратор (Decorator) або Оболонка (Wrapper) - GoF
- Заступник (Proxy) або Сурогат (Surrogate) - GoF
- Інформаційний експерт (Information Expert) - GRASP
- Компонувальник (Composite) - GoF
- Міст (Bridge), Handle (описувач) або Тіло (Body) - GoF
- Низька зв'язаність (Low Coupling) - GRASP
- Пристосуванець (Flyweight) - GoF
- Стійкий до змін (Protected Variations) - GRASP
- Фасад (Facade) – GoF

✓ о ***Патерни проектування поведінки класів / об'єктів***

- Інтерпретатор (Interpreter) - GoF
- Ітератор (Iterator) або Курсор (Cursor) - GoF
- Команда (Command), Дія (Action) або Транзакція (Транзакція) - GoF
- Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model - GoF
- Не розмовляйте з невідомими (Don't talk to strangers) - GRASP
- Відвідувач (Visitor) - GoF
- Посередник (Mediator) - GoF
- Стан (State) - GoF
- Стратегія (Strategy) - GoF
- Зберігач (Memento) - GoF
- Ланцюжок обов'язків (Chain of Responsibility) - GoF
- Шаблонний метод (Template Method) - GoF
- Високе зачеплення (High Cohesion) - GRASP
- Контролер (Controller) - GRASP
- Поліморфізм (Polymorphism) - GRASP
- Штучний (Pure Fabrication) - GRASP
- Перенаправлення (Indirection) - GRASP

✓ ***Твірні патерни проектування***

- Абстрактна фабрика (Abstract Factory, Factory), ін. назва Інструментарій (Kit) - GoF
- Одинак (Singleton) - GoF
- Прототип (Prototype) - GoF
- Творець примірників класу (Creator) - GRASP
- Будівельник (Builder) - GoF
- Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) - GoF

**Архітектурні системні патерни**

✓ ***Структурні патерни***

- Репозиторій
- Клієнт / сервер
- об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper)
- Багаторівнева система (Layers) чи абстрактна машина
- Потоки даних (конвеєр або фільтр)

✓ ***Патерни управління***

- Патерни централізованого управління
- Виклик - повернення (сценарій транзакції - окремий випадок)
- Диспетчер
- Патерни управління, засновані на подіях
- Передача повідомлень
- Керування перериваннями
- Патерни, що забезпечують взаємодію з базою даних
- Активний запис (Active Record)
- Одиниця роботи (Unit Of Work)
- Завантаження на вимогу (Lazy Load)
- Колекція об'єктів (Identity Map)
- Безліч записів (Record Set)
- Успадкування з однією таблицею (Single Table Inheritance)
- Успадкування з таблицями для кожного класу (Class Table Inheritance)
- Оптимістичне автономне блокування (Optimistic Offline Lock)
- Відображення з допомогою зовнішніх ключів
- Відображення з допомогою таблиці асоціацій (Association Table Mapping)
- Песимістичне автономне блокування (Pessimistic Offline Lock)
- Поле ідентифікації (Identity Field)
- Перетворювач даних (Data Mapper)
- Збереження сеансу на стороні клієнта (Client Session State)
- Збереження сеансу на стороні сервера (Server Session State)
- Шлюз запису даних (Row Data Gateway)
- Шлюз таблиці даних (Table Data Gateway)
- Патерни, призначені для представлення даних у Web
- Модель-представлення-контролера (Model View Controller)
- Контролер сторінок (Page Controller)
- Контролер запитів (Front Controller)
- Представлення за шаблоном (Template View)
- Представлення з перетворенням (Transform View)

- Двоетапне подання (Two Step View)
- Контролер додатка (Application Controller)

✓ ***Патерни інтеграції корпоративних інформаційних систем***

- Структурні патерни інтеграції
- Взаємодія "точка - точка"
- Взаємодія "зірка" (інтегруюча середа)
- Змішаний спосіб взаємодії
- Патерни за методом інтеграції
- Інтеграція систем за даними (data-centric)
- Функціонально-центричний (function-centric) підхід
- Об'єктно-центричний (object-centric)
- Інтеграція на основі єдиної понятійної моделі предметної області (concept-centric)
- Патерни інтеграції за типом обміну даними
- Файловий обмін
- Загальна база даних
- Віддалений виклик процедур
- Обмін повідомленнями

Також на сьогоднішній день існує ряд інших шаблонів:

• *Carrier Rider Mapper*, надання доступу до збереженої інформації;

• аналітичні шаблони, описують основний підхід для складання вимог для програмного забезпечення (requirement analysis) до початку самого процесу програмної розробки;

• комунікаційні шаблони, описують процес спілкування між окремими учасниками / співробітниками організації;

• організаційні шаблони, описують організаційну ієрархію підприємства / фірми;

• Анти-патерни (Anti-Design-Patterns) описують як не слід поступати при розробці програм, показуючи характерні помилки в дизайні і в реалізації;

Розглянемо докладніше деякі з патернів проектування.

Огляд патернів

Патерни проектування класів / об'єктів

***Структурні патерни (Structural)***

До структурних патернів відносяться:

• Адаптер (Adapter) - GoF;

• Декоратор (Decorator) або Оболонка (Wrapper) - GoF;

• Заступник (Proxy) або Сурогат (Surrogate) - GoF;

• Інформаційний експерт (Information Expert) - GRASP;

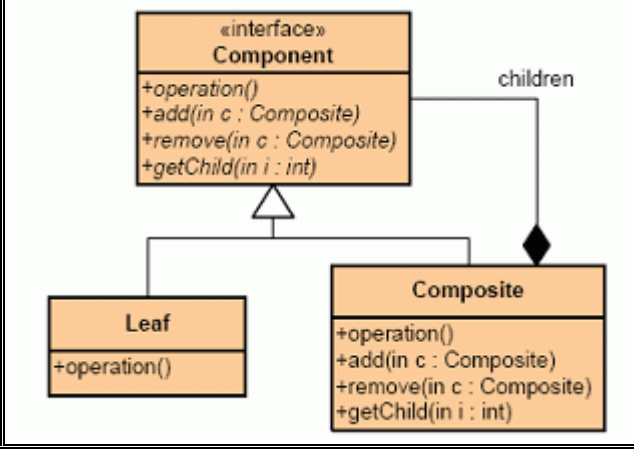
• Компонувальник (Composite) - GoF;

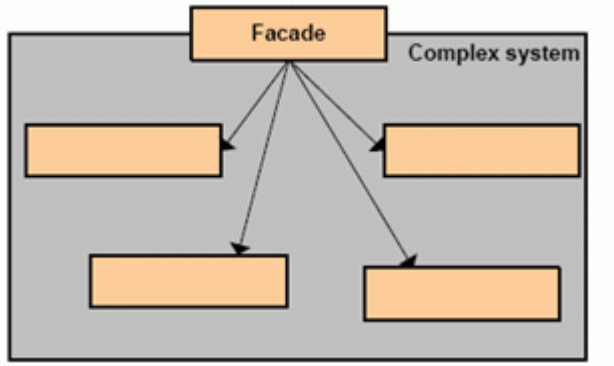
• Міст (Bridge), Handle (описувач) або Тіло (Body) - GoF;

• Низька зв'язаність (Low Coupling) - GRASP;
• Пристосуванець (Flyweight) - GoF;
• Стійкий до змін (Protected Variations) - GRASP;
• Фасад (Facade) - GoF.

Наведемо приклади 2-х даних патернів (табл. 1).

**Таблиця 1. Приклади структурних патернів класів/об'єктів**

<table>
<tr><td colspan="2">Компонувальник (Composite) – GoF</td></tr>
<tr><td>Проблема</td><td>Як обробляти групу або композицію структур об'єктів одночасно?</td></tr>
<tr><td>Рішення</td><td>Визначити класи для композитних і атомарних об'єктів таким чином, щоб вони реалізовували той самий інтерфейс.<br> 
</td></tr>
<tr><td colspan="2">Фасад (Facade) – GoF</td></tr>
<tr><td>Проблема</td><td>Як забезпечити уніфікований інтерфейс із набором розрізнених реалізацій або інтерфейсів, наприклад, з підсистемою, якщо небажано високе зв'язування із цією підсистемою або реалізація підсистеми може змінитися?</td></tr>
<tr><td>Рішення</td><td>Визначити одну точку взаємодії з підсистемою – фасадний об'єкт, що забезпечує загальний інтерфейс із підсистемою й покласти на нього обов'язок по взаємодії з її компонентами. Фасад – це зовнішній об'єкт, що забезпечує єдину точку входу для служб підсистеми. Реалізація інших компонентів підсистеми закрита, її не видно зовнішнім компонентам. Фасадний об'єкт забезпечує реалізацію патерна "Стійкий до змін" з погляду захисту від змін у реалізації підсистеми.</td></tr>
</table>

| | |
|---|---|
| |  |

*Патерни проектування поведінки (Behavioral)*

До поведінкових патернів належать:

• Інтерпретатор (Interpreter) - GoF;

• Ітератор (Iterator) або Курсор (Cursor) - GoF;

• Команда (Command), Дія (Action) або Транзакція (Транзакція) - GoF;

• Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model - GoF;

• Не розмовляйте з невідомими (Don't talk to strangers) - GRASP;

• Відвідувач (Visitor) - GoF;

• Посередник (Mediator) - GoF;

• Стан (State) - GoF;

• Стратегія (Strategy) - GoF;

• Зберігач (Memento) - GoF;

• Ланцюжок обов'язків (Chain of Responsibility) - GoF;

• Шаблонний метод (Template Method) - GoF;

• Високе зачеплення (High Cohesion) - GRASP;

• Контролер (Controller) - GRASP;

• Поліморфізм (Polymorphism) - GRASP;

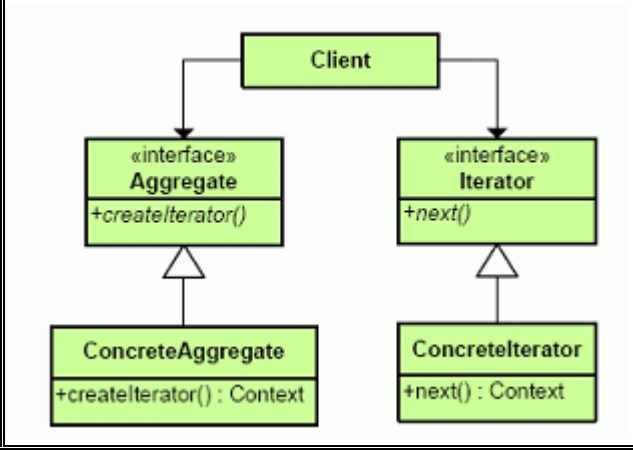
• Штучний (Pure Fabrication) - GRASP;

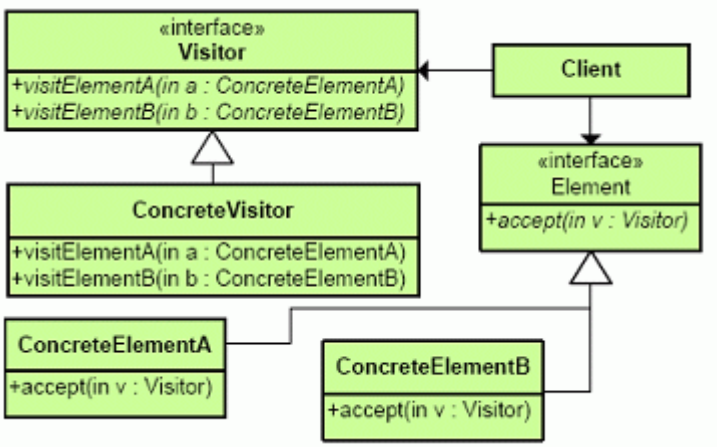
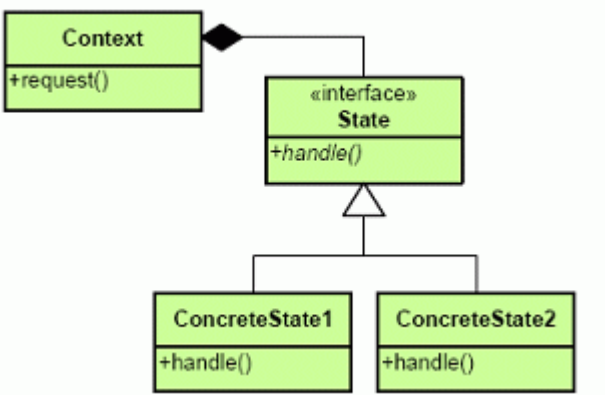
• Перенаправлення (Indirection) - GRASP.

Наведемо приклади 3-х даних патернів (табл. 2).

**Таблиця 2. Приклади поведінкових патернів класів/об'єктів**

| Ітератор (Iterator) або Курсор (Cursor) – GoF | |
|---|---|
| **Проблема** | Складений об'єкт, наприклад, список, повинен надавати доступ до своїх елементів (об'єктів), не розкриваючи їхню внутрішню структуру, причому перебирати список потрібно по-різному залежно від завдання. |
| **Рішення** | Створюється клас "Ітератор", який визначає інтерфейс для доступу і перебору елементів, "КонкретнийІтератор" реалізує інтерфейс класу "Ітератор" і стежить за поточною позицією при обході "Агрегат". "Агрегат" визначає інтерфейс для створення об'єкту - ітератора. "КонкретнийАгрегат" реалізує інтерфейс створення ітератора і |

| | |
|---|---|
| | повертає екземпляр класу "КонкретнийІтератор", "КонкретнийІтератор" відстежує поточний об'єкт в агрегаті і може обчислити наступний об'єкт при переборі.<br>Даний патерн підтримує різні способи перебору агрегату.<br>  <br> |
| **Відвідувач (Visitor) – Go** | |
| **Проблема** | Над кожним об'єктом деякої структури виконується операція. Визначити нову операцію, не змінюючи класи об'єктів. |
| **Рішення** | Клієнт, що використовує даний патерн, повинен створити об'єкт класу "КонкретнийВідвідувач", а потім відвідати кожен елемент структури. "Відвідувач", оголошує операцію "Відвідати" для кожного класу "КонкретнийЕлемент" (ім'я та сигнатура даної операції ідентифікують клас, елемент якого відвідує "Відвідувач" - тобто, відвідувач може звертатися до елемента прямо). "КонкретнийВідвідувач" реалізує всі операції, оголошені в класі "Відвідувач". Кожна операція реалізує фрагмент алгоритму, визначеного для класу відповідного об'єкта в структурі.<br>Клас "КонкретнийВідвідувач" надає контекст для цього алгоритму і зберігає його локальний стан. "Елемент" визначає операцію "Прийняти", яка приймає "Відвідувача" як аргумент, "КонкретнийЕлемент" реалізує операцію "Прийняти", яка приймає "Відвідувача" як аргумент. "СтруктураОб'єкта" може перерахувати свої аргументи і надати відвідувачеві високорівневий інтерфейс для відвідування своїх елементів.<br>Даний патерн логічно використовувати, якщо в структурі присутні об'єкти багатьох класів з різними інтерфейсами, і необхідно виконати над ними операції, що залежать від конкретних класів, або якщо класи, встановлюють структуру об'єктів, змінюються рідко, але нові операції над цією структурою додаються часто.<br>Даний патерн спрощує додавання нових операцій, об'єднує споріднені операції в класі "Відвідувач". |

| | |
|---|---|
| | У даному патерні утруднено додавання нових класів "КонкретнийЕлемент", оскільки потрібне оголошення нової абстрактної операції в класі "Відвідувач".<br>  |
| **Стан (State) – Go** | |
| **Проблема** | Варіювати поводження об'єкта залежно від його внутрішнього стану |
| **Рішення** | Клас "Контекст" делегує залежать від стану запити поточному об'єкту "КонкретнийСтан" (зберігає екземпляр підкласу "КонкретнийСтан", яким визначається поточний стан), і визначає інтерфейс, що представляє інтерес для клієнтів. " КонкретнийСтан" реалізує поведінку, асоційовану з якимось станом об'єкта "Контекст". "Стан" визначає інтерфейс для інкапсуляції поведінки, асоційованого з конкретним екземпляром "Контексту".<br>Даний патерн локалізує залежне від стану поведінку і ділить його на частини, що відповідають станам, переходи між станами стають явними.<br>  |

**Твірні патерни проектування**

*До породжувальних патернів належать*:

• Абстрактна фабрика (Abstract Factory, Factory) - GoF;

• Одинак (Singleton) - GoF;

• Прототип (Prototype) - GoF;

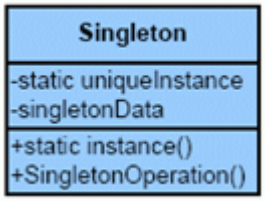
• Творець примірників класу (Creator) - GRASP;

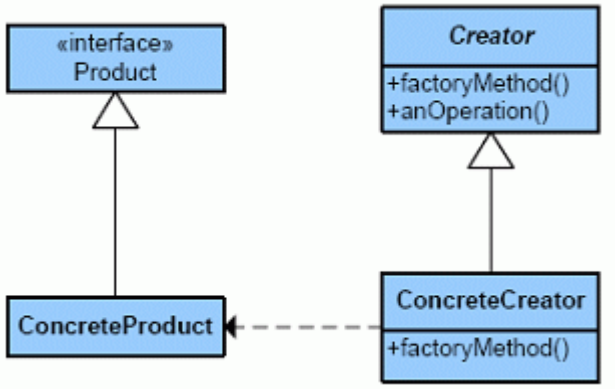
• Будівельник (Builder) - GoF;

• Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) - GoF.

Наведемо приклади 2-х даних патернів (табл. 3) .

**Таблиця 3. Приклади що породжують патерни класів/об'єктів**

| **Одинак (Singleton) – Go** | |
|---|---|
| **Проблема** | Який спеціальний клас повинен створювати "Абстрактну фабрику" і як одержати до неї доступ? Необхідний лише один екземпляр спеціального класу, різні об'єкти повинні звертатися до цього екземпляра через єдину точку доступу. |
| **Рішення** | Створити клас і визначити статичний метод класу, що повертає цей єдиний об'єкт. Розумніше створювати саме статичний екземпляр спеціального класу, а не оголосити необхідні методи статичними, оскільки при використанні методів екземпляра можна застосувати механізм спадкування й створювати підкласи. Статичні методи в мовах програмування не поліморфні й не допускають перекриття в похідних класах. Рішення на основі створення екземпляра є більше гнучким, оскільки згодом може знадобитися вже не єдиний екземпляр об'єкта.<br>  |
| **Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) – Go** | |
| **Проблема** | Визначити інтерфейс для створення об'єкту, але залишити підкласам рішення про те, який клас інстанціювати, тобто, делегувати інстанціювання підкласами. |
| **Рішення** | Абстрактний клас "Творець" оголошує Фабричний Метод, який повертає об'єкт типу "Продукт" (абстрактний клас, що визначає інтерфейс об'єктів, що створюються фабричним методом). "Творець" також може визначити реалізацію за замовчуванням Фабричного Методу, який повертає "КонкретнийПродукт". "КонкретнийТворець" заміщає Фабричний Метод, який повертає об'єкт "КонкретнийПродукт". "Творець" "покладається" на свої підкласи у визначенні Фабричного Методу, що повертає об'єкт "КонкретнийПродукт". Даний патерн позбавляє проектувальника від необхідності вбудовувати в код залежать класів від програми. Однак при застосуванні даного патерну виникає додатковий рівень підкласів. |

| | |
|---|---|
| |  |

Архітектурні системні патерни

*Структурні патерни*

До структурних патернів належать:

• Репозиторій;

• Клієнт / сервер;

• об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper);

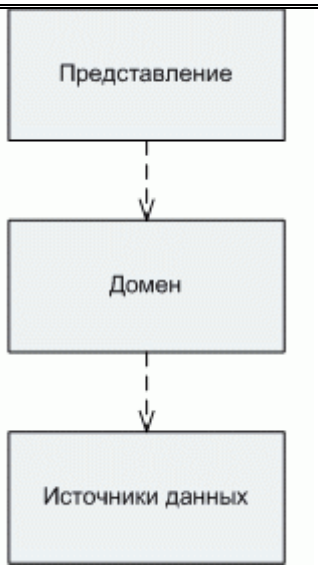
• Багаторівнева система (Layers) чи абстрактна машина;

• Потоки даних (конвеєр або фільтр).

Наведемо приклад одного із даних патернів (табл. 4).

Таблиця 4. Приклади стуктурних патернів архітектури

| **Багаторівнева система (Layers) або абстрактна машина** | |
|---|---|
| **Опис** | Відповідно до патерна "Багаторівнева система" структурні елементи системи організуються в окремі рівні з взаємопов'язаними обов'язками таким чином, щоб на нижньому рівні розташовувалися низькорівневі служби та служби загального призначення, а на більш високих - об'єкти рівня логіки додатка. При цьому взаємодія і зв'язування рівнів відбувається зверху вниз. Зв'язування об'єктів знизу вгору слід уникати.<br>На малюнку показані типові рівні логічної архітектури системи. |




Шар подання охоплює все, що має відношення до спілкування користувача з системою. До основних функцій шару подання відноситься відображення інформації й інтерпретація користувачем команд з перетворенням їх у відповідні операції в контексті домену (бізнес - логіка) і джерела даних.

Джерело даних - підмножина функцій, що забезпечує взаємодію зі сторонніми системами, які виконуються.

На відміну від архітектурного патерну "Клієнт - сервер", шари зовсім не обов'язково повинні розташовуватися на різних машинах.

Багаторівнева система може бути розроблена покрокова (Ітеративний).

Недоліками даного патерну є:

• Зміна вихідного коду тягне за собою переробку всіх елементів системи, оскільки всі елементи системи тісно пов'язані один з одним.

• Логіка програми тісно пов'язана з інтерфейсом користувача - важко міняти інтерфейс або принципи реалізації логіки. Через високу пов'язаність, роботу з реалізації системи складно розділити між розробниками і, крім того, складно модифікувати функції додатку або переходити на нові технології.

## Патерни управління

До патернів управління відносяться :

- ✓ Патерни централізованого управління
  - Виклик - повернення (сценарій транзакції - окремий випадок)
  - Диспетчер
- ✓ Патерни управління, засновані на подіях
  - Передача повідомлень
  - Керування перериваннями

- ✓ Патерни, що забезпечують взаємодію з базою даних
  - Активний запис (Active Record)
  - Одиниця роботи (Unit Of Work)
  - Завантаження на вимогу (Lazy Load)
  - Колекція об'єктів (Identity Map)
  - Безліч записів (Record Set)
  - Успадкування з однією таблицею (Single Table Inheritance)
  - Успадкування з таблицями для кожного класу (Class Table Inheritance)
  - Оптимістичне автономне блокування (Optimistic Offline Lock)
  - Відображення з допомогою зовнішніх ключів
  - Відображення з допомогою таблиці асоціацій (Association Table Mapping)
  - Песимістичне автономне блокування (Pessimistic Offline Lock)
  - Поле ідентифікації (Identity Field)
  - Перетворювач даних (Data Mapper)
  - Збереження сеансу на стороні клієнта (Client Session State)
  - Збереження сеансу на стороні сервера (Server Session State)
  - Шлюз запису даних (Row Data Gateway)
  - Шлюз таблиці даних (Table Data Gateway)

У даній лекції приклади патернів управління розглядатися не будуть.

### Патерни, призначені для представлення даних у Web

До патернів, призначених для представлення даних у Web, відносяться:

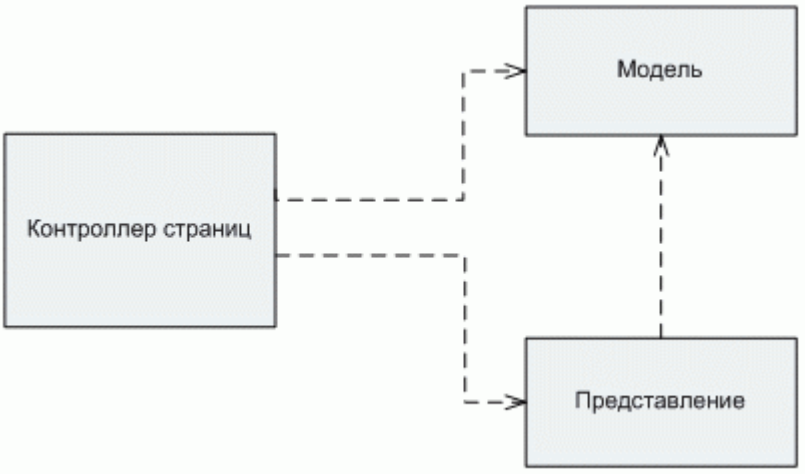
- Модель-представлення-контролера (Model View Controller);
- Контролер сторінок (Page Controller);
- Контролер запитів (Front Controller);
- Представлення за шаблоном (Template View);
- Представлення з перетворенням (Transform View);
- Двоетапне подання (Two Step View);
- Контролер додатка (Application Controller).

Наведемо приклад 4-х патернів (табл. 5) [18].

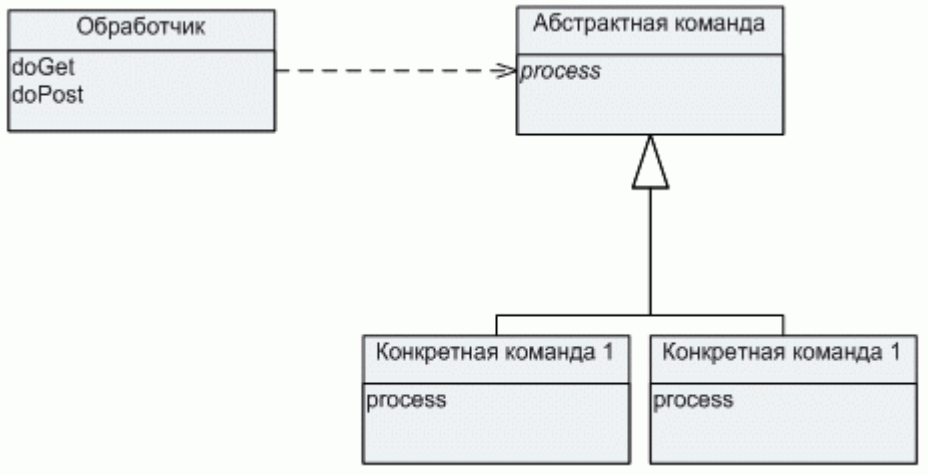
Таблиця 5. Приклади патернів, призначених для представлення даних у Web

| **Модель-подання-контролера (Model View Controller)** |
|---|

| | |
|---|---|
| **Опис** |  <br>Типове рішення модель-подання-контролера має на увазі виділення трьох окремих ролей. **Модель** - це об'єкт, що дає деяку інформацію про домен. У моделі немає візуального інтерфейсу, вона містить у собі всі дані і поведінку, не пов'язані з призначеним для користувача інтерфейсом. В об'єктно-орієнтованому контексті найбільш "чистою" формою моделі є об'єкт моделі предметної області. В якості моделі можна розглядати і сценарій транзакції, якщо він не містить в собі ніякої логіки, пов'язаної з призначеним для користувача інтерфейсом. Подібне визначення не дуже розширює поняття моделі, однак повністю відповідає розподілу ролей у розглянутому типовому рішенні.<br>Представлення відображає вміст моделі засобами графічного інтерфейсу. Таким чином, якщо модель - це об'єкт покупця, відповідне подання може бути фреймом з купою елементів управління або HTML-сторінкою, заповненою інформацією про покупця. Функції подання полягають тільки у відображенні інформації на екрані. Всі зміни інформації обробляються третім "учасником" системи - контролером. Контролер одержує вхідні дані від користувача, виконує операції над моделлю і вказує подання на необхідність відповідного оновлення. У цьому плані графічний інтерфейс можна розглядати як сукупність подання та контролера.<br>Говорячи про типове рішення модель-подання-контролер, не можна не підкреслити два основні типи поділу: відділення подання від моделі та відділення контролера від подання.<br>Відділення подання від моделі - це один з фундаментальних принципів проектування програмного забезпечення. Наявність такого поділу дуже важливо з ряду причин:<br>• Представлення і модель відносяться до абсолютно різних сфер |

| | |
|---|---|
| | програмування.<br>• Користувачі хочуть, щоб, залежно від ситуації, одна і та ж інформація могла бути відображена різними способами.<br>• Об'єкти, що не мають візуального інтерфейсу, набагато легше тестувати, ніж об'єкти з інтерфейсом.<br>Ключовим моментом у відділенні подання від моделі є спрямування залежностей: представлення залежить від моделі, але модель не залежить від подання. Це означає, що зміна уявлення не вимагає зміни моделі.<br>Відділення контролера від подання. Класичним прикладом необхідності такого поділу є підтримка редагованої і не редагованої поведінки. Цього можна досягти за наявності одного подання і двох контролерів (для двох варіантів використання), де контролери є стратегіями, використовуваними поданням. Тим часом на практиці в більшості систем кожному поданню відповідає тільки один контролер, тому поділ між ними не проводиться. Про це рішення згадали тільки з появою Web-інтерфейсів, де відділення контролера від подання виявилося надзвичайно корисним.<br>. |
| **Контролер сторінок (Page Controller)** | |
| **Опис** |  <br>В основі контролера сторінок лежить ідея створення компонентів, які будуть виконувати роль контролерів для кожної сторінки Веб-сайту. На практиці кількість контролерів не завжди в точності відповідає кількості сторінок, оскільки інколи при натисканні на посилання відкриваються сторінки з різним динамічним вмістом. Якщо говорити більш точно, контролер необхідний для кожної дії, під яким мається на увазі клацання на кнопці або гіперпосиланні.<br>Контролер сторінок може бути реалізований у вигляді сценарію (сценарію CGI) або сторінки сервера (ASP, PHP, JSP і т.д.). Використання сторінки сервера зазвичай передбачає поєднання в одному файлі контролера сторінок та подання за шаблоном. Це добре для подання за |

| | |
|---|---|
| | шаблоном, але не дуже підходить для контролера сторінок, оскільки значно ускладнює правильне структурування цього компоненту. Дана проблема не настільки важлива, якщо сторінка застосовується тільки для простого відображення інформації. Тим не менш, якщо використання сторінки припускає наявність логіки, пов'язаної з отриманням даних користувача або вибором подання для відображення результатів, сторінка сервера може заповнитися кодом впровадженого сценарію.<br>Щоб уникнути подібних проблем, можна скористатися допоміжним об'єктом (helper object). При отриманні запиту сторінка сервера викликає допоміжний об'єкт для обробки всієї наявної логіки. У залежності від ситуації, допоміжний об'єкт може повернути управління первісної сторінки сервера або ж звернутися до іншої сторінки сервера, щоб вона виступила як представлення. У цьому випадку обробником запитів є сторінка сервера, проте велика частина логіки контролера укладена у допоміжному об'єкті.<br>Можливою альтернативою описаного підходу є реалізація обробника і контролера у вигляді сценарію. У цьому випадку під час вступу запиту веб-сервер передає управління сценарієм; сценарій виконує всі дії, покладені на контролер, після чого відображає отримані результати з допомогою потрібного подання.<br>**Нижче перераховані основні обов'язки контролера сторінок.**<br>• Проаналізувати адресу URL і витягти дані, введені користувачем у відповідні форми, щоб зібрати всі відомості, необхідні для виконання дії.<br>• Створити об'єкти моделі і викликати їх методи, необхідні для обробки даних.<br>• Визначити подання, яке повинно бути використано для відображення результатів, і передати йому необхідну інформацію, отриману від моделі.<br>Контролер сторінок не обов'язково повинен являти собою єдиний клас, зате всі класи контролерів можуть використовувати одні й ті ж допоміжні об'єкти. Це особливо зручно, якщо веб-сервер повинен мати кілька обробників, що виконують аналогічні завдання. У цьому випадку використання допоміжного об'єкту дозволяє уникнути дублювання коду.<br>При необхідності одні адреси URL можна обробляти за допомогою сторінок сервера, а інші - за допомогою сценаріїв. Ті адреси URL, у яких немає або майже немає логіки контролера, добре обробляти за допомогою сторінок сервера, тому що останні є простим механізмом, зручним для розуміння та внесення змін. Всі інші адреси, для обробки яких необхідна більш складна логіка, вимагають застосування сценаріїв. |
| **Контролер запитів (Front Controller)** | |

| | |
|---|---|
| **Опис** |  <br>Контролер запитів обробляє всі запити, що надходять до Веб-сайту, і зазвичай складається з двох частин: Веб-обробника та ієрархії команд. Веб-обробник - це об'єкт, який виконує фактичне отримання POST або GET-запитів, що надійшли на Веб-сервер. Він витягає необхідну інформацію з адреси URL і вхідних даних запиту, після чого вирішує, яку дію необхідно ініціювати, і делегує його виконання відповідній команді.<br>Веб-обробник зазвичай реалізується у вигляді класу, а не сторінки сервера, оскільки він не генерує жодних відгуків. Команди також є класами, а не сторінками сервера, більше того, їм не потрібно знати про наявність Веб-оточення, не дивлячись на те, що їм часто передається інформація з HTTP-запитів. У більшості випадків Веб-обробник - це досить проста програма, функції якої полягають у виборі потрібної команди.<br>Вибір команди може відбуватися статично або динамічно. Статичний вибір команди - це проведення синтаксичного аналізу адреси URL і застосування умовної логіки, а динамічний - витяг деякого стандартного фрагмента адреси URL і динамічне створення екземпляра класу команди.<br>Переваги статичного вибору команди перебувають у використанні явного коду, наявності перевірки часу компіляції і високої гнучкості можливих варіантів написання адрес URL. У свою чергу, використання динамічного підходу дозволяє додавати нові команди, не вимагаючи зміни Веб-обробника.<br>При динамічному виборі команд ім'я класу команди можна помістити безпосередньо на адресу URL або скористатися файлом властивостей, який буде прив'язувати адреси URL до імен класів команд. Зрозуміло, це потребує створення додаткового файлу властивостей, однак дозволить легко і невимушено змінювати імена класів, не переглядаючи всі наявні на сервері Веб-сторінки. |
| **Подання за шаблоном (Template View)** | |

| | |
|---|---|
| **Опис** |  <br>Основна ідея, що лежить в основі типового рішення уявлення за шаблоном, - вставка маркерів в текст готової статичної HTML-сторінки. При виклику сторінки для обслуговування запиту ці маркери будуть замінені результатами деяких обчислень (наприклад, результатами виконання запитів до бази даних). Подібна схема дозволяє створювати статичну частину сторінки за допомогою звичайних засобів, наприклад текстових редакторів, що працюють за принципом WYSIWYG, і не вимагає знання мов програмування. Для отримання динамічної інформації маркери звертаються до окремих програм.<br>Подання за шаблоном використовується цілим рядом програмних засобів. Таким чином, завдання полягає не стільки в тому, щоб розробити дане рішення самому, скільки в тому, щоб навчитися його ефективно використовувати і познайомитися з можливими альтернативами. |

Патерни інтеграції корпоративних інформаційних систем

**Структурні патерни інтеграції**

До структурних патернів інтеграції належать :

• Взаємодія "точка - точка";

• Взаємодія "зірка" (інтегруюча середовище);

• Змішаний спосіб взаємодії.

У даній лекції приклади структурних патернів інтеграції розглядатися не будуть.

**Патерни за методом інтеграції**

До патернів за методом інтеграції належать:

• Інтеграція систем за даними (data-centric);

• Функціонально-центричний (function-centric) підхід;

• Об'єктно-центричний (object-centric);

• Інтеграція на основі єдиної понятійної моделі предметної області (concept-centric).

**Патерни інтеграції за типом обміну даними**

До патернів інтеграції за типом обміну даними відносяться :

• Файловий обмін;

• Загальна база даних;

• Віддалений виклик процедур;

• Обмін повідомленнями.

У даній лекції приклади патернів інтеграції за типом обміну розглядатися не будуть.

## MODEL-VIEWER-CONTROLLER

Стандартна схема архітектури «Модель-Вид-Контролер» зображена на наступному малюнку: (схема запозичена з книги «Ajax in action» видавничого дому «Вільямс»)

Схема архітектури MVC


Розберемо по пунктах дану схему.

У шаблоні MVC, як випливає з назви, є три основних компоненти: **Модель, Представлення,** і **Контролер**.

Представлення відповідає за відображення інформації, що надходить із системи або в систему.

Модель є «суттю» системи і відповідає за безпосередні алгоритми, розрахунки тощо внутрішній устрій системи.

Контролер є сполучною ланкою між «поданням» і «моделлю» системи, за допомогою якого і існує можливість зробити поділ між ними. Контролер отримує дані від користувача і передає їх в «модель». Крім того, він отримує повідомлення від моделі, і передає їх в «подання».

Стосовно до інтернет-додатків існує думка, що частини контролера і подання об'єднані, тому що за відображення і одночасно за введення інформації відповідає браузер. З цим можна погодитися, а можна не погоджуватися і виділити контролер в окрему частину, що ми і зробимо.

Отже, домовимося:

**Представлення.** Модуль виведення інформації. Це може бути шаблонізатор або що-небудь подібне, мета якого є тільки в поданні інформації у вигляді HTML на основі будь-яких готових даних.

**Контролер**. Модуль управління введенням і виведенням даних. Даний модуль повинен стежити за переданими в систему даними (через форму, рядок запиту, cookie або будь-яким іншим способом) і на основі введених даних вирішити:

• Передавати чи їх у модель

• Вивести повідомлення про помилку і запросити повторне введення (змусити модуль уявлення оновити сторінку з урахуванням умов)

Крім того, контролер зобов'язаний визначати тип даних, отриманих від моделі (чи є це готовий результат, відсутність повідомлення про помилку) і передавати інформацію в модуль уявлення.

**Модель.** Модуль, що відповідає за безпосередній розрахунок чого-небудь на основі отриманих від користувача даних. Результат, отриманий цим модулем, повинен бути переданий в контролер, і не повинен містити нічого, що відноситься до безпосереднього висновку (тобто має бути представлений у внутрішньому форматі програми).

Досить складно з першого разу розібратися і зрозуміти. На це потрібен час і відповідний проект.

Але насправді нічого надскладного в цьому немає.

Уявімо собі як представлення будь-якого класу, який за допомогою шаблонізатора виводить результат чи повідомлення про помилку. На його вхід подається або масив з даними (об'єкт або що-небудь інше), або змінна, що містить текст з помилкою.

В якості контролера буде виступати клас, що виробляє всі необхідні перевірки коректності даних і генерує повідомлення про помилки. Перевірку даних доцільно помістити саме в клас контролера, їх використовують досить часто. Як варіант, можна просто успадковувати клас контролера від більш загального класу, що реалізує перевірку вхідних даних за заданими правилами. Або, якщо так буде зручніше, включити в клас контролера клас або серію функцій перевірки даних.

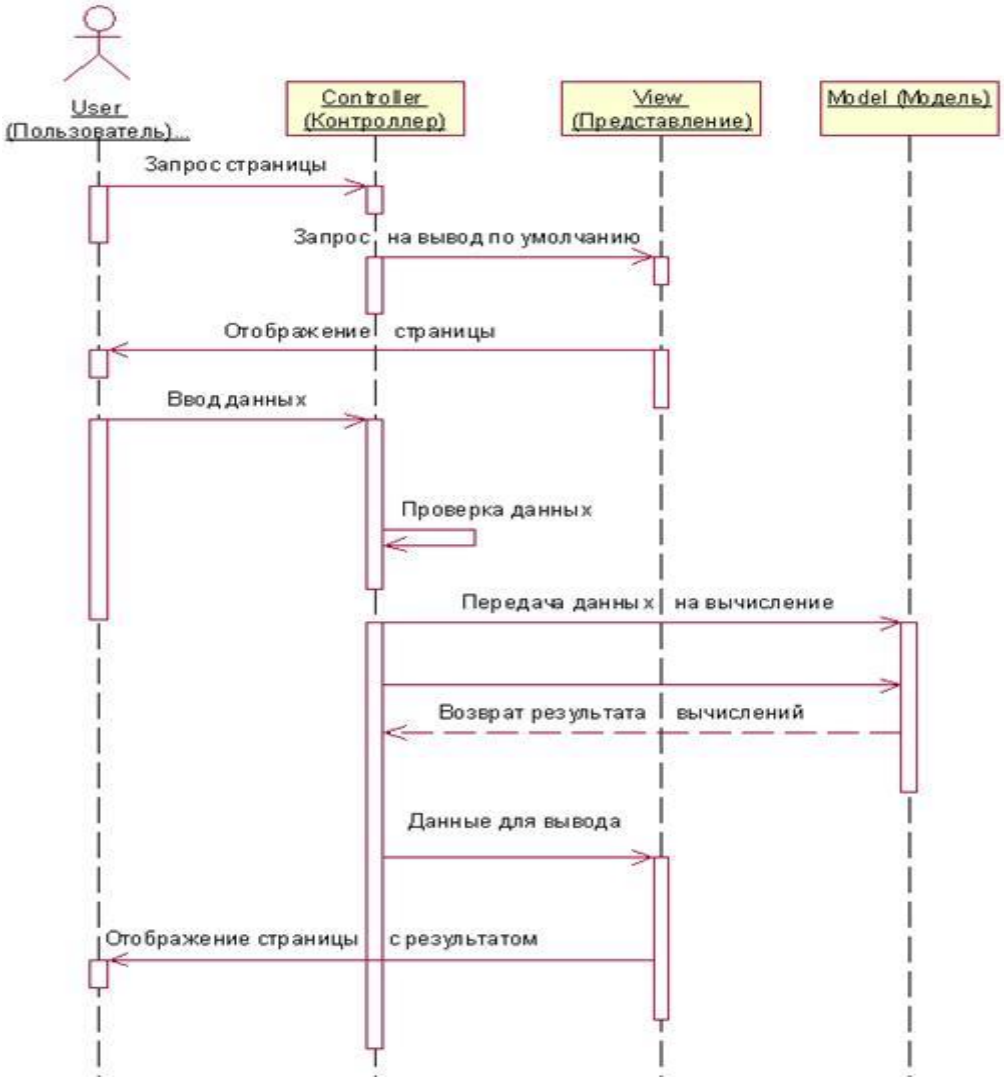
Цей же клас повинен передати дані, отримані в результаті роботи моделі, в клас представлення для виводу.

Одними словами схему потоків даних у цій архітектурі пояснити складно, тому звернемося до мови UML і до діаграми послідовностей зокрема (незначні відступи від UML, прийняті в діаграмах, полягають в тому, що в деяких випадках разом з іменами сутностей або об'єктів, дані переказують в дужках).

На цій діаграмі показана послідовність дій, а також послідовність переданих даних: від користувача, до користувача і між модулями.

Схема відображає типовий процес виведення форми, заповнення її користувачем і повернення користувачеві результатів. Ніяких помилок в даному випадку не відбувається.

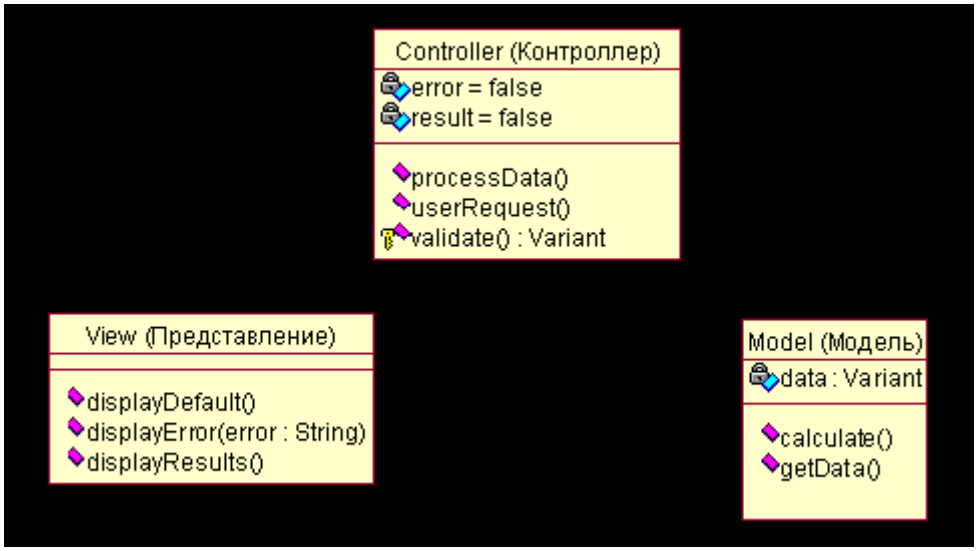
**Діаграма послідовностей**




Як видно з діаграми, звернення до моделі відбувається лише в разі надсилання користувачем вірних даних. На внутрішньому ж рівні додатку, модель відокремлена від подання та контролера. Контролер також відділений від моделі і уявлення, і його функція полягає в управлінні та перевірці.

Тепер спробуємо скласти діаграму класів для більшої наочності.

**Діаграма класів**




Діаграма класів містить три класи, по одному для кожного компонента архітектури MVC. Для зручності, вони так і названі: Model, View, Controller.

У поданні є три функції (хоча, цілком можливо обійтися тільки лише однією), які відповідають за відображення стану програми:

displayDefault () - висновок форми за замовчуванням.

displayError (error = false) - висновок форми з повідомленням про помилку,

displayResults () - висновок результатів обчислень

Контролер має не тільки методи, а й поля. З полями все просто: це помилка і результати обчислень. За замовчуванням їм задається значення false, що свідчить про те, що поки немає ні помилки, ні результатів.

Три методи, присутні в контролері, служать для управління та перевірки. Метод для перевірки (validate ()) є необов'язковим, і цілком може бути відсутнім, якщо ніяких перевірок не потрібно.

**Метод processData** () служить для виведення форми за замовчуванням, однак він включає також метод **userRequest** (), функціональність якого виконується лише в тому випадку, коли є введені користувачем дані. Саме метод **userRequest** () містить у собі функцію **validate** () (якщо дані не введені, отже, нема чого робити їх перевірку) і, крім того, повинен міститись виклик конструктора класу моделі.

У моделі може міститися будь-яка кількість полів і методів. Однак два методи повинні бути обов'язковими (або навіть один. Як зручніше буде).

**calculate** () - функція, яка виробляє основний розрахунок

**getData** () - функція, яка повертає дані результату.

Поділ функцій моделі швидше смислове. Цілком достатньо створити один метод, який буде і рахувати, і повертати результат.

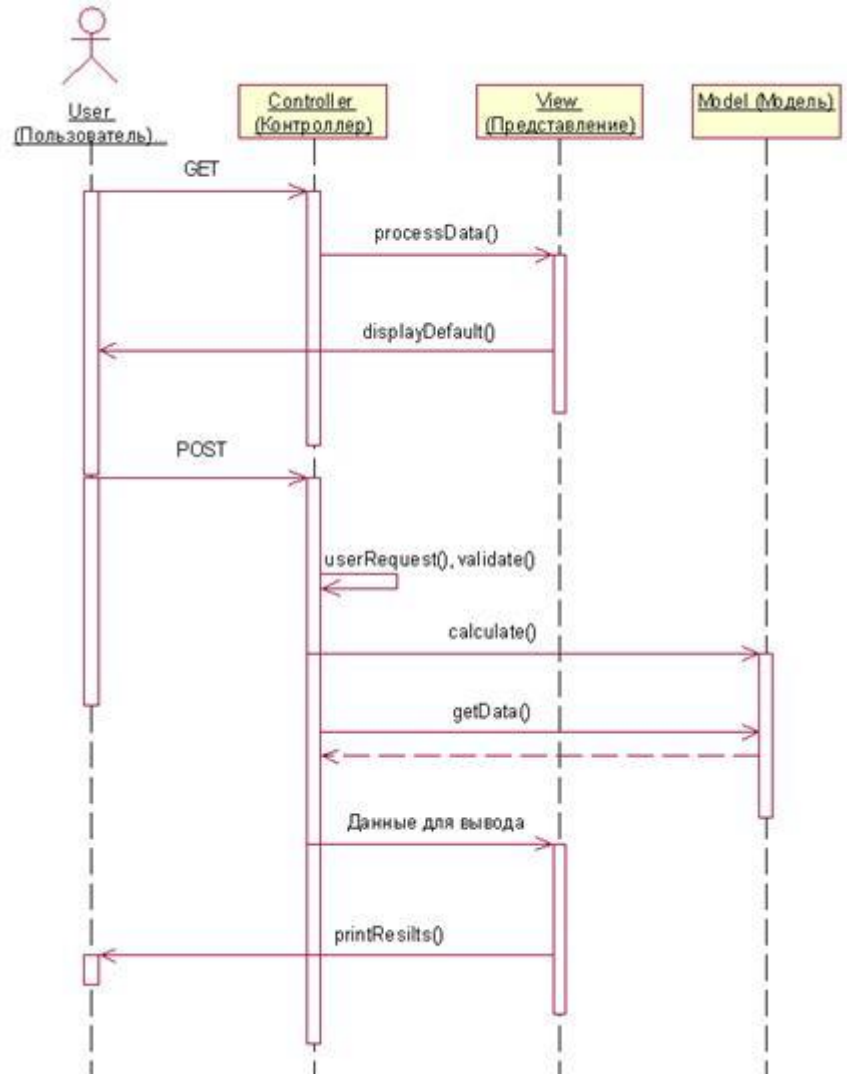
Повертаємося в метод **userRequest** () контролера. Після того, як у ньому був порахований результат і повернений в тому чи іншому вигляді, його можна сміливо віддавати на вхід функції **displayResults** () класу **View**. Однак

зауважимо, що в принципі, можна віддавати на вхід уявлення і екземпляр класу моделі, якщо висновок зберігається в його полях, а їх багато (якщо лінь, так би мовити, створювати структуру, масив або ще якої-небудь об'ємний тип даних).

Якщо функція validate () контролера виявила помилку, і встановила значення поля error в значення, відмінне від false, контролер сам викличе метод displayError () класу View.

Тепер доречно навести ту ж саму діаграму послідовності, але замінивши в ній смислові значення, назвами функцій класів з відповідної діаграми.

**Діаграма послідовностей**




Отже, власне, у нас є три класи та алгоритм взаємодії між ними. Суть архітектурного шаблону MVC полягає в тому, щоб чітко розділити уявлення, управління і модель системи. Це дуже зручно, адже якщо що-небудь зміниться в одній з частин системи, інших частин ці зміни не торкнуться.

Наприклад, у виставі ми можемо написати:

*/ / Це код на PHP*
*public function displayDefault () {*
*echo "<p> Введіть ім'я:";*
*echo "<input type='text' name='name' value=''>";*
*}*

А потім через місяць жахнутися, і перейти до використання шаблонізатора. Скажімо, smarty.

Або, наприклад, в моделі змінити пару розрахункових формул. Або в контролері забрати пару обмежень, або змінити метод прийому-передачі даних. Якщо ж взяти до уваги принципи спадкування в ООП, то архітектура MVC стане ще зручніше. Скажімо, коли є дві форми, що виглядають однаково, але дещо відрізняються алгоритмами розрахунку.

У висновку, хотілося б все ж таки привести деякий скелет коду на PHP для кращого засвоєння ідеї MVC.

```
<?
/ **
* Приклад реалізації MVC на PHP
*
** /
class Controller {
private $ error;
private $ result;
function __construct () {
$ This-> error = false;
$ This-> result = false;
}

function processData () {
$ This-> userRequest ();
if ($ this-> error)
View:: displayError ($ this-> error);
else
if ($ this-> result)
View:: displayResults ($ this-> result);
else
View:: displayDefault ();
}
function userRequest () {
/ / Дані відправлені
if (isset ($ _POST ['send'])) {
$ This-> validate ();
if (! $ this-> error) {
/ / Основні обчислення
$ Model = new Model ();
$ Model-> calculate ($ _POST ['name']);
$ Result = $ model-> getData ();
/ / Перевірка на помилки в самій моделі
if (! is_array ($ result))
$ This-> error = $ result;
```

```
else
$ This-> result = $ result;
}
}
}
function validate () {
if (empty ($ _POST ['name']))
$ This-> error = 'Не введено ім'я!';
else
if (strlen (strval ($ _POST ['name'])) <3)
$ This-> error = 'Ім'я занадто коротке!';
}
} / / Class Controller

class View {
static function displayDefault () {
echo "<form method='POST' action=''>";
echo "<p> Введіть ім'я:";
echo "<input type='text' name='name' value=''>";
echo "<input type='submit' name='send' value='Отправіть'>";
echo "</ form>";
}
static function displayError ($ error) {
echo "<p> <b> Помилка: </ b> {$ error}";
View:: displayDefault ();
}
static function displayResults ($ results) {
echo "<p> <b> Результати: </ b>";
echo "<p> Ваше ім'я <b>". $ results [0].
"</ B> означає <i>". $ Results [1 ]."</ i> ";
echo "<p> <a href ='".$_ SERVER ['REQUEST_URI'].
"'> Дізнатися ще про одне імені </ a>";
}
} / / Class View

class Model {
private $ data;
function __construct () {
$ This-> data = false;
}
function calculate ($ name) {
$ This-> data [] = $ name;
$ Len = strlen ($ name);
if ($ len == 3)
$ This-> data [] = 'стисліcть - сестра таланту';
else
if (($ len> 3) & & ($ len <6))
```

```
$ This-> data [] = '... немає особливого значення';
else
$ This-> data [] = 'неймовірно багата фантазія батьків';
}
function getData () {
if ($ this-> data)
return $ this-> data;
else
return 'Обчислення не проведені!';
}
} / / Class Model

$ Controller = new Controller ();
$ Controller-> processData ();
?>
```

Отже, ми отримали найпростішу MVC-систему. Виділимо позитивні і негативні сторони:

**До мінусів можна віднести**

• Збільшення обсягу коду

• Необхідність дотримання заздалегідь заданого інтерфейсу

• Для підтримки розробки потрібні більш кваліфіковані фахівці

Остання вимога до нашого прикладу не відноситься, але для реальних систем воно досить актуально.

**До плюсів віднесемо наступне:**

• Безсумнівно більш гнучкий код

• Можливість повторного використання кожної з трьох складових частин MVC

• Безболісна заміна моделі (інші алгоритми розрахунку, способу зберігання даних і т.д.)

• Досить просто перейти від одного подання, до іншого (від HTML до XML або JSON)

Треба сказати, що код прикладу не ідеальний. У ньому є простори для рефакторингу (незважаючи на те, що він займає трохи більше ста рядків). Скажімо, у прикладі бере участь всього лише одна змінна, яка надходить від користувача (name), але що якщо їх буде багато?

Вообще MVC это разделение на: **View** - это по сути чистый UI **Controller** - уровень который, обрабатывает данные представления, заносит в Модель, читает из Модели и возвращает в View, т.е. бизнес логика; **Model** - слой данных, взаимодействие с БД или с другим хранилищем (в этой роли очень хорошо выступает строготипизированный датасет, так что зря не пользуетесь всем функционалом VS) Так что сам по себе MVC не сложен и часто начинающие программисты ASP.NET применяют его не подозревая о существовании слова такого, Pattern. Упрощенная структура MVC приложения для ASP.NET выглядит примерно так: -View - html + css + js + asp-контроллы т.е. страничка как ее видит пользователь -Controller - модуль обрабатывающий действия пользователя с View. Может быть как файлом кода связанным со страничкой, так и целым самостоятельным классом (рекомендуется, ибо цель MVC уменьшить зависимость между частями программы, правда многие так и не понимают когда это желательно, а когда излишне, но как говориться: ты начальник - я дурак) -Model - модуль обработки + модуль работы с базой даннях

```
  Controller.cs
1 public static class Marka{
2 static DataTable marka;
3         public static void Reload(OleDbConnection connect)
4         {
5             using (OleDbDataAdapter adap = new OleDbDataAdapter("select*from
6 Марка", connect))
7             {
8                 marka = new DataTable();
9                 adap.Fill(marka);
10            }
11        }
12
13        public static DataTable MarkaTbl
14        {
15            get
16            {
17                if (marka == null)
18                    Reload(ConnectionAccess.Connection);
19                return marka;
20            }
21            set
22            {
23                marka = value;
24            }
25        }
26
27 //Добавление новой записи в бд
28        public static void InsertMarka(params object[] parameters)
29        {
30            int lastid = 0;
31            using (OleDbCommand comm = new OleDbCommand("insert into
32 Марка(Наименование)values(?)", ConnectionAccess.Connection))
33            {
34                foreach (object s in parameters)
35                    comm.Parameters.AddWithValue("PAR", s);
36                ConnectionAccess.Connection.Open();
37                comm.ExecuteNonQuery();
38                comm.CommandText = "select @@identity";
39                lastid = int.Parse(comm.ExecuteScalar().ToString());
40                ConnectionAccess.Connection.Close();
41            }
42
43            MarkaTbl.Rows.Add(lastid, parameters[0]);
44            MarkaTbl.AcceptChanges();
45        }
46 }
```

Код C#

```
1
2  //UI:
3      public partial class MainWindow : Window
4      {
5          public MainWindow()
6          {
7              InitializeComponent();
8
9              markaLv.ItemsSource =Marka.MarkaTbl;
10         }
11
12 //добавляем
```

Код C#

```
13          private void addmarka_Click(object sender, RoutedEventArgs e)
14          {
15              Marka.InsertMarka("Ваз");
16          }
17      }
18
```

## Архітектурні стилі

Архітектурний стиль визначає основні правила виділення компонентів і організації взаємодії між ними в рамках системи або підсистеми в цілому. Різні архітектурні стилі підходять для вирішення різних завдань в плані забезпечення нефункціональних вимог — різних рівнів продуктивності, зручності використання, переносимості і зручності супроводу. Одну і ту ж функціональність можна реалізувати, використовуючи різні стилі.

Робота по виділенню і класифікації архітектурних стилів була проведена в середині 1990-х років. Її результати представлені в роботах [56]. Нижче приведена таблиця деяких архітектурних стилів, виділених в цих роботах.

Таблиця 7.1. Деякі архітектурні стилі

| Види стилів і конкретні стилі | Контекст використання і основні рішення | Приклади |
|---|---|---|
| **Конвеєр обробки даних (data flow)** | Система видає чітко певні вихідні дані в результаті обробки чітко певних вхідних даних, при цьому процес обробки не залежить від часу, застосовується багато разів, однаково до будь-яких даних на вході. Обробка організовується у вигляді набору (необов'язково послідовності) окремих компонентів-обробників, передавальних свої результати на вхід іншим обробникам або на вихід всієї системи.<br><br>Важливими властивостями є чітко певна структура даних і можливість інтеграції з іншими системами | |
| Пакетна обробка (batch sequential) | Один-єдиний вивід проводиться на основі читання деякого одного набору даних на вході, проміжні перетворення організовуються у вигляді послідовності | Збірка програмної системи: компіляція, збірка системи, збірка документації, виконання тестів |
| Канали і фільтри (pipe-and-filter) | Потрібно забезпечити перетворення безперервних потоків даних. При цьому перетворення інкрементальні і наступне може бути почате до закінчення попереднього. Є, можливо, декілька входів і декілька виходів. | Утиліти UNIX |

| | | | |
|---|---|---|---|
| | | Надалі можливе додавання додаткових перетворень | |
| | Замкнутий цикл управління (closed-loop control) | Потрібно забезпечити обробку подій, що постійно поступають, в погано передбаченому оточенні.<br><br>Використовується загальний диспетчер подій, який класифікує подію і віддає його на асинхронну обробку обробникові подій такого типу, після чого диспетчер знову готовий сприймати події | Вбудовані системи управління в автомобілях, авіації, супутниках.<br><br>Обробка запитів на сильно завантажених Web-серверах.<br><br>Обробка дій користувача в GUI |
| **Виклик-повернення (call-return)** | | Порядок виконання дій чітко визначений, окремі компоненти не можуть виконувати корисну роботу, не отримуючи звернення від інших | |
| | Процедурна декомпозиція | Дані незмінні, процедури роботи з ними можуть трохи мінятися, можуть виникати нові.<br><br>Виділяється набір процедур, схема передачі управління між якими є деревом з основною процедурою в його корені | Основна схема побудови програм для мов C, Pascal, Ada |
| | Абстрактні типи даних (abstract data types) | У системі багато даних, структура яких може мінятися. Важливі можливості внесення змін і інтеграції з іншими системами.<br><br>Виділяється набір абстрактних типів даних, кожен з яких надає набір операцій для роботи з даними такого типу. Внутрішнє представлення даних ховається | Бібліотеки класів і компонентів |
| | Многоуровне-вая система (layers) | Є природне розшарування завдань системи на набори завдань, які можна було б вирішувати послідовно, — спочатку завдання першого рівня, потім, використовуючи отримані рішення, — другого, і так далі Важливі переносимість і можливість багатократного використання окремих компо- | Телекоммуникацион-ниє протоколи в моделі OSI (7 рівнів), реальні протоколи мереж передачі даних (звичайні 5 рівнів або менше).<br><br>Системи автоматизації підприємств (рівні інтерфейсу пользователя–обработки запросов–хранения даних) |

| | | |
|---|---|---|
| | нентів.<br><br>Компоненти розділяються на декілька рівнів таким чином, що компоненти даного рівня можуть використовувати для своєї роботи тільки сусідів або компоненти попереднього рівня. Можуть бути слабкіші обмеження, наприклад, компонентам верхніх рівнів дозволено використовувати компоненти всіх рівнів, що пролягають нижче | |
| Клієнт-сервер | Вирішувані завдання природно розподіляються між ініціаторами і обробниками запитів, можлива зміна зовнішнього представлення даних і способів їх обробки | Основна модель бизнес-приложений: клієнтські застосування, що сприймають запити користувачів і сервера, що виконують ці запити |
| **Інтерактивні системи** | Необхідність достатньо швидко реагувати на дії користувача, мінливість призначеного для користувача інтерфейсу | |
| Данниє–представленіє–обработка (model–view-controller, MVC) | Зміни в зовнішньому уявленні достатньо вірогідні, одна і та ж інформація представляється по-різному в декількох місцях, система повинна швидко реагувати на зміни даних.<br><br>Виділяється набір компонентів, відповідальних за зберігання даних, компоненти, відповідальні за їх уявлення для користувачів, і компоненти, що сприймають команди, що перетворюють дані і оновлюючі їх уявлення | Найчастіше використовується при побудові додатків з GUI.<br><br>Document-View в MFC (Microsoft Foundation Classes) — документ в цій схемі об'єднує ролі даних і обробника |
| Представленіє–абстракция–управленіє (presentation–abstraction–control) | Інтерактивна система на основі агентів, що мають власні стани і призначений для користувача інтерфейс, можливе додавання нових агентів.<br><br>Відмінність від попередньої схеми в тому, що для кожного окремого набору даних його модель, уявлення і компонент, що управляє, об'єднуються в агента, відповідального за всю роботу саме з цим набором даних. Агенти взаємодіють один з | |

| | | |
|---|---|---|
| | одним тільки через чітко певну частину інтерфейсу компонентів, що управляють | |
| **Системи на основі сховища даних** | Основні функції системи пов'язані із зберіганням, обробкою і представленням великої кількості даних | |
| Репозиторій (repository) | Порядок роботи визначається тільки потоком зовнішніх подій.<br><br>Виділяється загальне сховище даних — репозиторій. Кожен обробник запускається у відповідь на відповідну йому подію і якось перетворить частину даних в репозиторії | Середовища розробки і CASE-системы |
| Класна дошка (blackboard) | Спосіб рішення задачі в цілому невідомий або дуже трудомісткий, але відомі методи, частково вирішальні завдання, композиція яких здатна видавати прийнятні результати, можливе додавання нових споживачів даних або обробників.<br><br>Окремі обробники запускаються, тільки якщо дані репозиторії для їх роботи підготовлені. Підготовленість даних визначається за допомогою деякої системи шаблонів. Якщо можна запустити декілька обробників, використовується система їх пріоритетів | Системи розпізнавання тексту |

Багато хто з представлених стилів носить достатньо загальний характер і часто зустрічається в різних системах. Крім того, часто можна виявити, що в одній системі використовуються декілька архітектурних стилів — в одній частині переважає один, в іншій — інший, або ж один стиль використовується для виділення крупних підсистем, а інший — для організації дрібніших компонентів в підсистемах.

Докладнішого розгляду заслуговують стилі "Канали і фільтри", "Багаторівнева система". Далі слідують їх описи згідно [7].

## Канали і фільтри

**Назва.** Канали і фільтри (pipes and filters).

**Призначення.** Організація обробки потоків даних у тому випадку, коли процес обробки розпадається на декілька кроків. Ці кроки можуть виконуватися окремими обробниками, можливо, різними розробниками або навіть організаціями, що реалізовуються. При цьому потрібно приймати до уваги наступні чинники:

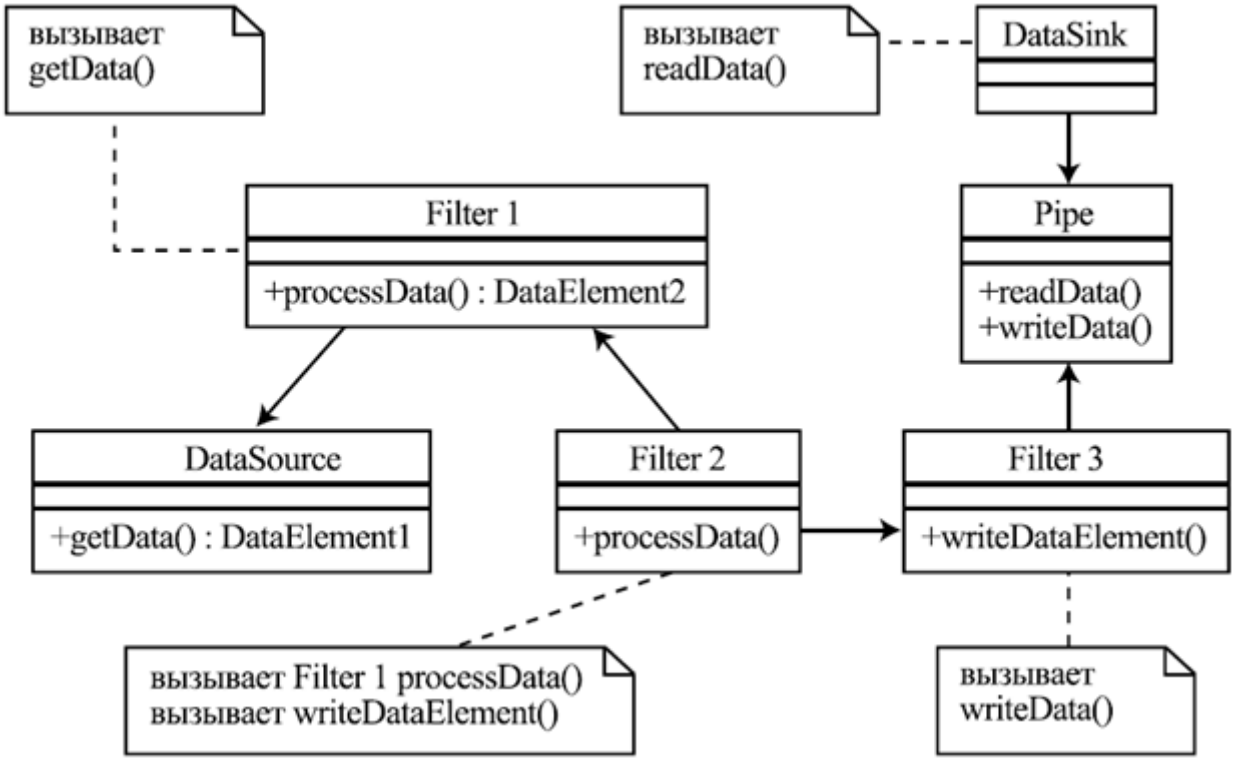
**Сили, що діють.**

- Повинні бути можливі зміни в системі за рахунок додавання нових способів обробки і перекомбинации наявних обробників, іноді самими кінцевими користувачами.
- Невеликі кроки обробки простіше переиспользовать в різних завданнях.
- Обробники, що не є сусідніми, не мають загальних даних.
- Є різні джерела вхідних даних — мережеві з'єднання, текстові файли, повідомлення апаратних датчиків, бази даних.
- Вихідні дані можуть бути затребувані в різних уявленнях.
- Явне зберігання проміжних результатів може бути неефективним, створить безліч тимчасових файлів, може привести до помилок, якщо в його організацію зможе втрутитися користувач.
- Можливе використання паралелізму для ефективнішої обробки даних.

**Рішення.** Кожне окреме завдання по обробці даних розбивається на декілька дрібних кроків. Вихідні дані одного кроку є вхідними для інших. Кожен крок реалізується спеціальним компонентом — **фільтром (filter)**. Фільтр споживає і видає дані инкрементально, невеликими порціями. Передача даних між фільтрами здійснюється по **каналах (pipes)**.

**Структура.** Основними ролями компонентів в рамках даного стилю є фільтр і канал. Іноді виділяють спеціальні види фільтрів — **джерело даних (data source)** і **споживач даних (data sink)**, які, відповідно, тільки видають дані або тільки їх споживають. Кожен потік обробки даних складається з фільтрів, що чергуються, і каналів, починається джерелом даних і закінчується їх споживачем.

Фільтр отримує на свій вхід дані і обробляє їх, доповнюючи їх результатами обробки, видаляючи якісь частини і трансформуючи їх в деяке інше уявлення. Іноді фільтр сам вимагає вхідні дані і видає вихідні після їх отримання, іноді він, навпаки, може реагувати на події приходу даних на вхід і вимоги даних на виході. Фільтр зазвичай споживає і видає дані деякими порціями.

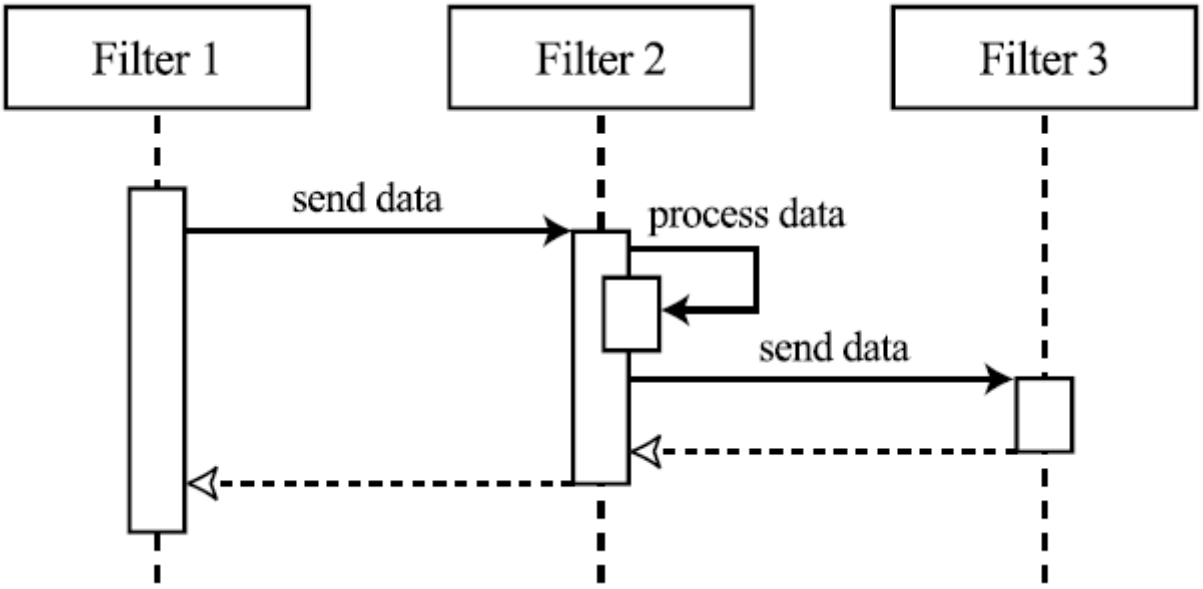



увеличить изображение

**Мал. 7.7.** Приклад структури класів для зразка канали і фільтри

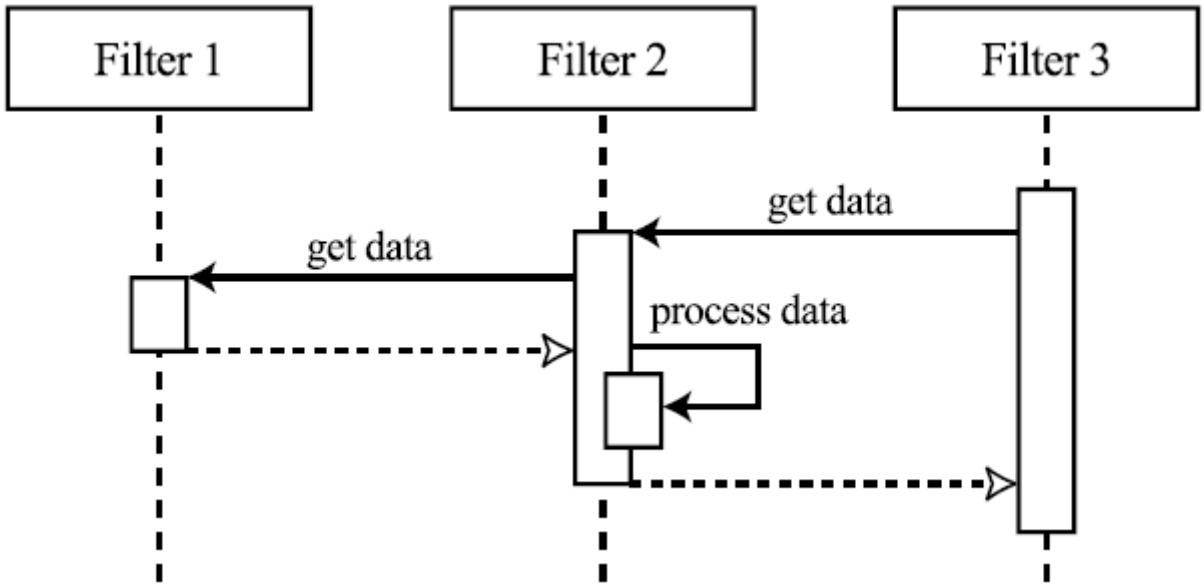
Канал забезпечує передачу даних, їх буферизацію і синхронізацію обробки їх сусідніми фільтрами (наприклад, якщо обидва сусідні фільтри активні, працюють в паралельних процесах). Якщо ніякий додатковий буферизації і синхронізації не потрібний, канал може бути простою передачею даних у вигляді параметра або результату виклику операції.

На рис. 7.7 показаний приклад діаграми класів для даного зразка, в якому 3 канали реалізовано неявно, — через виклики операцій і повернення результатів, а один — явно. З фільтрів, що беруть участь в даному прикладі, джерело і споживач даних, а також Filter 1 запрошують вхідні дані, Filter 3 сам передає їх далі, а Filter 2 і запрошує, і передає дані самостійно.

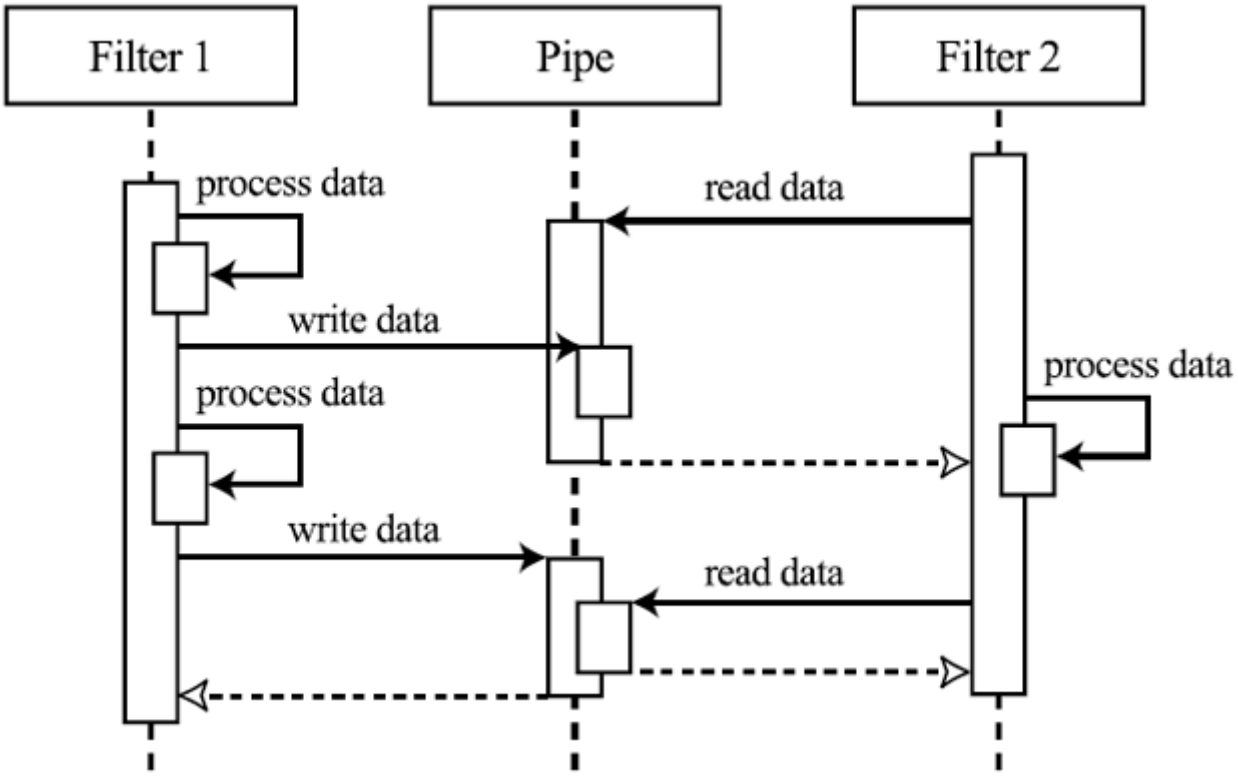



**Мал. 7.8.** Сценарій роботи проштовхуючого фільтру

**Динаміка.** Можливі три різні сценарії роботи одного фільтру — проштовхування даних (push model, фільтр сам передає дані наступному компоненту, а отримує їх тільки в результаті передачі попереднього), витягування даних (pull model, фільтр вимагає дані у попереднього компоненту, наступний сам повинен зажадати дані у нього) і змішаний варіант. Часто реалізується тільки один вид передачі даних для всіх фільтрів в рамках системи. Крім того, канал може буферизувати дані і синхронизовать фільтри, що взаємодіють з ним. Сценарії роботи системи в цілому будуються у вигляді різних комбінацій варіантів роботи окремих фільтрів.




**Мал. 7.9.** Сценарій роботи витягаючого фільтру




увеличить изображение

**Мал. 7.10.** Сценарій роботи того, що буферизує і синхронизующего каналу

**Реалізація.** Основні кроки реалізації наступні:

- Визначити кроки обробки даних, необхідні для вирішення завдань системи. Черговий крок повинен залежати тільки від вихідних даних попереднього кроку.
- Визначити формати даних при їх передачі по кожному каналу.
- Визначити спосіб реалізації кожного каналу, проштовхування або витягування даних, необхідність додаткової буферизації і синхронізації.
- Спроектувати і реалізувати необхідний набір фільтрів. Реалізувати канали, якщо для їх уявлення потрібні окремі компоненти.
- Спроектувати і реалізувати обробку помилок. Обробку помилок при застосуванні цього стилю достатньо важко організувати, тому нею часто нехтують. Проте потрібна, як мінімум, адекватна діагностика помилок, що трапляються на різних етапах.

  Можуть бути виділені спеціальні канали для передачі повідомлень про помилки.

  При виникненні помилок введення відповідний фільтр може ігнорувати подальші вхідні дані до отримання певного роздільника, що гарантує, що після нього йдуть дані, не пов'язані з попередніми.

- Конфігурувати необхідний конвеєр обробки даних, зібравши разом потрібні фільтри і канали, що сполучають їх.

**Следствія застосування зразка.**

Достоїнства:

- Проміжні дані можуть не зберігатися у файлах, але можуть і зберігатися, якщо це необхідно для якихось додаткових цілей.
- Фільтри можна легко замінювати, переиспользовать, міняти місцями, переставляти і комбінувати, реалізовуючи безліч функцій на основі одних і тих же компонентів.
- Конвеєрні системи обробки даних можуть бути розроблені дуже швидко, якщо є багатий набір фільтрів.
- Активні фільтри можуть працювати паралельно, даючи в результаті ефективніше рішення на багатопроцесорних системах.

Недоліки:

- Управління обробкою за допомогою великого загального стану, який іноді необхідний, не може бути ефективно реалізоване за допомогою цього стилю.
- Часто паралельна обробка не приносить ніякого підвищення продуктивності, оскільки передача даних між фільтрами може бути достатнє дорогою, фільтри можуть вимагати всіх вхідних даних, перш ніж видадуть хоч щось, і їх синхронізація за допомогою каналів може приводити до значних простоїв.
- Часто фільтри більший час витрачають на перетворення формату вхідних даних, що поступають, чим на їх обробку. Використання одного формату,

наприклад, текстового, також часто знижує ефективність їх використання.
- Обробка помилок в рамках даного стилю дуже складна. В тому випадку, якщо система, що розробляється, повинна бути дуже надійною, а повернення на сам початок роботи у разі виявлення помилки, а також її ігнорування, не є допустимими сценаріями, використовувати цей стиль не варто.

**Приклади**. Найбільш відомий приклад використання даного зразка — система утиліт UNIX [8]поповнена можливостями оболонки (shell) по організації каналів між процесами. Більшість утиліт можуть грати роль фільтрів при обробці текстових даних, а канали будуються за допомогою з'єднання стандартного введення однієї програми із стандартним виводом інший.

Іншим прикладом може служити часто використовувана архітектура компілятора як послідовності фільтрів, оброблювальних вхідну програму, — лексичного аналізатора (лексера), синтаксичного аналізатора (парсера), семантичного аналізатора, набору оптимізаторів і генератора результуючої коди. У такий спосіб можна достатньо швидко побудувати прототипний компілятор для нескладної мови. Продуктивніші компілятори, націлені на промислове використання, будуються за складнішою схемою, зокрема, використовуючи елементи стилю "Репозиторій".

## Багаторівнева система

**Назва.** Багаторівнева система (layers).

**Призначення.** Реалізація великих систем, які мають велику кількість різнопланових елементів, що використовують один одного. Деякі аспекти роботи таких систем можуть включати багато операцій, що виконуються різними компонентами на різних рівнях (тобто одне завдання вирішується за рахунок послідовних звернень між елементами різних рівнів, інша — теж, але що беруть участь у вирішенні цих завдань елементи можуть бути різні). При цьому потрібно приймати до уваги наступні чинники:

**Сили, що діють.**

- Зміни у вимогах до вирішення одного із завдань не повинні приводити до змін в коді численних компонентів, бажано, щоб вони зводилися до змін усередині одного компоненту. То ж торкається і змін платформи, на якій працює система.
- Інтерфейси між компонентами повинні бути стабільними або навіть відповідати наявним стандартам.
- Частини системи повинні бути замінювані. Компоненти повинні бути замінювані іншими, якщо ті реалізують такі ж інтерфейси. У ідеалі може навіть потрібно в ході роботи перемкнутися на іншу реалізацію, навіть якщо при початку роботи системи вона не була доступна.
- Низькорівневі компоненти повинні дозволяти розробляти інші системи швидше.
- Компоненти з схожими областями відповідальності повинні бути згруповані для підвищення зрозумілості системи і зручності внесення до неї змін.

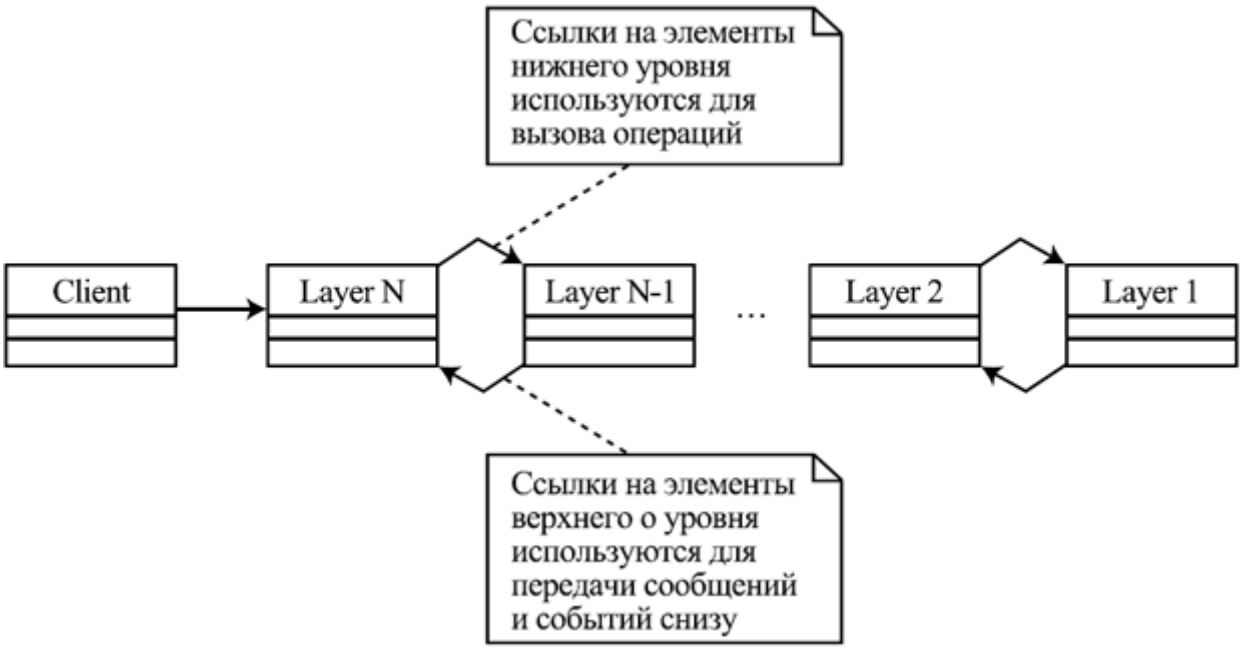
- Немає можливості виділити компоненти деякого стандартного розміру: одні з них вирішують достатньо складні завдання, інші — зовсім прості.
- Складні компоненти потребують подальшої декомпозиції.
- Використання великого числа компонентів може негативно позначитися на продуктивності, оскільки даним доведеться часто долати межі між компонентами.
- Розробка системи повинна бути ефективно поділена між окремими розробниками. При цьому інтерфейси і зони відповідальності компонентів, передаваних різним розробникам, повинні бути дуже чітко визначені.

**Рішення.** Виділяється деякий набір рівнів, кожен з яких відповідає за вирішення своїх власних підзадач. Для цього він використовує інтерфейс, що надається попереднім рівнем, надаючи, у свою чергу, деякий інтерфейс для наступного рівня.

Кожен окремий рівень може бути надалі декомпозирован на дрібніші компоненти.

**Структура.** Основними компонентами є рівні. Іноді виділяють клієнтів, що використовують інтерфейс сам верхнього рівня. Кожен рівень надає інтерфейс для вирішення певної безлічі завдань. Сам він вирішує їх, спираючись на інтерфейс попереднього рівня.

На кожному рівні може знаходитися багато дрібніших компонентів.



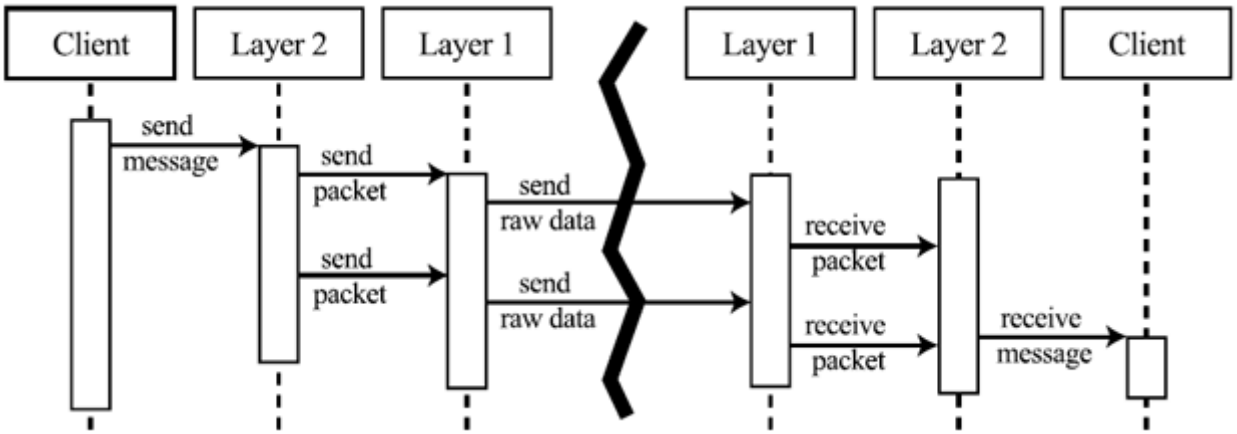

увеличить изображение

**Мал. 7.11.** Приклад структури багаторівневої системи.

Класи для рівнів умовні, вони зазвичай декомпозируются на набори дрібніших компонентів

**Динаміка.** Сценарії роботи системи можуть бути отримані компоновкою наступних чотири. Часто у вигляді багатьох рівнів реалізуються комунікаційні системи, дві такі системи можуть взаємодіяти через самий нижній рівень — при цьому пара симетричних сценаріїв (по подъему–спуску звернень) виконується в рамках одного загального сценарію на різних машинах.

- Звернення клієнта до верхнього рівня ініціює ланцюжок звернень з верхнього рівня до самого нижнього.
- Подія на нижньому рівні (наприклад, прихід повідомлення по мережі або натиснення на кнопку миші) ініціює ланцюжок звернень, що йде від низу до верху, аж до деякої події самої верхнього рівня, видимого клієнтам.
- Звернення клієнта до верхнього рівня приводить до ланцюжка викликів, який, проте, не доходить до самого низу. Така ситуація реалізується, якщо, наприклад, один з рівнів кэширует відповіді на запити і може видати відповідь раніше запит, що вже подавався, без звернення до нижчих рівнів.
- Те ж саме може відбутися і з подією, яка передається з самого нижнього рівня. Дійшовши до деякого рівня, воно може поглинутися їм (із зміною стану якихось компонентів), тому що не відповідає ніякій події на вищих рівнях. Наприклад, натиснення клавіші Capslock не приводить само по собі ні до яких реакцій програми, але змінює значення клавіш, що натискаються після цього, міняючи їх регістр на протилежний.




**Мал. 7.12.** Складений сценарій пересилки повідомлення по мережі

**Реалізація.** Основні кроки реалізації наступні:

- Визначити критерії угрупування завдань по рівнях. Це критично важливий крок. Неправильний розподіл завдань, швидше за все, приведе до необхідності перепроектувати систему.
- Визначити кількість рівнів, які будуть реалізовані, і їх імена. Часто доводиться об'єднувати концептуально різні завдання, щоб добитися більшої ефективності системи. З іншого боку, довільне змішення завдань на рівні веде до незрозумілої архітектури і до системи, дуже незручної для супроводу. При нагоді помістити деяке завдання на декілька рівнів, варто розміщувати її на найвищому рівні з тих, де вона може бути вирішена з достатньою продуктивністю.
- Визначити інтерфейси, що надаються нижніми рівнями верхнім. Тут потрібно пам'ятати, що за допомогою декілька більшого, ніж мінімально необхідний, набору інтерфейсних операцій нижнього рівня можна добитися значного підвищення продуктивності системи в цілому.
- Визначити компоненти і їх взаємодію в рамках кожного окремого рівня.

- Визначити способи взаємодії сусідніх рівнів. Можна використовувати проштовхування, витягування даних або комбінацію цих підходів.
- Відокремити сусідні рівні. У ідеалі нижні рівні не повинні знати нічого про верхніх, кожен рівень повинен знати тільки про безпосередньо передуванні йому. Для цього передачу даних з нижнього рівня можна організувати у вигляді зворотних викликів (callbacks) — покажчик на функцію, яку потрібно викликати для передачі повідомлення вгору, верхній рівень може передавати як параметр при попередніх запитах.
- Спроектувати і реалізувати обробку помилок. Помилки краще обробляти на самому нижньому рівні, який в змозі їх відмітити.

**Следствія застосування зразка.**

Достоїнства:

- Можливість легко замінювати і переиспользовать компоненти одного рівня, не роблячи впливу на решту рівнів. Можливість відладжувати і тестувати рівні окремо.
- Підтримка стандартів. Многоуровневость системи робить можливою підтримку стандартних інтерфейсів, таких як POSIX.

Недоліки:

- Зміна функціональності одного рівня може привести до каскадної зміни всіх рівнів. Істотне зростання продуктивності нижнього рівня і вимога забезпечити відповідне зростання продуктивності на вищих рівнях також можуть привести до перевизначення всіх інтерфейсів.
- Падіння продуктивності із-за необхідності проводити всі виклики і дані через всі рівні.
- Часто рівні дублюють роботу один одного, наприклад, при обробці помилок, оскільки вони розробляються незалежно і не мають інформації про деталі реалізації один одного.
- Велика кількість рівнів може привести до істотного підвищення складності системи і падіння її продуктивності. З іншого боку, дуже мале число рівнів (наприклад, два) часто не дозволяє забезпечити необхідну гнучкість і переносимість.

**Приклади**. Найбільш відомий приклад використання даного зразка — стандартна модель протоколів зв'язку відкритих систем (Open System Interconnection, OSI) [9]. Вона складається з 7 рівнів:

- Самий нижній рівень — **фізичний**. Він відповідає за передачу окремих бітів по каналах зв'язку. Основні його завдання — гарантувати правильне визначення нуля і одиниці різними системами, визначити тимчасові характеристики передачі (за який час передається один біт), забезпечити передачу в одному або двох напрямах, і тому подібне

- Другий рівень — **канальний** або **рівень передачі даних**. Його завдання — надати верхнім рівням такі сервіси, щоб для них передача даних виглядала б як посилка і прийом потоку байт без втрат і без перевантажень.
- Третій рівень — **мережевий**. Його завдання — забезпечити прозорий зв'язок між комп'ютерами, не сполученими безпосередньо, а також забезпечувати нормальну роботу великих мереж, по яких одночасно подорожує дуже багато пакетів даних.
- Четвертий рівень — **транспортний**. Він забезпечує надійну передачу даних верхніх рівнів немов по деякій трубі — пакети приходять обов'язково в тій же послідовності, в якій вони були відправлені. Відмітимо, що канальний рівень вирішує таку ж задачу, але тільки для машин, що безпосередньо зв'язуються один з одним.
- П'ятий, **сеансовий** рівень надає можливість встановлювати сеанси зв'язки (або сесії), що містять деякий набір передаваних туди і назад повідомлень, і управляти ними.
- Шостий, **рівень уявлення**, визначає формати передаваних даних. Наприклад, саме тут визначається, що ціле число представлятиметься 4 байтами, причому старші біти числа йдуть раніше молодших, перший біт інтерпретується як знак, а негативні числа представляються в додатковій системі (тобто 0x0000000f позначає 15, а 0x80000000f — 2147483633 = –(231–15)).
- Нарешті, сьомий рівень — прикладний — містить набір протоколів, якими безпосередньо користуються програми і з якими працюють користувачі, — HTTP, FTP, SMTP, POP3 і ін.

Модель OSI опинилася все ж таки дуже складна для використання на практиці. Зараз найбільш широко вживані набори протоколів будуються за урізаною схемою OSI — в ній відсутні п'ятий і шостий рівні, прикладні протоколи користуються безпосередньо службами протоколів транспортного рівня.

Інший приклад багаторівневої архітектури — архітектура сучасних інформаційних систем або систем автоматизації бізнесу. Вона включає наступні рівні [3]:

- Інтерфейс взаємодії із зовнішнім середовищем.

  Найчастіше цей рівень розглядається як інтерфейс користувача. У його рамках визначається представлення даних для передачі іншим системам або користувачам, набір екранів, форм і звітів, з якими мають справу користувачі.

- Бізнес-логіка. На цьому рівні реалізуються основні правила функціонування даного бізнесу, даній організації.
- Наочна область. Даний рівень містить концептуальну схему даних, з якими має справу організація. Ці ж дані можуть використовуватися і іншими організаціями в своїй роботі.
- Рівень управління ресурсами.

  На нім знаходяться всі ресурси, якими користується система, зокрема інші системи. Дуже часто використовувані ресурси зводяться до набору баз даних, необхідних для роботи організації. На цьому рівні визначається структура використовуваних ресурсів

і способи управління ними, зокрема, конкретне розміщення даних по таблицях реляційної бази даних або класах об'єктної бази даних і відповідний набір індексів. Найчастіше схеми баз даних оптимізуються під конкретний набір запитів, і тому їх структура декілька відрізняється від концептуальної схеми даних, що знаходиться на попередньому рівні.

Часто два середні рівні об'єднуються в один — рівень функціонування додатків, що дає в результаті широку використовувану **триланкову архітектуру** інформаційних систем.

## Зміст

# ЧАСТИНА III. ПРОЕКТУВАННЯ

## 10. Архітектурне проектування

### Цілі

Мета справжнього розділу – познайомити читача з архітектурою програмного забезпечення і концепціями архітектурного проектування. Прочитавши цей розділ, ви винні:

- ❑ розуміти, для чого необхідне архітектурне проектування ПО;
- ❑ знати різні моделі, використовувані при документуванні системної архітектури;
- ❑ мати уявлення про різні типи архітектури ПО: структурній моделі системи, моделі системної декомпозиції по управлінню і моделі модульної декомпозиції;
- ❑ знати моделі проблемно-зависимых архітектури, яка використовується як основа для архітектури спеціалізованих програмних систем і як еталон при порівнянні різної архітектури.

Великі системи завжди можна розбити на підсистеми, що надають зв'язані набори сервісів. *Архітектурним проектуванням* називають перший етап процесу проектування, на якому визначаються підсистеми, а також структура управління і взаємодії підсистем. Метою архітектурного проектування є опис *архітектури програмного забезпечення.*

У розділі 3.4 розглядалася загальна структура процесу проектування. На мал. 3.9 була представлена модель процесу проектування, першим етапом якого є архітектурне проектування, що служить сполучаючою ланкою між процесом проектування і процесом розробки вимог до створюваної системи. У ідеалі в специфікації вимог не повинно бути інформації про структуру системи. Насправді ж це справедливо тільки для невеликих систем. Архітектурна декомпозиція системи необхідна для структуризації і організації системної специфікації. Хорошим прикладом тому може служити зображена на мал. 2.3 система управління повітряними польотами. Модель системної архітектури часто є відправною крапкою для створення специфікації різних частин системи. В процесі архітектурного проектування розробляється базова структура системи, тобто визначаються основні компоненти системи і взаємодії між ними.

Існують різні підходи до процесу архітектурного проектування, які залежать від професійного досвіду, а також майстерності і інтуїції розробників. Та все ж можна виділити декілька етапів, загальних для всіх процесів архітектурного проектування.

1. *Структуризація системи.* Програмна система структурується у вигляді сукупності щодо незалежних підсистем. Також визначаються взаємодії між підсистемами. Цей етап розглядається в розділі 10.1.
2. *Моделювання управління.* Розробляється базова модель управління взаєминами між частинами системи. Цей етап розглядається в розділі 10.2.
3. *Модульна декомпозиція.* Кожна визначена на першому етапі підсистема розбивається на окремі модулі. Тут визначаються типи модулів і типи їх взаємозв'язків. Цей етап розглядається в розділі 10.3.

Як правило, ці етапи перемежаються і накладаються один на одного. Етапи повторюються для все більш детального опрацьовування архітектури до тих пір, поки архітектурний проект не задовольнятиме системним вимогам.

Чітких відмінностей між підсистемами і модулями немає, але, думаю, будуть корисними наступні визначення.

1. *Підсистема* – це система (тобто задовольняє "класичному" визначенню "система"), операції (методи) якої не залежать від сервісів, що надаються іншими підсистемами.

Підсистеми складаються з модулів і мають певні інтерфейси, за допомогою яких взаємодіють з іншими підсистемами.

2. *Модуль* – це звичайно компонент системи, який надає один або декілька сервісів для інших модулів. Модуль може використовувати сервіси, підтримувані іншими модулями. Як правило, модуль ніколи не розглядається як незалежна система. Модулі зазвичай складаються з ряду інших, простіших компонентів.

Результатом процесу архітектурного проектування є документ, що відображає архітектуру системи. Він складається з набору графічних схем представлень моделей системи з відповідним описом. У описі повинно бути вказано, з яких підсистем складається система і з яких модулів складається кожна підсистема. Графічні схеми моделей системи дозволяють поглянути на архітектуру з різних сторін. Як правило, розробляється чотири архітектурні моделі.

1. Статична структурна модель, в якій представлені підсистеми або компоненти, що розробляються надалі незалежно.
2. Динамічна модель процесів, в якій представлена організація процесів під час роботи системи.
3. Інтерфейсна модель, яка визначає сервіси, що надаються кожною підсистемою через загальний інтерфейс.
4. Моделі відносин, в яких показані взаємовідношення між частинами системи, наприклад потік даних між підсистемами.

Ряд дослідників при описі архітектури систем пропонує використовувати спеціальні мови опису архітектури. У книзі [29] розглядаються основні властивості цих мов. У них основними архітектурними елементами є компоненти і коннектори (об'єднуючі ланки); ці мови також пропонують принципи і правила побудови архітектури. Проте, як і інші спеціалізовані мови, вони мають один недолік, а саме: всі вони зрозумілі фахівцям, що тільки освоїли їх, і майже не використовуються на практиці. Фактично використання мов опису архітектури тільки ускладнює аналіз систем. Тому я вважаю, що для опису архітектури краще використовувати неформальні моделі і системи нотації, подібні пропонованою, наприклад уніфікована мова моделювання UML.

Архітектура системи може будуватися відповідно до певної архітектурної моделі [126]. Дуже важливо знати ці моделі, їх недоліки, переваги і можливості застосування. У цьому розділі розглядаються структурні моделі, моделі управління і декомпозиції.

Разом з тим архітектуру великих систем неможливо описати за допомогою якої-небудь одній моделі. При розробці окремих частин великих систем можна використовувати різні архітектурні моделі. Але в цьому випадку архітектура системи може виявитися дуже складною, оскільки буде побудована на комбінації різних архітектурних моделей. Розробник повинен підібрати найбільш відповідну модель, потім модифікувати її відповідно вимогам того, що розробляється ПО. У розділі 10.4 розглядається приклад архітектури компілятора, моделі репозиторія і моделі потоків даних, що базується на комбінації.

Архітектура системи впливає на продуктивність, надійність, зручність супроводу і інші характеристики системи. Тому моделі архітектура, вибрана для даної системи, може залежати від нефункціональних системних вимог.

1. *Продуктивність.* Якщо критичною вимогою є продуктивність системи, слід розробити таку архітектуру, щоб за всі критичні операції відповідали якомога менше підсистем з максимально малою взаємодією між ними. Щоб зменшити взаємодію між компонентами, краще використовувати крупномодульные компоненти, а не дрібні структурні елементи.
2. *Захищеність.* В цьому випадку архітектура повинна мати багаторівневу структуру, в якій найбільш критичні системні елементи захищені на внутрішніх рівнях, а перевірка безпеки цих рівнів здійснюється на більш високому рівні.

3. *Безпека.* В цьому випадку архітектуру слід спроектувати так, щоб за всі операції, що впливають на безпеку системи, відповідали якомога менше підсистем. Такий підхід дозволяє понизити вартість розробки і вирішує проблему перевірки надійності.
4. *Надійність.* В цьому випадку слід розробити архітектуру з включенням надмірних компонентів, щоб можна було замінювати і оновлювати їх, не перериваючи роботу системи. Архітектура відмовостійких систем з високою працездатністю розглядається в розділі 18.
5. *Зручність супроводу.* В цьому випадку архітектуру системи слід проектувати на рівні дрібних структурних компонентів, які можна легко змінювати. Програми, що створюють дані, повинні бути відокремлені від програм, що використовують ці дані. Слід також уникати структури сумісного використання даних.

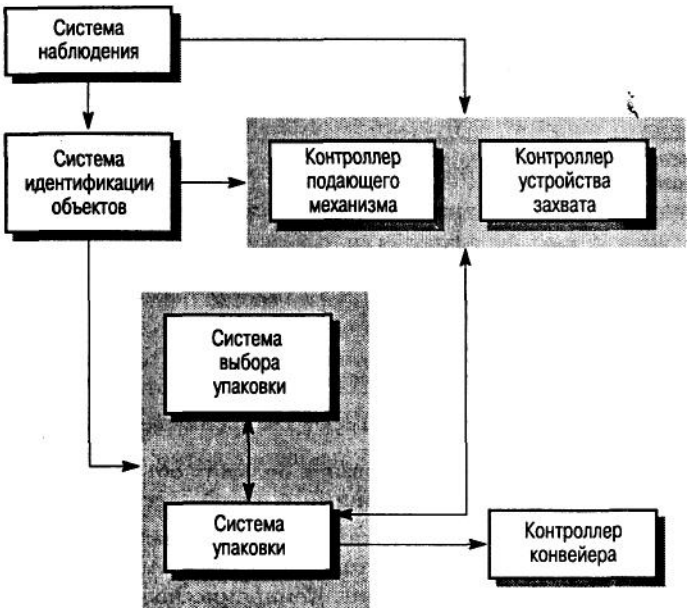
Очевидно, що деякі з перерахованої архітектури противоречат один одному. Наприклад, для того, щоб підвищити продуктивність, необхідно використовувати крупномодульные компоненти, в той же час супровід системи набагато спрощується, якщо вона складається з дрібних структурних компонентів. Якщо необхідно врахувати обидві вимоги, слід шукати компромісне рішення. Раніше вже було сказано, що один із способів вирішення подібних проблем полягає в застосуванні різних архітектурних моделей для різних частин системи.

## 10.1. Структуризація системи

На першому етапі процесу проектування архітектури система розбивається на декілька взаємодіючих підсистем. На найабстрактнішому рівні архітектуру системи можна зобразити графічно за допомогою блок-схеми, в якій окремі підсистеми представлені окремими блоками. Якщо підсистему також можна розбити на декілька частин, на діаграмі ці частини зображаються прямокутниками усередині великих блоків. Потоки даних і/або потоки управління між підсистемами позначається стрілками. Така блок-схема дає загальне уявлення про структуру системи.

На мал. 10.1 представлена структурна модель архітектури для системи управління автоматичною упаковкою різних типів об'єктів. Вона складається з декількох частин. Підсистема спостереження вивчає об'єкти на конвеєрі, визначає тип об'єкту і вибирає для нього відповідний тип упаковки. Потім об'єкти знімаються з конвеєра, упаковуються і поміщаються на інший конвеєр. Приклади іншої архітектури приведені на мал. 2.2 і 2.3.

Бесс (Bass [29]) вважає, що подібні блок-схеми є даремними представленнями системної архітектури, оскільки з них не можна нічого дізнатися ні про природу взаємин між компонентами системи, ні про їх властивості. З погляду розробника програмного забезпечення, це абсолютно вірно. Проте такі моделі виявляються ефективними на етапі попереднього проектування системи. Ця модель не переобтяжена деталями, з її допомогою зручно представити структуру системи. У структурній моделі визначені всі основні підсистеми, які можна розробляти незалежно від решти підсистем, отже, керівник проекту може розподілити розробку цих підсистем між різними виконавцями. Звичайно, для уявлення архітектура використовується не тільки блок-схеми, проте подібне представлення системи не менш корисний, чим інші архітектурні моделі.




*Мал. 10.1. Блок-схема системи управління автоматичною упаковкою*

Звичайно, можна розробляти більш деталізовані моделі структури, в яких було б показано, як саме підсистеми розділяють дані і як взаємодіють один з одним. У цьому розділі розглядаються три стандартні моделі, а саме: модель репозиторія, модель клієнт/сервер і модель абстрактної машини.

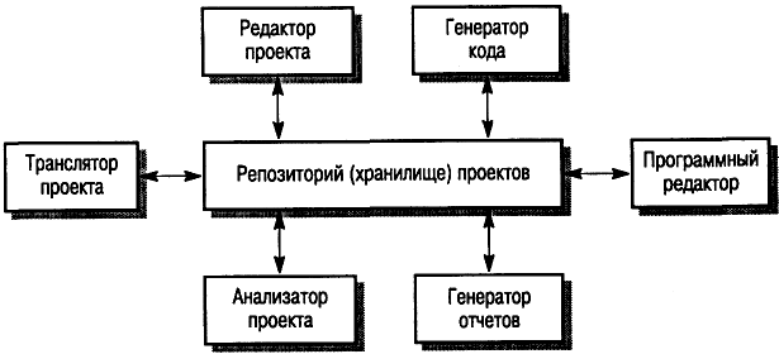
### 10.1.1. Модель репозиторія

Для того, щоб підсистеми, складові систему, працювали ефективніше, між ними повинен йти обмін інформацією. Обмін можна організувати двома способами.

1. Всі спільно використовувані дані зберігаються в центральній базі даних, доступній всім підсистемам. Модель системи, заснована на сумісному використанні бази даних, часто називають *моделлю репозиторія.*
2. Кожна підсистема має власну базу даних. Взаємообмін даними між підсистемами відбувається за допомогою передачі повідомлень.

Більшість систем, оброблювальних великі об'єми даних, організовані навколо спільно використовуваної бази даних, або репозиторія. Тому *така* модель підійде до додатків, в яких дані створюються в одній підсистемі, а використовуються в іншій. Прикладами можуть служити системи управління інформацією, системи автоматичного проектування і CASE-средства.

На мал. 10.2 представлений приклад архітектури інтегрованого набору CASE-инструментов, заснований на спільно використовуваному репозиторії. Вважається, що для CASE-средств перший спільно використовуваний репозиторії був розроблений на початку 1970-х років англійською компанією ICL в процесі створення своєї операційної системи [234]. Широку популярність ця модель здобула після того, як була застосована для підтримки розробки систем, написаних на мові Ada. З тих пір багато CASE-средства розробляються з використанням загального репозиторія.




*Мал. 10.2. Архітектура інтегрованого набору CASE-средств*

Спільно використовувані репозиторії мають як переваги, так і недоліки.

1. Очевидно, що сумісне використання великих об'ємів даних ефективне, оскільки не потрібно передавати дані з однієї підсистеми в інших.
2. З іншого боку, підсистеми повинні бути узгоджені з моделлю репозиторія даних. Це завжди приводить до необхідності компромісу між вимогами, що пред'являються до кожної підсистеми. Компромісне рішення може знизити їх продуктивність. Якщо формати даних нових підсистем не підходять під узгоджену модель представлення даних, інтегрувати такі підсистеми складно або неможливо.
3. Підсистемам, в яких створюються дані, не потрібно знати, як ці дані використовуються в інших підсистемах.
4. Оскільки відповідно до узгодженої моделі даних генеруються великі об'єми інформації, модернізація таких систем проблематична. Переклад системи на нову модель даних буде дорогим і складним, а деколи навіть неможливим.
5. У системах з репозиторієм такі засоби, як резервне копіювання, забезпечення безпеки, управління доступом і відновлення даних, централізовані, оскільки входять в систему управління репозиторієм. Ці засоби виконують тільки свої основні операції і не займаються іншими питаннями.
6. З іншого боку, до різних підсистем пред'являються різні вимоги, що стосуються безпеки, відновлення і резервування даних. У моделі репозиторія до всіх підсистем застосовується однакова політика.
7. Модель сумісного використання репозиторія прозора: якщо нові підсистеми сумісні з узгодженою моделлю даних, їх можна безпосередньо інтегрувати в систему.
8. Проте складно розмістити репозиторії на декількох машинах, оскільки можуть виникнути проблеми, пов'язані з надмірністю і порушенням цілісності даних.

У даній моделі репозиторії є пасивним елементом, а управління їм покладене на підсистеми, що використовують дані з репозиторія. Для систем штучного інтелекту розроблений альтернативний підхід. Він заснований на моделі "робочої області", яка ініціює підсистеми тоді, коли конкретні дані стають доступними. Такий підхід застосовний до систем, в яких форма даних добре структурована. Ця модель обговорюється в роботі [255].

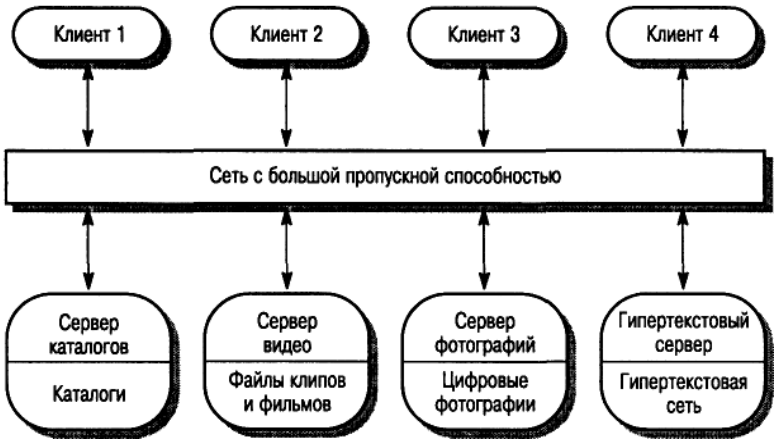
### 10.1.2. Модель клієнт/сервер

Модель архітектури клієнт/сервер – це модель розподіленої системи, в якій показаний розподіл даних і процесів між декількома процесорами. Модель включає три основні компоненти.

1. Набір автономних серверів, що надають сервіси іншим підсистемам. Наприклад, сервер друку, який надає послуги друку, файлові сервери, що надають сервіси управління файлами, і сервер-компілятор, який пропонує сервіси по компіляції початкових код програм.

2. Набір клієнтів, які викликають сервіси, що надаються серверами. У контексті системи клієнти є звичайними підсистемами. Допускається паралельне виконання декількох екземплярів клієнтської програми.
3. Мережа, за допомогою якої клієнти дістають доступ до сервісів. В принципі немає ніякої заборони на те, щоб клієнти і сервери запускалися на одній машині. На практиці, проте, модель клієнт/сервер в такій ситуації не використовується.

Клієнти повинні знати імена доступних серверів і сервісів, які вони надають. В той же час серверам не потрібно знати ні імена клієнтів, ні їх кількість. Клієнти дістають доступ до сервісів, сервером, що надається, за допомогою видаленого виклику процедур.




*Мал. 10.3. Архітектура бібліотечної системи фільмів і фотографій*

Приклад системи, організованої за типом моделі клієнт/сервер, показаний на мал. 10.3. Це многопользовательская гіпертекстова система, призначена для підтримки бібліотек фільмів і фотографій. У ній міститься декілька серверів, які розміщують різні типи медиафайлов і управляють ними. Відеофайли потрібно передавати швидко і синхронно, але з відносно малим дозволом. Вони можуть зберігатися в стислому стані. Фотографії повинні передаватися з високим дозволом. Каталоги повинні забезпечувати роботу безліччю запитів і підтримувати зв'язки з використанням гіпертекстової системи. Тут клієнтська програма є просто інтегрованим інтерфейсом користувача.

Підхід клієнт/сервер можна використовувати при реалізації систем, заснованих на репозиторії, який підтримується як сервер системи. Підсистеми, що мають доступ до репозиторія, є клієнтами. Але зазвичай кожна підсистема управляє власними даними. Під час роботи сервери і клієнти обмінюються даними, проте при обміні великими об'ємами даних можуть виникнути проблеми, пов'язані з пропускною спроможністю мережі. Правда, з розвитком все більш швидких мереж ця проблема втрачає своє значення.

Найбільш важлива перевага моделі клієнт/сервер полягає в тому, що вона є розподіленою архітектурою. Її ефективно використовувати в мережевих системах з безліччю розподілених процесорів. У систему легко додати новий сервер і інтегрувати його з рештою частини системи або ж відновити сервери, не впливаючи на інші частини системи. В розділі 11 архітектури розподілених систем розглядаються детальніше.

### 10.1.3. Модель абстрактної машини

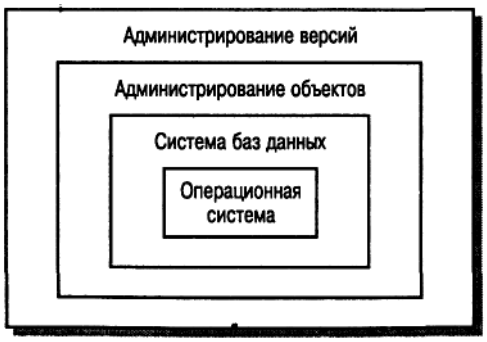
Модель архітектури абстрактної машини (іноді звана багаторівневою моделлю) моделює взаємодія підсистем. Вона організовує систему у вигляді набору рівнів, кожен з яких надає свої сервіси. Кожен рівень визначає *абстрактну машину,* машинна мова якої (сервіси, що надаються рівнем) використовується для реалізації наступного рівня абстрактної машини. Наприклад, найбільш поширений спосіб реалізації мови програмування полягає у визначенні ідеальної "мовної машини" і компіляції програм, написаних на даній мові, в код цієї машини.

На наступному кроці трансляції код абстрактної машини конвертується в реальний машинний код.

Добре відомим прикладом такого походу може служити модель OSI* мережевих протоколів [352], обговорювана в розділі 10.4. Іншим прикладом є трирівнева модель середовища програмування на мові Ada [66]. На мал. 10.4 зображена подібна модель і показано, як за допомогою моделі абстрактної машини можна представити систему адміністрування версій.

** OSI (Open System Interconnection - взаємодія відкритих систем) - міжнародна програма стандартизації обміну даними між комп'ютерними системами на основі семирівневої моделі протоколів передачі даних у відкритих системах. Ця модель запропонована Міжнародною організацією по стандартизації ISO (International Standards Organization).* - Прим. ред.

Система адміністрування версій заснована на управлінні версіями об'єктів і надає засоби для повного управління конфігурацією системи (див. розділ 29). Для підтримки засобів управління конфігурацією використовується система адміністрування об'єктів, що підтримує систему бази даних і сервіси управління об'єктами. У свою чергу, в системі баз даних підтримуються різні сервіси, наприклад управління транзакціями, відкоту назад, відновлення і управління доступом. Для управління базами даних використовуються засоби основної операційної системи і її файлова система.




*Мал. 10.4. Модель абстрактної машини для системи адміністрування версій*

Багаторівневий підхід забезпечує покроковий розвиток систем – при розробці якого-небудь рівня сервіси, що надаються ним, стають доступні користувачам. Крім того, така архітектура легко змінна і переносима на різні платформи. Зміна інтерфейсу будь-якого рівня вплине тільки на суміжний рівень. Оскільки в багаторівневих системах залежності від машинної платформи локалізовані на внутрішніх рівнях, такі системи можна реалізувати на інших платформах, оскільки потрібно буде змінити тільки самі внутрішні рівні.

Недоліком багаторівневого підходу є досить складна структура системи. Основні засоби, такі як управління файлами, необхідні всім абстрактним машинам, надаються внутрішніми рівнями. Тому сервісам, запрошуваним користувачем, можливо, буде потрібно доступ до внутрішніх рівнів абстрактної машини. Така ситуація приводить до руйнування моделі, оскільки зовнішній рівень залежить не тільки від передування йому рівня, але і від нижчих рівнів.

## 10.2. Моделі управління

У моделі структури системи показані всі підсистеми, з яких вона складається. Для того, щоб підсистеми функціонували як єдине ціле, необхідно управляти ними. У структурних моделях немає (і не повинно бути) ніякої інформації по управлінню. Проте розробник архітектури повинен організувати підсистеми згідно деякої моделі управління, яка доповнювала б наявну модель структури. У моделях управління на рівні архітектури проектується потік управління між підсистемами.

Можна виділити два основні типи управління в програмних системах.

1. *Централізоване управління.* Одна з підсистем повністю відповідає за управління, запускає і завершує роботу решти підсистем. Управління від першої підсистеми може перейти до іншої підсистеми, проте потім обов'язково повертається до першої.
2. *Управління, засноване на подіях.* Тут замість однієї підсистеми, відповідальної за управління, на зовнішні події може відповідати будь-яка підсистема. Події, на які реагує система, можуть відбуватися або в інших підсистемах, або в зовнішньому оточенні системи.

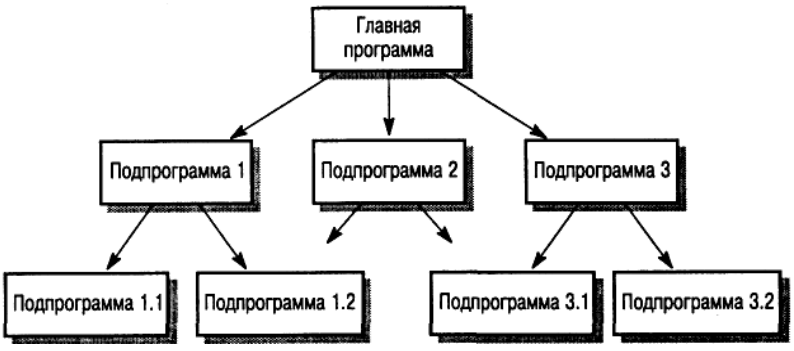
Модель управління доповнює структурні моделі. Всі описані раніше структурні моделі можна реалізувати за допомогою централізованого управління або управління, заснованого на подіях.

### 10.2.1. Централізоване управління

У моделі централізованого управління одна з систем призначається головною і управляє роботою інших підсистем. Такі моделі можна розбити на два класи, залежно від того, послідовно або паралельно реалізовано виконання керованих підсистем.

1. *Модель виклику-повернення.* Це відома модель організації виклику програмних процедур зверху "вниз", в якій управління починається на вершині ієрархії процедур і через виклики передається на більш нижні рівні ієрархії. Дана модель застосовна тільки в послідовних системах.
2. *Модель диспетчера.* Застосовується в паралельних системах. Один системний компонент призначається диспетчером і управляє запуском, завершенням і координуванням інших процесів системи. Процес (виконувана підсистема або модуль) може протікати паралельно з іншими процесами. Модель такого типу застосовна також в послідовних системах, де програма, що управляє, викликає окремі підсистеми залежно від значень деяких змінних стану. Звичайне таке управління реалізується через оператора case.

Модель виклику-повернення представлена на мал. 10.5. З головної програми можна викликати підпрограми 1, 2 і 3, з підпрограми 1 – підпрограми 1.1 і 1.2, з підпрограми 3 – підпрограми 3.1 і 3.2 і так далі Така модель виконання підпрограм *не є структурною* – підпрограма 1.1 не обов'язково є частиною підпрограми 1.




*Мал. 10.5. Модель виклику-повернення*

Подібна модель вбудована в мови програмування Ada, Pascal і C. Управленіє переходить від програми, розташованої на самому верхньому рівні ієрархії, до підпрограми більш нижнього рівня. Потім відбувається повернення управління в точку виклику підпрограми. За управління відповідає та підпрограма, яка виконується у нинішній момент; вона може або викликати інші підпрограми, або повернути управління підпрограмі, що викликала її. Недосконаліть даного стилю програмування при поверненні до певної крапки в програмі очевидно.

Модель виклику-повернення можна використовувати на рівні модулів для управління функціями і об'єктами. Підпрограми в мові програмування, які викликаються з інших підпрограм, є природно функціональними. Проте в багатьох об'єктно-орієнтованих системах операції в об'єктах (методи) реалізовані у вигляді процедур або функцій. Наприклад, об'єкт Java запрошує сервіс з іншого об'єкту за допомогою виклику відповідного методу.

Жорстка і обмежена природа моделі виклику-повернення є одночасно і перевагою і недоліком. Переваги моделі виявляються у відносно простому аналізі потоків управління, а також при виборі системи, що відповідає за конкретне введення даних. Недолік моделі, як ви дізнаєтеся з розділу 18, полягає в складній обробці виняткових ситуацій.

На мал. 10.6 представлена модель централізованого управління для паралельної системи. Подібна модель часто використовується в "м'яких" системах реального часу, в яких немає занадто строгих временіых. обмежень. Центральний контроллер управляє виконанням безлічі процесів, пов'язаних з датчиками і виконавчими механізмами. Система, що використовує таку модель управління, розглянута в розділі 13.


*Мал. 10.6. Модель централізованого управління для системи реального часу*

Контроллер системи, залежно від змінних стану системи, визначає моменти запуску або завершення процесів. Він перевіряє, чи генерується в решті процесів інформація, для того, щоб потім обробити її або передати іншим процесам на обробку. Зазвичай контроллер працює постійно, перевіряючи датчики і інші процеси або відстежуючи зміни стану, тому дану модель іноді називають моделлю із зворотним зв'язком.

### 10.2.2. Системи, керовані подіями

У моделях централізованого управління, як правило, управління системою визначається значеннями деяких змінних її стану. В протилежність таким моделям існують системи, управління якими засноване на зовнішніх подіях. У даному контексті під *подією* мається на увазі не тільки бінарний сигнал типу "да-нет". Тут сигнал може приймати деякий діапазон значень. Відмінність між подією і звичайними вхідними даними полягає в тому, що планування події виходить за рамки управління процесом, оброблювальним це подія. Для обробки події підсистемі необхідний доступ до інформації стану, проте така інформація зазвичай не визначається потоком управління.

Розроблена безліч різних типів подієво-керованих систем. До них відносяться електронні таблиці, в яких зміна значення в якому-небудь осередку змінює вміст інших осередків, системи штучного інтелекту, в яких при виконанні деякої умови відбувається ініціація дії або використовуються активні об'єкти, тоді при зміні значення властивості об'єкту ініціюється деяка дія. У книзі [128] детально розглядаються різні типи подієво-керованих систем.

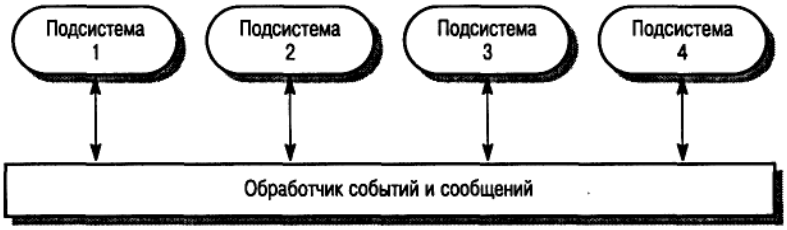
У цьому розділі описано дві моделі систем, керованих подіями.

1. *Моделі передачі повідомлень.* У цих моделях подія є передачею повідомлення всім підсистемам. Будь-яка підсистема, яка обробляє дану подію, відповідає на нього.

2. *Моделі, керовані перериваннями.* Такі моделі зазвичай використовуються в системах реального часу, де зовнішні переривання реєструються обробником переривань, а обробляються іншим системним компонентом.

Моделі передачі повідомлень ефективні при інтеграції підсистем, розподілених на різних комп'ютерах, які об'єднані в мережу. Моделі, керовані перериваннями, використовуються в системах реального часу із строгими тимчасовими вимогами.

У моделі передачі повідомлень (мал. 10.7) підсистеми реагують на певні події. Якщо відбулася деяка подія, управління переходить до підсистеми, оброблювальної дану подію. Між моделлю передачі повідомлень і моделлю централізованого управління, показаною на мал. 10.6, існує відмінність: алгоритм управління не вбудований в обробник повідомлень і подій. Підсистеми визначають, які події їм потрібні, а обробник повідомлень і подій стежить, щоб дані події були відправлені саме їм.




*Мал. 10.7. Модель управління, заснована на передачі повідомлень*

Всі події можуть генерувати повідомлення всіх підсистем, але при цьому значно збільшується навантаження при обробці даних. Часто обробник подій і повідомлень управляє регістром підсистем і подіями, на які вони реагують. Підсистеми генерують події, в яких, можливо, є дані для обробки. Обробник реєструє подію, зважаючи на його регістр, і передає цю подію в ті підсистеми, які на нього реагують.

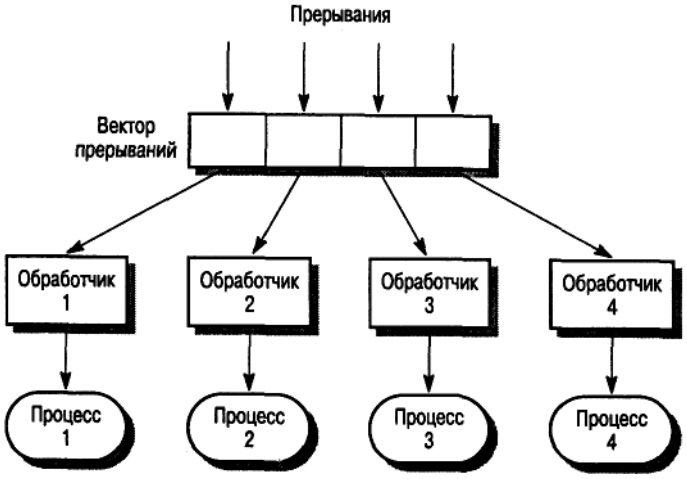
Обробник події завжди підтримує двоточкову взаємодію. Тому підсистеми можуть явно відправити повідомлення іншої підсистеми. Існує безліч різновидів цієї моделі [291, 114, 14*]. Брокери запитів до об'єктів, що обговорюються в розділі 11, також підтримують дану модель управління при взаємодії між розподіленими об'єктами.

Перевагою моделі передачі повідомлень є відносно проста модернізація систем, побудованих відповідно до цієї моделі. Нову підсистему можна інтегрувати в систему, реєструючи її події в обробнику подій. Кожна підсистема може активізувати будь-яку іншу підсистему, не знаючи її імені або розміщення. Підсистеми також можна реалізувати на різних машинах.

Недоліком даної моделі є те, що підсистемам невідоме, коли відбудеться обробка події. Генеруючи подію, підсистема не знає, яка саме система прореагує на нього. Цілком допустима ситуація, коли різні підсистеми реагують на однакові події. Це може привести до конфліктів при діставанні доступу до результатів обробки події.

Системи реального часу, в яких однією з вимог є швидка обробка зовнішніх подій, повинні бути подієво-керованими. Наприклад, система реального часу, що управляє системою безпеки автомобіля, повинна визначити можливу аварію і встигнути наповнити повітрям подушку безпеки до того, як голова водія удариться об кермо. Для того, щоб забезпечити швидку реакцію на події, необхідно використовувати управління, засноване на перериваннях.

На мал. 10.8 показана модель управління, заснована на перериваннях. Для кожного типу переривань існує свій обробник. Кожен тип переривання асоціюється з елементом пам'яті, в якій зберігається адреса обробника переривання. При отриманні певного переривання апаратний перемикач негайно передає управління обробникові переривання. У відповідь на подію, викликану перериванням, обробник може запустити або завершити інші процеси.




*Мал. 10.8. Модель управління, заснована на перериваннях*

Дана модель використовується тільки в жорстких системах реального часу, де потрібна негайна реакція на певні події. Можна скомбінувати цю модель з моделлю централізованого управління. Центральний диспетчер обробляє нормальний хід виконання системи, а в критичних ситуаціях використовується управління, засноване на перериваннях.

Перевагою такого підходу є миттєва реакція системи на події, що відбуваються, недоліками – складність програмування і атестації системи. Практично неможливо імітувати всі переривання в процесі тестування системи. Складно змінювати системи, розроблені на основі такої моделі, оскільки число переривань обмежене апаратурою. Ніякі інші типи подій не обробляються, якщо досягнута ця межа. Обмеження іноді можна обійти, якщо на одному перериванні визначити декілька типів подій і надати для їх обробки окремий обробник. Проте, якщо потрібна дуже швидка реакція на переривання, такий підхід виявляється непрактичним.

## 10.3. Модульна декомпозиція

Після етапу розробки системної структури в процесі проектування слідує етап декомпозиції підсистем на модулі. Між розбиттям системи на підсистеми і підсистем на модулі немає принципових відмінностей. На цьому етапі можна використовувати моделі, розглянуті в розділі 10.1. Проте компоненти модулів звичайні менше компонентів підсистем, тому можна використовувати спеціальні моделі декомпозиції.

Тут розглядаються дві моделі, використовувані на етапі модульної декомпозиції підсистем.

1. *Об'єктно-орієнтована модель.* Система складається з набору взаємодіючих об'єктів.
2. *Модель потоків даних.* Система складається з функціональних модулів, які отримують на вході дані і перетворять їх деяким чином у вихідні дані. Такий підхід часто називається конвеєрним.

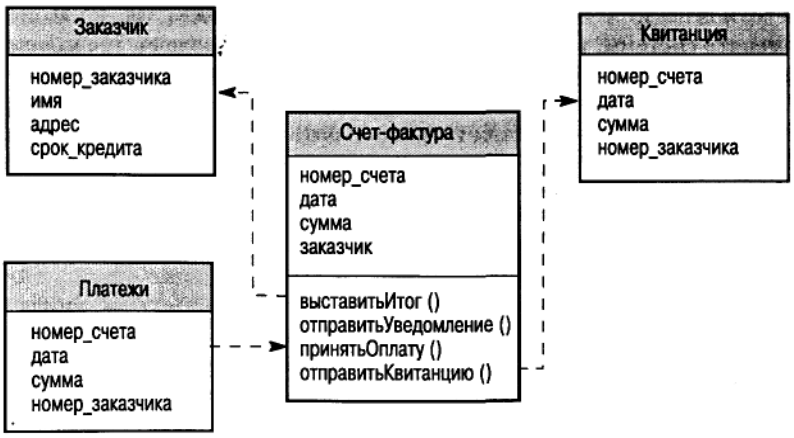
У об'єктно-орієнтованій моделі модулями є об'єкти з власними станами і певними операціями над цими станами. У моделі потоків даних модулі виконують функціональні перетворення. У обох моделях модулі реалізовані або як послідовні компоненти, або як процеси.

По можливості розробникам не варто ухвалювати поспішних рішень про те, чи буде система паралельною або послідовною. Проектування послідовної системи має ряд переваг: послідовні програми легко проектувати, реалізувати, перевіряти і тестувати, чим паралельні системи, де дуже складно формалізувати, управляти і перевіряти тимчасові залежності між процесами. Краще спочатку розбити систему на модулі, а на етапі реалізації вирішити, як організувати їх виконання – послідовно або паралельно.

### 10.3.1. Об'єктні моделі

Об'єктно-орієнтована архітектурна модель структурує систему у вигляді сукупності слабо зв'язаних об'єктів з чітко певними інтерфейсами. Об'єкти викликають сервіси, що надаються іншими об'єктами. З деякими об'єктними моделями ви познайомилися в розділі 7, детальніше вони розглядаються в розділі 12.

На мал. 10.9 представлений приклад об'єктно-орієнтованої архітектурної моделі для системи обробки рахунків. Дана система виписує рахунку замовникам, отримує платежі, відправляє квитанції по платежах, що поступили, і повідомлення по неоплачених рахунках. В даному прикладі використовується система нотації мови моделювання UML (див. розділ 7), в якій класи об'єктів мають імена і набір атрибутів. Операції, якщо вони є, визначаються в нижній частині прямокутника, що позначає об'єкт. Штрихові стрілки означають, що об'єкти використовують властивості або сервіси, що надаються іншими об'єктами.




*Мал. 10.9. Об'єктна модель системи обробки рахунків*

На етапі об'єктно-орієнтованої декомпозиції визначаються класи об'єктів, їх властивості і операції. При реалізації системи з цих класів створюються об'єкти; для координації операцій об'єктів використовується яка-небудь модель управління. У нашому конкретному прикладі клас Рахівниць має різні зв'язані операції (методи), які реалізують функціональні засоби системи. Цей клас використовує інші класи, що представляють замовників, платежі і квитанції.

Переваги об'єктно-орієнтованого підходу добре відомі. Оскільки об'єкти слабо зв'язані між собою, можна змінювати реалізацію того або іншого об'єкту, не впливаючи на решту об'єктів. Структуру системи легко зрозуміти, оскільки об'єкти часто є об'єктами реального миру. Для безпосередньої реалізації системних компонентів можна використовувати об'єктно-орієнтовані мови програмування.

Разом з тим об'єктно-орієнтований підхід має і недоліки. При використанні сервісів об'єкти повинні явно посилатися на імена інших об'єктів і знати їх інтерфейс. Якщо при зміні системи потрібно змінити інтерфейс, необхідно оцінити ефект від такої зміни з урахуванням всіх користувачів змінного об'єкту. Багато об'єктів реального миру складно представити у вигляді системних об'єктів.

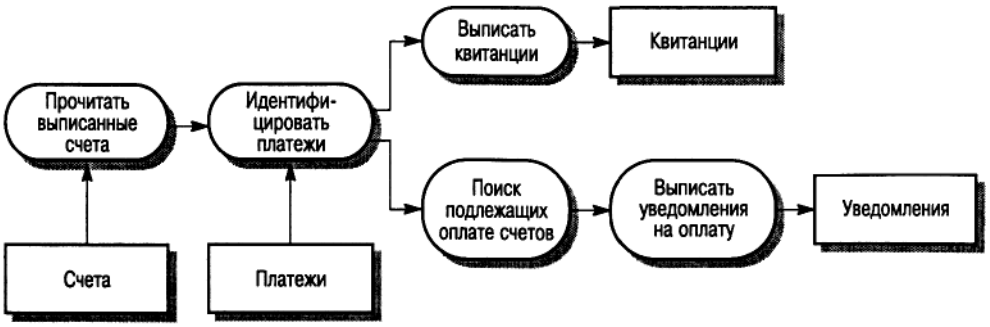
### 10.3.2. Моделі потоків даних

У керованій потоками даних моделі дані проходять через послідовність перетворень. Кожен крок обробки даних реалізований у вигляді перетворення. Дані, системи, що поступають на вхід, проходять через всі перетворення і досягають виходу системи. Перетворення можуть виконуватися послідовно або паралельно. Обробка даних може бути пакетною або поелементною.

Якщо перетворення представлені у вигляді окремих процесів, таку модель іноді називають конвеєром або моделлю фільтрів, слідуючи термінології, прийнятій в системі Unix. Остання підтримує конвеєри, які діють як сховища дані, і набір команд, що представляють функціональні перетворення. Тут використовується термін "фільтр", оскільки перетворення "фільтрує" дані під час обробки потоку даних.

Різні варіанти моделі потоків даних виникли разом з появою перших комп'ютерів, призначених для автоматизованої обробки даних. Коли перетворення послідовно обробляють пакети даних, така архітектурна модель називається пакетною послідовною моделлю. Вона є основою для багатьох класів систем обробки даних. Прикладом можуть бути системи (наприклад, системи обробки рахунків), які генерують велику кількість вихідних звітів, отриманих за допомогою нескладних обчислень, але з великою кількістю вхідних записів.

Приклад такого типу системної архітектури показаний на мал. 10.10. Тут організація виписує рахунку замовникам. Раз на тиждень платіжні квитанції узгоджуються з рахунками. Для сплачених рахунків видається квитанція. По рахунках, не сплачених протягом встановленого терміну, видається відповідне повідомлення.




*Мал. 10.10. Модель потоків даних для системи обробки рахунків*

Відзначимо, що дана модель представляє тільки частину системи обробки рахунків – при виписці рахунків використовуються інші перетворення. Порівняєте дану модель з об'єктно-орієнтованою, розглянутою в попередньому розділі. Об'єктна модель абстрактніша, оскільки в ній не міститься інформація про послідовність дій.

Дана архітектура має ряд переваг.

1. Можливість повторного використання перетворень.
2. Зрозумілість, оскільки більшість людей також мислять в термінах обробки вхідних і вихідних даних.
3. Можливість модифікації системи шляхом безпосереднього додавання нових перетворень.
4. Простота реалізації як послідовною, так і паралельною систем.

Принциповий недолік моделі пов'язаний з необхідністю використання деякого загального формату передачі даних, який повинен розпізнаватися всіма перетвореннями. Кожне перетворення або слід погоджувати з суміжними перетвореннями щодо формату оброблюваних даних, або потрібно запропонувати стандартний формат для всіх оброблюваних даних. Кожне перетворення повинне виконувати граматичний розбір вхідних даних і синтезувати вихідні дані по відповідній формі, при цьому обчислювальне навантаження на систему зростає. Неможливо інтегрувати перетворення, що використовують несумісні формати даних.

Взаємодіючі системи важко описати за допомогою моделі потоків даних через відсутність прогнозованого потоку оброблюваних даних. Хоча звичайне текстовою введення і вивід можна змоделювати за допомогою моделі потоків даних, графічні інтерфейси користувача мають складніші формати введення-виводу і управління, засноване на різноманітних подіях (наприклад, клацання кнопкою миші або вибір з меню). Переклад їх у форму, сумісну з моделлю потоків даних, є достатньо складним завданням.

## 10.4. Проблемно-завісимиє архітектура

Розглянуті раніше архітектурні моделі є узагальненими. Вони широко застосовуються для багатьох класів додатків. Разом з основними моделями, використовуються архітектурні мо-

делі, характерні для конкретної наочної області додатку. Ці моделі називаються *проблемно-зависимыми архітектурою.*

Можна виділити два типи проблемно-зависимых архітектурних моделей.

1. *Моделі класів систем.* Відображають класи реальних систем, увібравши в себе основні характеристики цих класів. Як правило, архітектурні моделі класів зустрічаються в системах реального часу, наприклад в системах збору даних, моніторингу і так далі
2. *Базові моделі.* Абстрактніші і надають розробникам інформацію по загальній структурі якого-небудь типу систем. Наприклад, в роботі [298] запропонована базова модель розробки ПО.

Звичайно, чітких відмінностей між цими видами моделей немає. В деяких випадках моделі класів служать як базові. Тут я провожу відмінність між ними, оскільки базові моделі можна безпосередньо повторно використовувати в проекті. Базові моделі зазвичай використовуються в системах комунікацій і при порівнянні можливої системної архітектури. Різні також процеси розробки цих моделей. Моделі класів розробляються як узагальнення існуючих систем "від низу до верху", тоді як розробка базових моделей йде зверху "вниз".

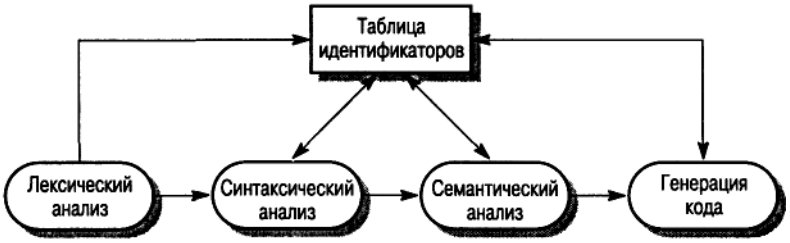
### 10.4.1. Моделі класів систем

Модель компілятора, ймовірно, найбільш відомий приклад архітектурної моделі класу систем. В даний час розроблені тисячі компіляторів. Вважається, що компілятор повинен включати перераховані нижче модулі.

1. Лексичний аналізатор, що транслює вхідну мову в деякий внутрішній код.
2. Таблиця ідентифікаторів, що видається лексичним аналізатором, в якій міститься інформація про використовувані в програмі імена і типи.
3. Синтаксичний аналізатор, який перевіряє синтаксис компільованої коди. Він використовує задану граматику мови і створює синтаксичне дерево.
4. Синтаксичне дерево, яке представляє внутрішню структуру компільованої програми.
5. Семантичний аналізатор, який перевіряє семантичну коректність програм на підставі інформації, отриманої з синтаксичного дерева і таблиці ідентифікаторів.
6. Генератор коди, яка проходить по синтаксичному дереву і генерує машинний код.

У компіляторі можуть бути і інші компоненти, які перетворять синтаксичне дерево, підвищують ефективність і усувають надмірність з машинної коди, що згенерувала.

Компоненти, з яких складається компілятор, можна організовувати відповідно до різних архітектурних моделей. Можна застосувати архітектуру потоків даних, в якій таблиця ідентифікаторів служить сховищем спільно використовуваних даних. Цей підхід відображений на мал. 10.11, тут етапи лексичного, синтаксичного і семантичного аналізу виконуються послідовно.

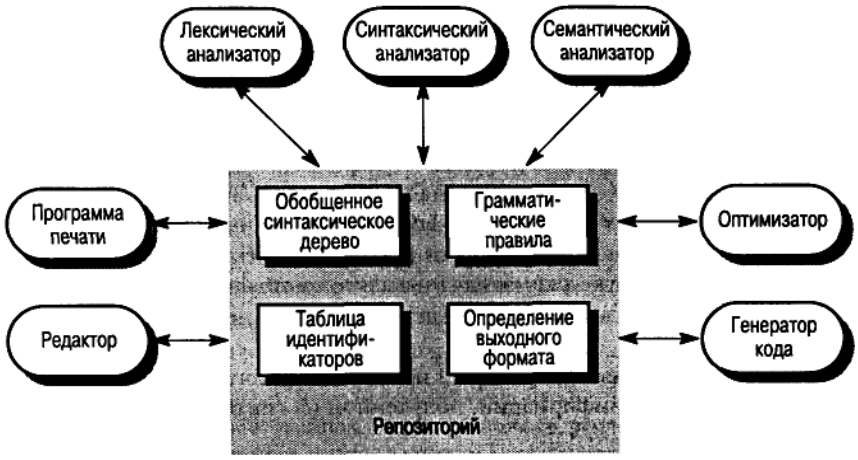



*Мал. 10.11. Модель потоків даних для компілятора*

Така модель до цих пір широко застосовується. Вона ефективна там, де програми компілюються і виконуються без участі користувача, тобто в пакетному режимі. Проте такі моделі виявляються менш ефективними, якщо компілятор інтегрований з іншими мовними засоба-

ми, наприклад системою редагування структур, інтерактивним відладчиком, програмою підготовки друкарських документів і тому подібне В цьому випадку, як показано на мал. 10.12, компоненти системи можна організувати відповідно до моделі репозиторія.

У представленій моделі компілятора таблиця ідентифікаторів і синтаксичне дерево використовуються як центральне сховище інформації, або репозиторій, через який взаємодіють інструментальні засоби. Інші дані, такі як граматичні правила і визначення вихідного формату програми, беруться з інструментальних засобів і також поміщаються в репозиторій.




*Мал. 10.12. Модель репозиторія для компілятора*

В даний час розроблена відносно невелика кількість проблемно-зависимых моделей класів систем. Організації, що займаються розробкою подібних моделей, розглядають їх як цінну інтелектуальну власність, оскільки вони є основою розробки програмних систем. Такі моделі часто представляють архітектуру цілої серії програмних продуктів. Наприклад, для всіх драйверів принтерів, незалежно від властивостей конкретного принтера, може використовуватися однакова початкова архітектура. Подібна архітектура розглядається в розділі 14.

### 10.4.2. Базова архітектура

Архітектурні моделі класів систем відображає архітектура вже існуючих систем. В протилежність їм базові моделі зазвичай з'являються як результат досліджень в певній наочній області додатку. Вони є архітектурою, що ідеалізується, в якій відбиті особливості, властиві системам, що працюють в даній наочній області.

Прикладом базової архітектури може служити модель OSI [352], що є стандартом взаємодії відкритих систем (короткий опис цієї моделі приведений в розділі 10.1.3). Якщо деяка система сумісна з цією моделлю, вона може взаємодіяти з будь-якими іншими системами, що підтримують цей стандарт. Таким чином, система управління асортиментом товарів в супермаркеті, розроблена на основі моделі OSI, може безпосередньо здійснювати обмін даними з системою замовлень постачальників, також розробленою на основі цієї моделі.

З іншого боку, базові моделі зазвичай не розглядаються як методи реалізації. Їх основне призначення – служити еталоном для порівняння різних систем в якій-небудь наочній області, тобто базова модель є стандартом при оцінці різних систем.

Зображена на мал. 10.13 модель OSI є семирівневою моделлю взаємодії відкритих систем. Точне призначення різних шарів тут не істотно. Відмітимо тільки, що нижні рівні забезпечують фізичну взаємодію, середні рівні – передачу даних, а верхні рівні – передачу семантично значущої інформації, наприклад стандартизованих документів і тому подібне


*Мал. 10.13. Архітектура базової моделі OSI*

Перед розробниками моделі OSI була поставлена конкретна мета – створити стандарт, на основі якого здійснювалася б взаємодія між сумісними системами. Кожен рівень залежав би тільки від рівня, що пролягав нижче. У міру розвитку технології будь-яким з рівнів можна було б реалізувати наново, не впливаючи при цьому на інші рівні системи.

На практиці багаторівневий підхід до архітектурного моделювання носить компромісний характер. Якщо відмінності між комп'ютерними мережами дуже великі, проста взаємодія між ними неможлива. Хоча функціональні характеристики кожного рівня чітко визначені, залишаються невизначеними нефункціональні характеристики системи. Тому розробники, реалізовуючи власні засоби високого рівня, можуть "пропустити" деякі рівні моделі. Інакше кажучи, щоб підвищити продуктивність системи, розробники можуть проектувати нестандартні засоби, які не укладаються в рамки моделі. Після цього проста заміна рівнів в такій моделі навряд чи можлива. Проте від цього ефективність моделі не знижується. Дана модель забезпечує основу для загальної структуризації системи і для реалізації взаємодій між системами.

Іншими базовими моделями є CASE-среды [104, 6*] і модель розробки програмного забезпечення [298, 14*]. Деякі архітектурні шаблони [124, 13*, 32*] також можна вважати базовою архітектурою. Ці моделі обговорюються в розділі 14.

## КЛЮЧОВІ ПОНЯТТЯ

- Архітектура ПО – це опис структури програмної системи. Різні архітектурні моделі, такі як структурна модель, модель управління і модель модальної декомпозиції, розробляються архітектурного проектування.
- Великі системи рідко зводяться до однієї архітектурної моделі. Вони неоднорідні і на різних рівнях узагальнення використовують різні моделі.
- До структурних моделей відносяться моделі репозиторія, моделі клієнт/сервер і моделі абстрактної машини. У моделі репозиторія спільно використовувані дані знаходяться в загальному сховищі (репозиторії). У моделях клієнт/сервер дані, як правило, розподілені. Моделі абстрактної машини є багаторівневими, причому кожен рівень реалізований за допомогою засобів, що надаються попереднім рівнем.
- До моделей управління відносяться моделі централізованого управління і моделі, керовані подіями. У централізованих моделях управління залежить від стану системи; у моделях, керованих подіями, системою управляють зовнішні події.
- До моделей модульної декомпозиції відносяться моделі потоків даних і об'єктні моделі. Моделі потоків даних функціональні, тоді як об'єктні моделі є сукупністю слабо зв'язаних між об'єктів з власними станами і операціями.
- Проблемно-завісимиє архітектурні моделі відображають властивості і характеристики прикладних областей. Проблемно-завісимиє моделі діляться на моделі класів систем, що проектуються на основі існуючих систем, і базові моделі, які є абстрактними моделями, що ідеалізуються, для конкретної наочної області.

**Вправи**

**10.1.** Поясните, чому архітектуру системи необхідно розробити до закінчення створення специфікації.

**10.2.** Створіть таблицю, що описує переваги і недоліки різних структурних моделей, що обговорювалися в даному розділі.

**10.3.** Запропонуєте відповідну структурну модель для перерахованих нижче систем. Обгрунтуйте свій вибір.

- Система автоматичного продажу залізничних квитків.
- Система відеоконференцій, керована комп'ютером, з можливістю одночасного перегляду комп'ютерних, аудио- і відеоданих декількома учасниками.
- Робот-прибиральник, який прибирає відносно вільні простори, наприклад коридори. Робот повинен фіксувати стіни і інші перешкоди.

**10.4.** На основі вибраної моделі розробіть архітектуру для систем з попередньої вправи. Зробіть припущення про системні вимоги.

**10.5.** Поясните, чому модель управління виклику-повернення зазвичай не підходить для систем реального часу, керівників певним процесом.

**10.6.** Запропонуєте відповідну модель управління для перерахованих нижче систем. Обгрунтуйте свій вибір.

- Пакетна система обробки даних, яка на підставі інформації про відпрацьований годинник і ставок заробітної плати формує заявки на зарплату персоналу і передає інформацію в банк.
- Набір інструментальних програмних засобів від різних виробників, які повинні працювати спільно.
- Телевізійний контроллер, який відповідає на сигнали, що поступають від видаленого блоку управління.

**10.7.** Обговорите переваги і недоліки моделі потоків даних і об'єктної моделі в припущенні, що необхідно розробити як локальну, так і розподілену версії програмного застосування.

**10.8.** Існує два набори інструментальних CASE-средств. Необхідно порівняти їх. Продумайте, як це зробити за допомогою базової моделі CASE-средств [61].

**10.9.** Припустимо, існує конкретна посада "архітектор програмного забезпечення"; його роль полягає в проектуванні системної архітектури незалежно від того, для якого замовника виконується даний проект. Така посада може бути, наприклад, в компанії, розробкою, що займається, ПО. Які труднощі можуть виникнути при введенні даної посади?

# 11. Архітектура розподілених систем

**Цілі**

Мета справжнього розділу – вивчення архітектури розподілених програмних систем. Прочитавши цей розділ, ви винні:

- ❑ знати основні переваги і недоліки розподілених систем;
- ❑ мати уявлення про різні підходи, використовувані при розробці архітектури клієнт/сервер;
- ❑ розуміти відмінності між архітектурою клієнт/сервер і архітектурою розподілених об'єктів;
- ❑ знати концепцію брокера запитів до об'єктів і принципи, реалізовані в стандартах CORBA.

В даний час практично всі великі програмні системи є розподіленими. Розподіленою називається така система, в якій обробка інформації зосереджена не на одній обчислювальній машині, а розподілена між декількома комп'ютерами. При проектуванні розподілених систем, яке має багато загального з проектуванням будь-якого іншого ПО, все ж таки слід враховувати ряд специфічних особливостей. Деякі з них вже згадувалося у введенні до розділу 10 при розгляді архітектури клієнт/сервер, тут вони обговорюються детальніше.

Оскільки в наші дні розподілені системи набули широкого поширення, розробники ПО повинні бути знайомі з особливостями їх проектування. До недавнього часу всі великі системи в основному були централізованими, які запускалися на одній головній обчислювальній машині (мейнфреймі) з підключеними до неї терміналами. Термінали практично не займали-

ся обробкою інформації – всі обчислення виконувалися на головній машині. Розробникам таких систем не доводилося замислюватися про проблеми розподілених обчислень.

Всі сучасні програмні системи можна розділити на три великі класи.

1. Прикладні програмні системи, призначені для роботи тільки на одному персональному комп'ютері або робочій станції. До них відносяться текстові процесори, електронні таблиці, графічні системи і тому подібне
2. Вбудовані системи, призначені для роботи на одному процесорі або на інтегрованій групі процесорів. До них відносяться системи управління побутовими пристроями, різними приладами і ін.
3. Розподілені системи, в яких програмне забезпечення виконується на слабо інтегрованій групі паралельно працюючих процесорів, зв'язаних через мережу. До них відносяться системи банкоматів, що належать якому-небудь банку, видавничі системи, системи ПО колективного користування і ін.

В даний час між перерахованими класами програмних систем існують чіткі межі, які надалі все більш стиратимуться. З часом, коли високошвидкісні безпровідні мережі стануть широкодоступними, з'явиться можливість динамічно інтегрувати пристрої з вбудованими програмними системами, наприклад електронні органайзери з більш загальними системами.

У книзі [81] виділено шість основних характеристик розподілених систем.

1. *Сумісне використання ресурсів.* Розподілені системи допускають сумісне використання апаратних і програмних ресурсів, наприклад жорстких дисків, принтерів, файлів, компіляторів і тому подібне, зв'язаних за допомогою мережі. Очевидно, що розділення ресурсів можливе також в многопользовательских системах, проте в цьому випадку за надання ресурсів і їх управління повинен відповідати центральний комп'ютер.
2. *Відвертість.* Це можливість розширювати систему шляхом додавання нових ресурсів. Розподілені системи – це відкриті системи, до яких підключають апаратне і програмне забезпечення від різних виробників.
3. *Паралельність.* У розподілених системах декілька процесів можуть одночасно виконуватися на різних комп'ютерах в мережі. Ці процеси можуть (але не обов'язково) взаємодіяти один з одним під час їх виконання.
4. *Масштабованість.* В принципі всі розподілені системи є масштабованими: щоб система відповідала новим вимогам, її можна нарощувати за допомогою додавання нових обчислювальних ресурсів. Але на практиці нарощування може обмежуватися мережею, об'єднуючою окремі комп'ютери системи. Якщо підключити багато нових машин, пропускна спроможність мережі може виявитися недостатньою.
5. *Відмовостійка.* Наявність декількох комп'ютерів і можливість дублювання інформації означає, що розподілені системи стійкі до певних апаратних і програмних помилок (див. розділ 18). Більшість розподілених систем у разі помилки, як правило, можуть підтримувати хоч би часткову функціональність. Повний збій в роботі системи відбувається тільки у разі мережевих помилок.
6. *Прозорість.* Ця властивість означає, що користувачам наданий повністю прозорий доступ до ресурсів і в той же час від них прихована інформація про розподіл ресурсів в системі. Проте у багатьох випадках конкретні знання про організацію системи допомагають користувачеві краще використовувати ресурси.

Зрозуміло, розподіленим системам властивий ряд недоліків.

- *Складність.* Розподілені системи складніше централізованих. Набагато важче зрозуміти і оцінити властивості розподілених систем в цілому, а також тестувати ці системи. Наприклад, тут продуктивність системи залежить не від швидкості роботи одного процесора, а від смуги пропускання мережі і швидкості роботи різних процесорів. Перемі-

щаючи ресурси з однієї частини системи в іншу, можна радикально вплинути на продуктивність системи.

- *Безпека.* Зазвичай доступ до системи можна отримати з декількох різних машин, повідомлення в мережі можуть бути видимими або перехоплюватися. Тому, в розподіленій системі набагато складніше підтримувати безпеку.
- *Керованість.* Система може складатися з різнотипних комп'ютерів, на яких можуть бути встановлені різні версії операційних систем. Помилки на одній машині можуть розповсюдитися на інші машини з непередбачуваними наслідками. Тому потрібно значно більше зусиль, щоб управляти і підтримувати систему в робочому стані.
- *Непередбачуваність.* Як відомо всім користувачам Web-сети, реакція розподілених систем на певні події непередбачувана і залежить від повного завантаження системи, її організації і мережевого навантаження. Оскільки всі ці параметри можуть постійно мінятися, час, витрачений на виконання запиту користувача, в той або інший момент може істотно розрізнятися.

При обговоренні переваг і недоліків розподілених систем в книзі [81] визначається ряд критичних проблем проектування таких систем (табл. 11.1). У цьому розділі основна увага приділяється архітектурі розподіленого ПО, оскільки я вважаю, що при розробці програмних продуктів найбільш значущим є саме цей момент. Якщо вас цікавлять інші теми, звернетеся до спеціалізованих книг по розподілених системах.

**Таблиця 11.1. Проблеми проектування розподілених систем**

| Проблема проектування | Опис |
|---|---|
| Ідентифікація ресурсів | Ресурси в розподіленій системі розташовуються на різних комп'ютерах, тому систему імен ресурсів слід продумати так, щоб користувачі могли без зусиль відкривати необхідні ним ресурси і посилатися на них. Прикладом може служити система уніфікованого покажчика ресурсів URL, яка визначає адреси Web-страниц. Без легковоспринимаемой і універсальної системи ідентифікації велика частина ресурсів виявиться недоступною користувачам системи |
| Комунікації | Універсальна працездатність Internet і ефективна реалізація протоколів TCP/IP в Internet для більшості розподілених систем служать прикладом найбільш ефективного способу організації взаємодії між комп'ютерами. Проте там, де на продуктивність, надійність і інше накладаються спеціальні вимоги, можна скористатися альтернативними способами системних комунікацій |
| Якість системного сервісу | Якість сервісу, пропонована системою, відображає її продуктивність, працездатність і надійність. На якість сервісу впливає цілий ряд чинників: розподіл системних процесів, розподіл ресурсів, системні і мережеві апаратні засоби і можливості адаптації системи |
| Архітектура програмного забезпечення | Архітектура програмного забезпечення описує розподіл системних функцій по компонентах системи, а також розподіл цих компонентів по процесорах. Якщо необхідно підтримувати високу якість системного сервісу, вибір правильної архітектури виявляється вирішальним чинником |

Завдання розробників розподілених систем – спроектувати програмне або апаратне забезпечення так, щоб надати всі необхідні характеристики розподіленої системи. А для цього потрібно знати переваги і недоліки різної архітектури розподілених систем. Тут виділяється два споріднені типи архітектури розподілених систем.

1. *Архітектура клієнт/сервер.* У цій моделі систему можна представити як набір сервісів, що надаються серверами клієнтам. У таких системах сервери і клієнти значно відрізняються один від одного.
2. *Архітектура розподілених об'єктів.* В цьому випадку між серверами і клієнтами немає відмінностей і систему можна представити як набір взаємодіючих об'єктів, місцеположення яких не має особливого значення. Між постачальником сервісів і їх користувачами не існує відмінностей.

У розподіленій системі різні системні компоненти можуть бути реалізовані на різних мовах програмування і виконуватися на різних типах процесорів. Моделі даних, представлення інформації і протоколи взаємодії – все це не обов'язково буде однотипним в розподіленій системі. Отже, для розподілених систем необхідне таке програмне забезпечення, яке могло б управляти цими різнотипними частинами і гарантувати взаємодію і обмін даними між ними. *Проміжне програмне забезпечення* відноситься саме до такого класу ПО. Воно знаходиться як би посередині між разными частями розподілених компонентів системи.

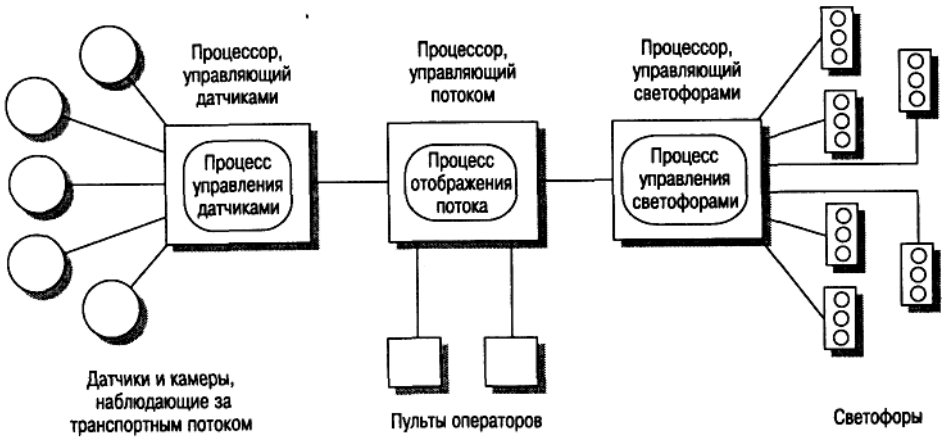
У статті [37] описані різні типи проміжного ПО, яке може підтримувати розподілені обчислення. Як правило, таке ПО складається з готових компонентів і не вимагає від розробників спеціальних доопрацювань. Як приклади проміжного ПО можна привести програми управління взаємодією з базами даних, менеджери транзакцій, перетворювачі даних, комунікаційні інспектори і ін. Далі в розділі буде описана структура розподілених систем як клас проміжний ПО.

Розподілені системи зазвичай розробляються на основі об'єктно-орієнтованого підходу. Ці системи створюються із слабо інтегрованих частин, кожна з яких може безпосередньо взаємодіяти як з користувачем, так і з іншими частинами системи. Ці частини по можливості повинні реагувати на незалежні події. Програмні об'єкти, побудовані на основі таких принципів, є природними компонентами розподілених систем. Якщо ви ще не знайомі з концепцією об'єктів, рекомендую спочатку прочитати розділ 12, а потім знов повернутися до даного розділу.

## 11.1. Багатопроцесорна архітектура

Найпростішою розподіленою системою є багатопроцесорна система. Вона складається з безлічі різних процесів, які можуть (але не обов'язково) виконуватися на різних процесорах. Дана модель часто використовується у великих системах реального часу. Як ви дізнаєтеся з розділу 13, ці системи збирають інформацію, ухвалюють на її основі рішення і відправляють сигнали виконавчому механізму, який змінює системне оточення. В принципі всі процеси, пов'язані із збором інформації, ухваленням рішень і управлінням виконавчим механізмом, можуть виконуватися на одному процесорі під управлінням планувальника завдань. Використання декількох процесорів підвищує продуктивність системи і її здатність до відновлення. Розподіл процесів між процесорами може перевизначатися (властиво критичним системам) або ж знаходитися під управлінням диспетчера процесів.

На мал. 11.1 показаний приклад системи такого типу. Це спрощена модель системи управління транспортним потоком. Група розподілених датчиків збирає інформацію про величину потоку. Зібрані дані перед відправкою в диспетчерську обробляються на місці. На підставі отриманої інформації операторів приймають рішення і управляють світлофорами. В даному прикладі для управління датчиками, диспетчерською і світлофорами є окремі логічні процеси. Це можуть бути як окремі процеси, так і група процесів. У нашому прикладі вони виконуються на різних процесорах.

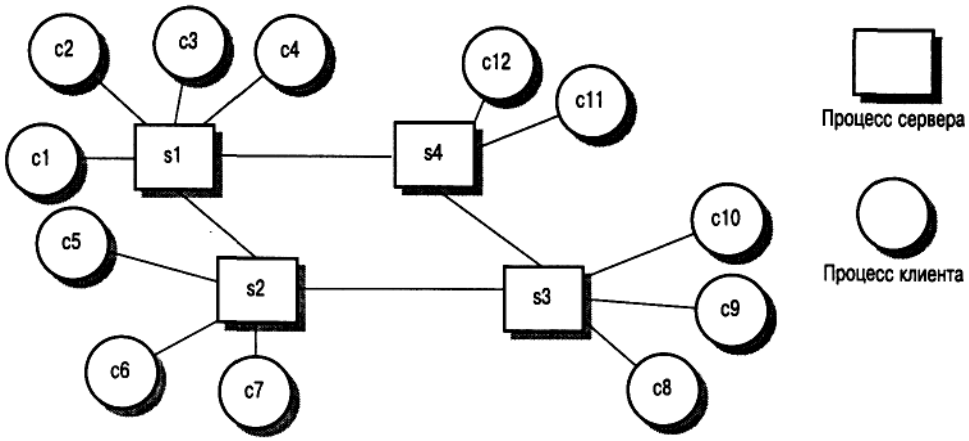



*Мал. 11.1. Багатопроцесорна система управління рухом транспорту*

Системи ПО, що одночасно виконують безліч процесів, не обов'язково є розподіленими. Якщо в системі більш за один процесор, реалізувати розподіл процесів не представляє праці. Проте при створенні багатопроцесорних програмних систем не обов'язково відштовхуватися тільки від розподілених систем. При проектуванні систем такого типу, по суті, використовується той же підхід, що і при проектуванні систем реального часу, які розглядаються в розділі 13.

## 11.2. Архітектура клієнт/сервер

В розділі 10 вже розглядалася концепція клієнт/сервер. У архітектурі клієнт/сервер програмне застосування моделюється як набір сервісів, що надаються серверами, і безліч клієнтів, що використовують ці сервіси [264, 2*]. Клієнти повинні знати про доступні (що є) сервери, хоча можуть і не мати уявлення про існування інших клієнтів. Як видно з мал. 11.2, на якому представлена схема розподіленої архітектури клієнт/сервер, клієнти і сервери представляють різні процеси.




*Мал. 11.2. Система клієнт/сервер*

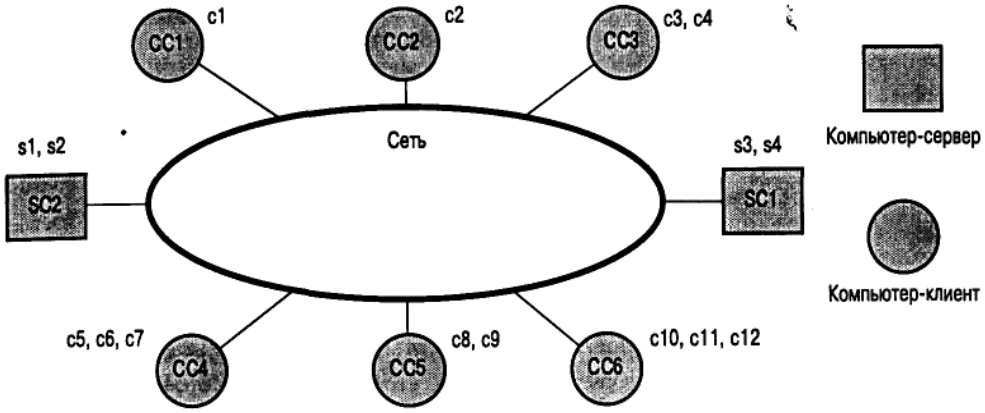
У системі між процесами і процесорами не обов'язково повинне дотримуватися відношення "один до одного". На мал. 11.3 показана фізична архітектура системи, яка складається з шести клієнтських машин і двох серверів. На них запускаються клієнтські і серверні процеси, зображені на мал. 11.2. У загальному випадку, кажучи про клієнтів і сервери, я маю на увазі швидше логічні процеси, чим фізичні машини, на яких виконуються ці процеси.

Архітектура системи клієнт/сервер повинна відображати логічну структуру програмного застосування, що розробляється. На мал. 11.4 пропонується ще один погляд на програмне застосування, структуроване у вигляді трьох рівнів. Рівень уявлення забезпечує інформацію

для користувачів і взаємодію з ними. Рівень виконання додатку реалізує логіку роботи додатку. На рівні управління даними виконуються всі операції з базами даних. У централізованих системах між цими рівнями немає чіткого розділення. Проте при проектуванні розподілених систем необхідно розділяти ці рівні, щоб потім розташувати кожен рівень на різних комп'ютерах.

Найпростішою архітектурою клієнт/сервер є дворівнева, в якій додаток складається з сервера (або безліч ідентичних серверів) і групи клієнтів. Існує два види такої архітектури (мал. 11.5).

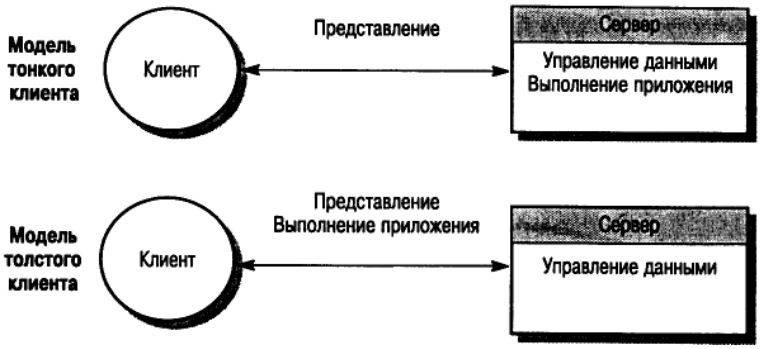
1. *Модель тонкого клієнта.* У цій моделі вся робота додатку і управління даними виконуються на сервері. На клієнтській машині запускається тільки ПО рівня уявлення.
2. *Модель товстого клієнта.* У цій моделі сервер тільки управляє даним. На клієнтській машині реалізована робота додатку і взаємодія з користувачем системи.




*Мал. 11.3. Комп'ютери в мережі клієнт/сервер*


*Мал. 11.4. Рівні програмного застосування*

Тонкий клієнт дворівневої архітектури – найпростіший спосіб перекладу існуючих централізованих систем (див. розділ 26) в архітектуру клієнт/сервер. Призначений для користувача інтерфейс в цих системах "переселяється" на персональний комп'ютер, а само програмне застосування виконує функції сервера, тобто виконує всі процеси додатку і управляє даними. Модель тонкого клієнта можна також реалізувати там, де клієнти є звичайними мережевими пристроями, а не персональними комп'ютерами або робочими станціями. Мережеві пристрої запускають Internet-броузер і призначений для користувача інтерфейс, реалізований усередині системи.

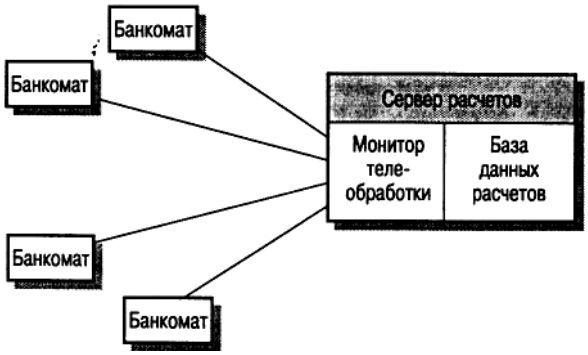



*Мал. 11.5. Моделі тонкого і товстого клієнтів*

Головний недолік моделі тонкого клієнта – велика завантаженість сервера і мережі. Всі обчислення виконуються на сервері, а це може привести до значного мережевого трафіку між клієнтом і сервером. У сучасних комп'ютерах достатньо обчислювальної потужності, але вона практично не використовується в моделі тонкого клієнта банку.

Навпаки, модель товстого клієнта використовує обчислювальну потужність локальних машин: і рівень виконання додатку, і рівень уявлення поміщаються на клієнтський комп'ютер. Сервер тут, по суті, є сервером транзакцій, який управляє всіма транзакціями баз даних. Прикладом архітектури такого типу можуть служити системи банкоматів, в яких банкомат є клієнтом, а сервер – центральним комп'ютером, обслуговуючим базу даних по розрахунках з клієнтами.

На мал. 11.6 показана мережева система банкоматів. Відмітимо, що банкомати пов'язані з базою даних розрахунків не безпосередньо, а через монітор телеобробки. Цей монітор є проміжною ланкою, яка взаємодіє з видаленими клієнтами і організовує запити клієнтів в послідовність транзакцій для роботи з базою даних. Використання послідовних транзакцій при виникненні збоїв дозволяє системі відновитися без втрати даних.

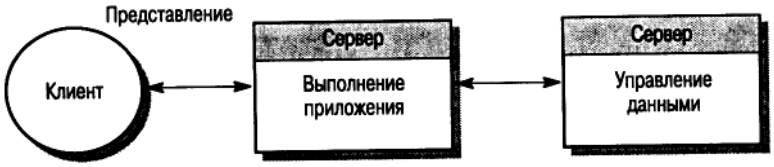



*Мал. 11.6. Система клієнт/сервер для мережі банкоматів*

Оскільки в моделі товстого клієнта виконання програмного застосування організоване ефективніше, ніж в моделі тонкого клієнта, управляти такою системою складніше. Тут функції додатку розподілені між безліччю різних машин. Необхідність заміни додатку приводить до його повторної інсталяції на всіх клієнтських комп'ютерах, що вимагає великих витрат, якщо в системі сотні клієнтів.

Появу мови Java і завантажуваних аплетов дозволили розробляти моделі клієнт/сервер, які знаходяться десь посередині між моделями тонкого і товстого клієнта. Частина програм, складових додаток, можна завантажувати на клієнтській машині як аплеты Java і тим самим розвантажити сервер. Інтерфейс користувача будується за допомогою Web-броузера, який запускає аплеты Java. Проте Web-броузеры від різних виробників і навіть різні версії Web-броузеров від одного виробника не завжди виконуються однаково. Раніші версії броузеров на старих машинах не завжди можуть запустити аплеты Java. Отже, такий підхід можна використовувати тільки тоді, коли ви упевнені, що у всіх користувачів системи встановлені броузеры, сумісні з Java.

У дворівневій моделі клієнт/сервер істотною проблемою є розміщення на двох комп'ютерних системах трьох логічних рівнів – уявлення, виконання додатку і управління даними. Тому в даній моделі часто виникають або проблеми з масштабованістю і продуктивністю, якщо вибрана модель тонкого клієнта, або проблеми, пов'язані з управлінням системою, якщо використовується модель товстого клієнта. Щоб уникнути цих проблем, необхідно застосувати альтернативний підхід – трирівневу модель архітектури клієнт/сервер (мал. 11.7). У цій архітектурі рівням уявлення, виконання додатку і управління даними відповідають окремі процеси.



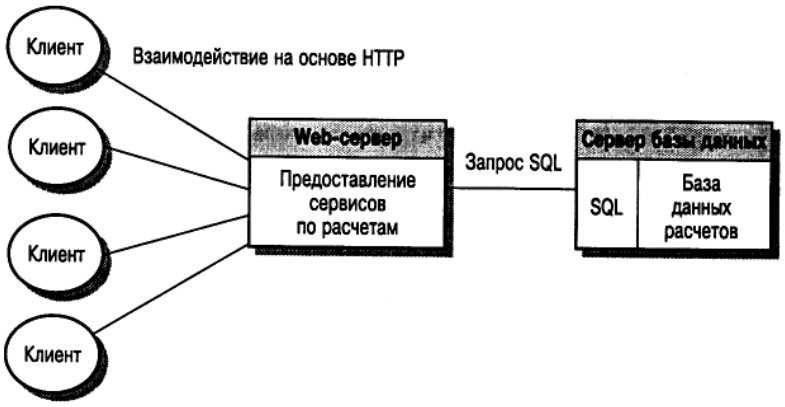

*Мал. 11.7. Трирівнева архітектура клієнт/сервер*

Архітектура ПО, побудована по трирівневій моделі клієнт/сервер, не вимагає, щоб в мережу були об'єднані три комп'ютерні системи. На одному комп'ютері-сервері можна запустити і виконання додатку, і управління даними як окремі логічні сервери. В той же час, якщо вимоги до системи зростуть, можна буде відносно просто розділити виконання додатку і управління даними і виконувати їх на різних процесорах.

Банківську систему, що використовує Internet-сервисы, можна реалізувати за допомогою трирівневої архітектури клієнт/сервер. База даних розрахунків (зазвичай розташована на головному комп'ютері) надає сервіси управління даними, Web-сервер підтримує сервіси додатку, наприклад засоби переказу грошей, генерацію звітів, оплату рахунків і ін. А комп'ютер користувача з Internet-броузером є клієнтом. Як показано на мал. 11.8, ця система масштабована, оскільки в неї відносно просто додати нові Web-серверы при збільшенні кількості клієнтів.

Використання трирівневої архітектури в даному прикладі дозволило оптимізувати передачу даних між Web-сервером і сервером бази даних. Взаємодія між цими системами не обов'язково будувати на стандартах Internet, можна використовувати швидші комунікаційні протоколи низького рівня. Зазвичай інформацію від бази даних обробляє ефективне проміжне ПО, яке підтримує запити до бази даних на мові структурованих запитів SQL.

В деяких випадках трирівневу модель клієнт/сервер можна перевести в багаторівневу, додавши в систему додаткові сервери. Багаторівневі системи можна використовувати і там, де додаткам необхідно мати доступ до інформації, що знаходиться в різних базах даних. В цьому випадку об'єднуючий сервер розташовується між сервером, на якому виконується додаток, і серверами баз даних. Об'єднуючий сервер збирає розподілені дані і представляє їх в додатку таким чином, ніби вони знаходяться в одній базі даних.




*Мал. 11.8. Розподілена архітектура банківської системи з використанням Internet-сервисов*

Розробники архітектури клієнт/сервер, вибираючи найбільш відповідну, повинні враховувати ряд чинників. У табл. 11.2 перераховані різні випадки застосування архітектури клієнт/сервер.

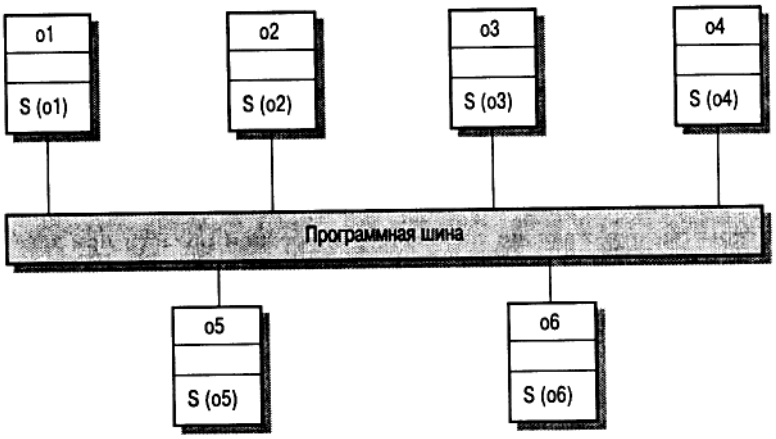
**Таблиця 11.2. Застосування різних типів архітектури клієнт/сервер**

| Архітектура | Додатки |
|---|---|
| Дворівнева архітектура тонкого клієнта | Успадковані системи, в яких недоцільно розділяти виконання додатку і управлін даними. |
| | Додатки з інтенсивними обчисленнями, наприклад компілятори, але з незначн об'ємом управління даними. |
| | Додатки, в яких обробляються великі масиви даних (запити), але з невеликим о ємом обчислень в самому застосуванні |
| Дворівнева архітектура товстого клієнта | Додатки, де користувачеві потрібна інтенсивна обробка даних (наприклад, візуа зація даних або великі об'єми обчислення). |
| | Додатки з відносно постійним набором функцій на стороні користувача, вживан в середовищі з добре відладженим системним управлінням |
| Трирівнева і багаторівнева архітектура клієнт/сервер | Великі застосування з сотнями і тисячами клієнтів. |
| | Додатки, в яких часто міняються і дані, і методи обробки. |
| | Додатки, в яких виконується інтеграція даних з багатьох джерел |

## 11.3. Архітектура розподілених об'єктів

У моделі клієнт/сервер розподіленої системи між клієнтами і серверами існують відмінності. Клієнт запрошує сервіси тільки у сервера, hq не у інших клієнтів; сервери можуть функціонувати як клієнти і запрошувати сервіси у інших серверів, але не у клієнтів; клієнти повинні знати про сервіси, що надаються певними серверами, і про те, як взаємодіють ці сервери. Така модель відмінно підходить до багатьом типам додатків, але в той же час обмежує розробників системи, які вимушені вирішувати, де надавати сервіси. Вони також повинні забезпечити підтримку масштабованості і розробити засоби включення клієнтів в систему на розподілених серверах.

Більш загальним підходом, вживаним в проектуванні розподілених систем, є стирання відмінностей між клієнтом і сервером і проектування архітектури системи як архітектура розподілених об'єктів. У цій архітектурі (мал. 11.9) основними компонентами системи є об'єкти, що надають набір сервісів через свої інтерфейси. Інші об'єкти викликають ці сервіси, не роблячи відмінностей між клієнтом (користувачем сервісу) і сервером (постачальником сервісу).




*Мал. 11.9. Архітектура розподілених об'єктів*

Об'єкти можуть розташовуватися на різних комп'ютерах в мережі і взаємодіяти за допомогою проміжного ПО. По аналогії з системною шиною, яка дозволяє підключати різні пристрої і підтримувати взаємодію між апаратними засобами, проміжну ПО можна розглядати як шину програмного забезпечення. Вона надає набір сервісів, що дозволяє об'єктам взаємодіяти один з одним, додавати або видаляти їх з системи. Проміжне ПО називають брокером запитів до об'єктів. Його завдання – забезпечувати інтерфейс між об'єктами. Брокери запитів до об'єктів розглядаються в розділі 11.4.

Нижче перераховані основні переваги моделі архітектури розподілених об'єктів.

- Розробники системи можуть не поспішати з ухваленням рішень щодо того, де і як надаватимуться сервіси. Об'єкти, що надають сервіси, можуть виконуватися в будь-якому місці (вузлі) мережі. Отже, відмінність між моделями товстого і тонкого клієнтів стають неістотними, оскільки немає необхідності заздалегідь планувати розміщення об'єктів для виконання додатку.
- Системна архітектура достатньо відкрита, що дозволяє при необхідності додавати в систему нові ресурси. У наступному розділі наголошується, що стандарти програмної шини постійно удосконалюються, що дозволяє об'єктам, написаним на різних мовах програмування, взаємодіяти і надавати сервіси один одному.
- Гнучкість і масштабованість системи. Для того, щоб справитися з системними навантаженнями, можна створювати екземпляри системи з однаковими сервісами, які надаватимуться різними об'єктами або різними екземплярами (копіями) об'єктів. При збільшенні навантаження в систему можна додати нові об'єкти, не перериваючи при цьому роботу інших її об'єктів.
- Існує можливість динамічно переконфігурувати систему за допомогою об'єктів, мігруючих в мережі по запитах. Об'єкти, що надають сервіси, можуть мігрувати на той же процесор, що і об'єкти, що запрошують сервіси, тим самим підвищуючи продуктивність системи.

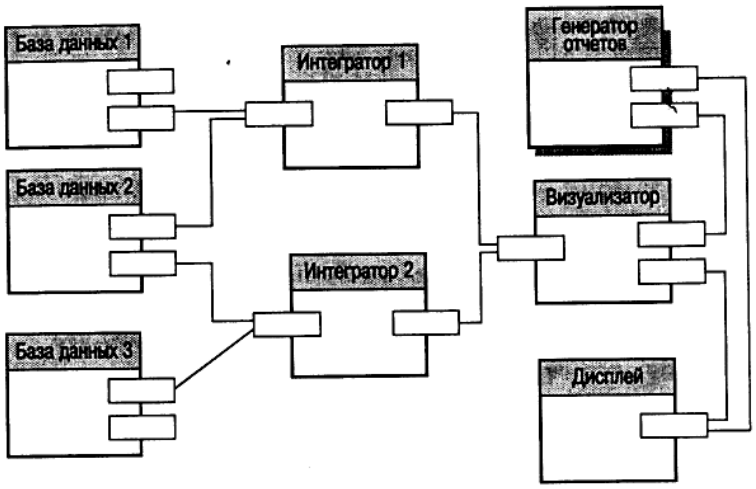
В процесі проектування систем архітектуру розподілених об'єктів можна використовувати двояко.

1. У вигляді логічної моделі, яка дозволяє розробникам структурувати і спланувати систему. В цьому випадку функціональність додатку описується тільки в термінах і комбінаціях сервісів. Потім розробляються способи надання сервісів за допомогою декількох розподілених об'єктів. На цьому рівні, як правило, проектують крупномодульные об'єкти, які надають сервіси, що відображають специфіку конкретної області додатку. Наприклад, в програму обліку роздрібної торгівлі можна включити об'єкти, які б вели облік стану запасів, відстежували взаємодію з клієнтами, класифікували товари і ін.

2. Як гнучкий підхід до реалізації систем клієнт/сервер. В цьому випадку логічна модель системи – це модель клієнт/сервер, в якій клієнти і сервери реалізовані як розподілені об'єкти, що взаємодіють за допомогою програмної шини. При такому підході легко замінити систему, наприклад дворівневу на багаторівневу. В цьому випадку ні сервер, ні клієнт не можуть бути реалізовані в одному об'єкті, проте можуть складатися з безлічі невеликих об'єктів, кожен з яких надає певний сервіс.

Прикладом системи, якою підходить архітектура розподілених об'єктів, може служити система обробки даних, що зберігаються в різних базах даних (мал. 11.10). В даному прикладі будь-яку базу даних можна представити як об'єкт з інтерфейсом, що надає доступ до даним "тільки читання". Кожен з об'єктів-інтеграторів займається певними типами залежностей між даними, збираючи інформацію з баз даних, щоб спробувати прослідкувати ці залежності.

Об'екти-візуалізатори взаємодіють з об'єктами-інтеграторами для представлення даних в графічному вигляді або для складання звітів за аналізованими даними. Способи представлення графічної інформації розглядаються в розділі 15.




*Мал. 11.10. Архітектура розподіленої системи обробки даних*

Для такого типу додатків архітектура розподілених об'єктів підходить більше, ніж архітектура клієнт/сервер, по трьом причинам.

1. У цих системах (у відмінність, наприклад, від системи банкоматів) немає одного постачальника сервісу, на якому були б зосереджені всі сервіси управління даними.
2. Можна збільшувати кількість доступних баз даних, не перериваючи роботу системи, оскільки кожна база даних є просто об'єктом. Ці об'єкти підтримують спрощений інтерфейс, який управляє доступом до даним. Доступні бази даних можна розмістити на різних машинах.
3. За допомогою додавання нових об'єктів-інтеграторів можна відстежувати нові типи залежностей даними.

Головним недоліком архітектури розподілених об'єктів є те, що їх складніше проектувати, чим системи клієнт/сервер. Виявляється, що системи клієнт/сервер надають природніший підхід до створення розподілених систем. У нім відбиваються взаємини між людьми, при яких одні люди користуються послугами інших людей, що спеціалізуються на наданні конкретних послуг. Набагато важче розробити систему відповідно до архітектури розподілених об'єктів, оскільки індустрія створення ПО поки що не накопичила достатнього досвіду в проектуванні і розробці крупномодульных об'єктів.

## 11.4. CORBA

Як вже наголошувалося в попередньому розділі, при реалізації архітектури розподілених об'єктів необхідне проміжне програмне забезпечення (брокери запитів до об'єктів), організуюче взаємодію між розподіленими об'єктами. Тут можуть виникнути певні проблеми, оскільки об'єкти в системі можуть бути реалізовані на різних мовах програмування, можуть запускатися на різних платформах і їх імена не повинні бути відомі всім іншим об'єктам системи. Тому проміжне ПО повинно виконувати велику роботу для того, щоб підтримувалася постійна взаємодія об'єктів.

Зараз для підтримки розподілених об'єктних обчислень існує два основні стандарти проміжного ПО.

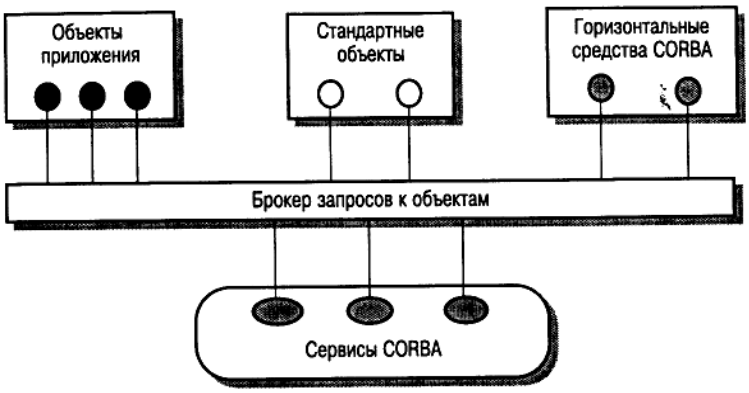
1. CORBA (Common Object Request Broker Architecture– архітектура брокерів запитів до загальних об'єктів). Це набір стандартів для проміжного ПО, розроблений групою OMG (Object Management Group – група по управлінню об'єктами). OMG є консорціумом фірм-виробників програмного і апаратного забезпечення, в числі яких такі компанії, як Sun, Hewlett-Packard і IBM. Стандарти CORBA визначають загальний машинонезалежний підхід до розподілених об'єктних обчислень. Різними виробниками розроблена безліч реалізацій цього стандарту. Стандарти CORBA підтримуються операційною системою Unix і операційними системами від Microsoft.
2. DCOM (Distributed Component Object Model – об'єктна модель розподілених компонентів). DCOM є стандартом, розробленим і реалізованим компанією Microsoft і інтегрований в її операційні системи. Дана модель розподілених обчислень менш універсальна, чим CORBA і пропонує більш обмежені можливості мережевих взаємодій. Зараз використання DCOM обмежується операційними системами Microsoft.

Тут я вирішив приділити увагу технології CORBA, оскільки вона більш універсальна. Крім того, я вважаю, що, ймовірно, CORBA, DCOM і інші технології, наприклад RMI (Remote Method Invocation – виклик видаленого методу, технологія побудови розподілених застосувань на мові Java), поступово зближуватимуться один з одним і це зближення базуватиметься на стандартах CORBA. Тому немає необхідності в ще одному стандарті. Різні стандарти стоятимуть тільки на заваді в подальшому розвитку.

Стандарти CORBA визначені групою OMG, яка об'єднує більше 500 компаній, що підтримують об'єктно-орієнтовані розробки. Роль OMG – створення стандартів для об'єктно-орієнтованих розробок, а не забезпечення конкретних реалізацій цих стандартів. Ці стандарти знаходяться у вільному доступі на Web-узле OMG. Група займається не тільки стандартами CORBA, але також визначає широкий діапазон інших стандартів, включаючи мову моделювання UML.

Представлення розподілених застосувань в рамках CORBA показане на мал. 11.11. Це спрощена схема архітектури управління об'єктами, узята із статті [317]. Передбачається, що розподілене застосування повинне складатися з перерахованих нижче компонентів.

1. Об'єкти додатки, які створені і розроблені для даного програмного продукту.
2. Стандартні об'єкти, які визначені групою OMG для специфічних завдань. Під час написання книги безліч фахівців займалися розробкою стандартів об'єктів в області фінансування, страхування, електронної комерції, охорона здоров'я і багато інших.
3. Основні сервіси CORBA, що підтримують базові сервіси розподілених обчислень, наприклад каталоги, управління захистом і ін.
4. Горизонтальні засоби CORBA, наприклад призначені для користувача інтерфейси, засоби управління системою і тому подібне Під горизонтальними маються на увазі засоби, загальні для багатьох застосувань.




*Мал. 11.11. Структура розподіленого застосування, заснованого на стандартах CORBA*

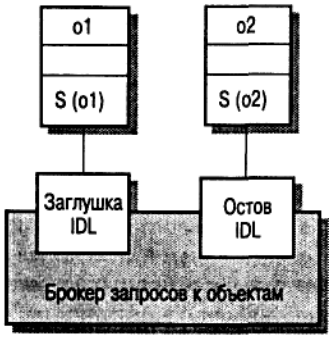
Стандарти CORBA описують чотири основні елементи.

1. Модель об'єктів, в якій об'єкт CORBA інкапсулює стани за допомогою чіткого опису на мові IDL (Interface Definition Language – мова опису інтерфейсів).
2. Брокер запитів до об'єктів (Object Request Broker– ORB), який управляє запитами до сервісів об'єктів. ORB розміщує об'єкти, що надають сервіси, готує їх до отримання запитів, передає запит до сервісу і повертає результати об'єкту, що зробив запит.
3. Сукупність сервісів об'єктів, які є основними сервісами, і необхідні в багатьох розподілених застосуваннях. Прикладами можуть бути служби каталогів, сервіси транзакцій і сервіси підтримки тимчасових об'єктів.
4. Сукупність загальних компонентів, побудованих на верхньому рівні основних сервісів. Вони можуть бути як вертикальними, такими, що відображають специфіку конкретної області, так і горизонтальними універсальними компонентами, використовуваними в багатьох програмних застосуваннях. Ці компоненти розглядаються в розділі 14.

У моделі CORBA об'єкт інкапсулює атрибути і сервіси як звичайний об'єкт. Разом з тим в об'єктах CORBA ще повинне міститися визначення різних інтерфейсів, що описують глобальні атрибути і операції об'єкту. Інтерфейси об'єктів CORBA визначаються на стандартній універсальній мові опису інтерфейсів IDL. Якщо один об'єкт запрошує сервіси, що надаються іншими об'єктами, він дістає доступ до цих сервісів через IDL-интерфейс. Об'єкти CORBA мають унікальний ідентифікатор, званий IOR (Interoperable Object Reference – посилання на взаємодіючий об'єкт). Коли один об'єкт відправляє запити до сервісу, що надається іншим об'єктом, використовується ідентифікатор IOR.

Брокерові запитів до об'єктів відомі об'єкти, що запрошують сервіси і їх інтерфейси. Він організовує взаємодію між об'єктами. Взаємодіючим об'єктам не потрібне що-небудь знати про розміщення інших об'єктів, а також про їх реалізацію. Оскільки інтерфейс IDL відокремлює об'єкти від брокера, реалізацію об'єктів можна змінювати, не зачіпаючи інші компоненти системи.

На мал. 11.12 показано, як об'єкти ol і о2 взаємодіють за допомогою брокера запитів до об'єктів. Зухвалий об'єкт (ol) пов'язаний із заглушкою (stub) IDL, яка визначає інтерфейс об'єкту, що надає сервіс. Конструктор об'єкту ol при запиті до сервісу упроваджує виклики в заглушку своєї реалізації об'єкту. Мова IDL є розширенням C++, тому, якщо ви програмуєте на мовах C++, З або Java, дістати доступ до заглушки зовсім просто. Переклад опису інтерфейсу об'єкту на IDL також можливий і для інших мов, наприклад Ada або COBOL. Але в цих випадках необхідна відповідна інструментальна підтримка.



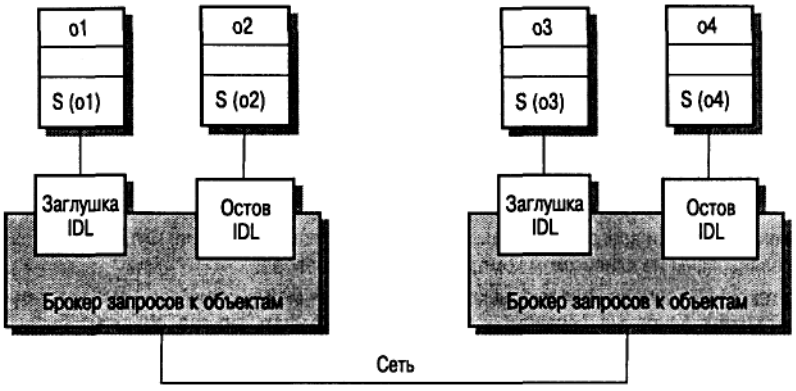

*Мал. 11.12. Взаємодія об'єктів за допомогою брокера запитів до об'єктів*

Об'єкт, що надає сервіс, пов'язаний з остовом (skeleton) IDL, який пов'язує інтерфейс з реалізацією сервісів. Іншими словами, коли сервіс викликається через інтерфейс, остов IDL транслює виклик до сервісу незалежно від того, яка мова використовувалася в реалізації. Після завершення методу або процедури остов транслює результати в мову IDL, так що вони стають доступними зухвалому об'єкту. Якщо об'єкт одночасно надає сервіси іншим об'єктам або використовує сервіси, які надані ще десь, йому потрібні і остов IDL, і заглушка IDL. Остання необхідна всім використовуваним об'єктам.

Брокер запитів до об'єктів зазвичай реалізується не у вигляді окремих процесів, а як каркас (див. розділ 14), який пов'язаний з реалізацією об'єктів. Тому в розподіленій системі кожен комп'ютер, на якому працюють об'єкти, повинен мати власний брокер запитів до об'єктів, який оброблятиме всі локальні виклики об'єктів. Але якщо запит зроблений до сервісу, який наданий видаленим об'єктом, потрібна взаємодія між брокерами.

Така ситуація проілюстрована на мал. 11.13. У даному прикладі, якщо об'єкт ol або о2 відправляє запити до сервісів, що надаються об'єктами о3 або о4, то необхідна взаємодія пов'язаних з цими об'єктами брокерів. Стандарти CORBA підтримують взаємодію "брокер-брокер", яке забезпечує брокерам доступ до описів інтерфейсів IDL, і пропонують розроблений групою OMG стандарт узагальненого протоколу взаємодії брокерів GIOP (Generic INTER-ORB Protocol). Даний протокол визначає стандартні повідомлення, якими можуть обмінюватися брокери при виконанні викликів видаленого об'єкту і передачі інформації. У поєднанні з протоколом Internet низького рівня TCP/IP цей протокол дозволяє брокерам взаємодіяти через Internet.

Перші варіанти CORBA були розроблені ще в 1980-х роках. Ранні версії CORBA просто були пов'язані з підтримкою розподілених об'єктів. Проте з часом стандарти розвивалися, ставали більш розширеними. Подібно до механізмів взаємодії розподілених об'єктів, стандарти CORBA зараз визначають деякі стандартні сервіси, які можна використовувати для підтримки об'єктно-орієнтованих застосувань.




*Мал. 11.13. Взаємодія між брокерами запитів до об'єктів*

Сервіси CORBA є засобами, які необхідні в багатьох розподілених системах. Ці стандарти визначають приблизно 15 загальних служб (сервісів). Ось деякі з них.

1. Служба імен, яка дозволяє об'єктам знаходити інші об'єкти в мережі і посилатися на них. Служба імен є сервісом каталогів, який привласнює імена об'єктам. При необхідності об'єкти через цю службу можуть знаходити ідентифікатори IOR інших об'єктів.
2. Служба реєстрації, яка дозволяє об'єктам реєструвати інші об'єкти після здійснення деяких подій. За допомогою цієї служби об'єкти можна реєструвати по їх участі в певній події, а коли дана подія вже відбулася, воно автоматично реєструється сервісом.
3. Служба транзакцій, яка підтримує елементарні транзакції і відкіт назад у разі помилок або збоїв. Ця служба є відмовостійким засобом (див. розділ 18), що забезпечує відновлення у разі помилок під час операції оновлення. Якщо дії з оновлення об'єкту приведуть до помилок або збою системи, даний об'єкт завжди можна повернути назад до того стану, який був перед початком оновлення.

Вважається, що стандарти CORBA повинні містити визначення інтерфейсів для широкого діапазону компонентів, які можуть використовуватися при побудові розподілених застосувань. Ці компоненти можуть бути вертикальними або горизонтальними. Вертикальні компоненти розробляються спеціально для конкретних застосувань. Як вже наголошувалося, розробкою визначень цих компонентів зайнята безліч фахівців з різних сфер діяльності. Горизонтальні компоненти універсальні, наприклад компоненти призначеного для користувача інтерфейсу.

Під час написання цієї книги специфікації компонентів були вже розроблені, але ще не узгоджені. На мій погляд, ймовірно, саме тут найбільш слабке місце стандартів CORBA, і, можливо, буде потрібно декілька років, щоб досягти того, що в наявності будуть і специфікації, і реалізації компонентів.

## КЛЮЧОВІ ПОНЯТТЯ

- Всі великі системи в тому або іншому ступені є розподіленими, в яких програмні компоненти виконуються на інтегрованій в мережу групі процесорів.
- Розподіленим системам властиві наступні риси: використання ресурсів, відвертість, паралельність, масштабованість, стійкість до помилок і прозорість.
- Системи клієнт/сервер є розподіленими. Такі системи моделюються як набір сервісів, що надаються сервером клієнтським процесам.
- У системі клієнт/сервер інтерфейс користувача на стороні клієнта, а управління даними завжди підтримується на сервері, що розділяється. Функції додатку можуть бути реалізовані на клієнтському комп'ютері або на сервері.
- У архітектурі розподілених об'єктів немає відмінностей між клієнтами і серверами. Об'єкти надають основні сервіси, які можуть викликати інші об'єкти. Такий же підхід можна використовувати в реалізації систем клієнт/сервер.
- У системах розподілених об'єктів повинні бути проміжне програмне забезпечення, призначене для обробки взаємодій між об'єктами, а також додавання або видалення об'єктів з системи. Концептуально проміжне ПО можна представити як програмну шину, до якої підключені об'єкти.
- Стандарти CORBA є набором стандартів для проміжного ПО, такого, що підтримує архітектуру розподілених об'єктів. До них відносяться визначення моделі об'єктів, брокера запитів до об'єктів і загальних сервісів. В даний час існує декілька реалізацій стандартів CORBA.

### Вправи

**11.1.** Поясните, чому розподілені системи завжди більш масштабовані, ніж централізовані. Яка вірогідна межа масштабованості програмних систем?

**11.2.** У чому основна відмінність між моделями товстого і тонкого клієнта в розробці систем клієнт/сервер? Поясните, чому використання Java як мови реалізації згладжує відмінності між цими моделями?

**11.3.** На основі моделі додатку, зображеної на мал. 11.4, розглянете можливі проблеми, які можуть виникнути при перетворенні системи 1980-х років, реалізованої на мейнфрей-

ме і призначеної для роботи у сфері охорони здоров'я, в систему архітектури клієнт/сервер.

**11.4.** Розподілені системи, що базуються на моделі клієнт/сервер, розроблялися з 1980-х років, але тільки недавно такі системи, засновані на розподілених об'єктах, були реалізовані. Приведіть три причини, чому так вийшло.

**11.5.** Поясните, чому використання розподілених об'єктів сумісне з брокером запитів до об'єктів спрощує реалізацію масштабованих систем клієнт/сервер. Проілюструйте свою відповідь прикладом.

**11.6.** Яким чином використовується мова IDL для підтримки взаємодії між об'єктами, реалізованими на різних мовах програмування? Поясните, чому такий підхід може викликати проблеми, пов'язані з продуктивністю, якщо між мовами, які використовуються при реалізації об'єктів, є радикальні відмінності.

**11.7.** Які базові засоби повинен надавати брокер запитів до об'єктів?

**11.8.** Можна показати, що розробка стандартів CORBA для горизонтальних і вертикальних компонентів обмежує конкуренцію. Якщо вони вже створені і адаптовані, це перешкоджає розробці кращих компонентів дрібнішими компаніями. Обговорите роль стандартизації в підтримці або обмеженні конкуренції на ринку програмного забезпечення.

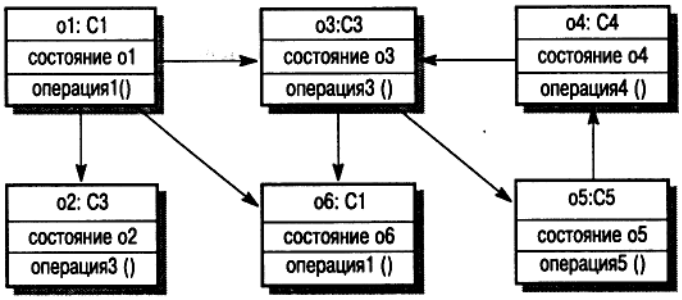
# 12. Об'єктно-орієнтоване проектування

## Цілі

Мета справжнього розділу – познайомити з підходом до проектування програмного забезпечення, в якому система представляється у вигляді взаємодіючих об'єктів. Прочитавши цей розділ, ви винні:

- ❑ знати, що структуру програми можна представити у вигляді сукупності взаємодіючих об'єктів, керівників власним станом і операціями;
- ❑ мати уявлення про основні етапи процесу об'єктно-орієнтованого проектування;
- ❑ розуміти різні моделі, які використовуються при документуванні об'єктно-орієнтованої структури;
- ❑ познайомитися з представленням цих моделей за допомогою UML.

Об'єктно-орієнтоване проектування є стратегією, в рамках якої розробники системи замість операцій і функцій мислять в поняттях *об'єкти.* Програмна система складається з взаємодіючих об'єктів, які мають власний локальний стан і можуть виконувати певний набір операцій, визначуваний станом об'єкту (мал. 12.1). Об'єкти приховують інформацію про представлення станів і, отже, обмежують до них доступ. Під процесом об'єктно-орієнтованого проектування мається на увазі проектування класів об'єктів і взаємин між цими класами. Коли проект реалізований у вигляді виконуваної програми, всі необхідні об'єкти створюються динамічно за допомогою визначень




*Мал. 12.1. Система взаємодіючих об'єктів*

Об'єктно-орієнтоване проектування – тільки частина *об'єктно-орієнтованого процесу розробки системи,* де впродовж всього процесу створення ПО використовується об'єктно-орієнтований підхід. Цей підхід має на увазі виконання трьох етапів.

- *Об'єктно-орієнтований аналіз.* Створення об'єктно-орієнтованої моделі наочної області додатку ПО. Тут об'єкти відображають реальні об'єкти-суть, також визначаються операції, що виконуються об'єктами.
- *Об'єктно-орієнтоване проектування.* Розробка об'єктно-орієнтованої моделі системи ПО (системної архітектури) з урахуванням системних вимог. У об'єктно-орієнтованій моделі визначення всіх об'єктів підпорядковане рішенню конкретної задачі.
- *Об'єктно-орієнтоване програмування.* Реалізація архітектури (моделі) системи за допомогою об'єктно-орієнтованої мови програмування. Такі мови, наприклад Java, безпосередньо виконують реалізацію певних об'єктів і надають засоби для визначення класів об'єктів.

Дані етапи можуть "перетікати" один в одного, тобто можуть не мати чітких рамок, причому на кожному етапі зазвичай застосовується одна і та ж система нотації. Перехід на наступний етап приводить до удосконалення результатів попереднього етапу шляхом детальнішого опису визначених раніше класів об'єктів і визначення нових класів. Оскільки дані

приховані усередині об'єктів, детальні рішення про представлення даних можна відкласти до етапу реалізації системи. В деяких випадках можна також не поспішати з ухваленням рішень про розташування об'єктів і про те, чи будуть ці об'єкти послідовними або паралельними. Все сказане означає, що розробники ПО не обмежені деталями реалізації системи.

Об'єктно-орієнтовані системи можна розглядати як сукупність автономних і до певної міри незалежних об'єктів. Зміна реалізації якого-небудь об'єкту або додавання нових функцій не впливає на інші об'єкти системи. Часто існує чітка відповідність між реальними об'єктами (наприклад, апаратними засобами) і керівниками ними об'єктами програмної системи. Такий підхід полегшує розуміння і реалізацію проекту.

Потенційно всі об'єкти є повторно використовуваними компонентами, оскільки вони незалежно інкапсулюють дані про стан і операцію. Архітектуру ПО можна розробляти на базі об'єктів, вже створених в попередніх проектах. Такий підхід знижує вартість проектування, програмування і тестування ПО. Крім того, з'являється можливість використовувати стандартні об'єкти, що зменшує ризик, пов'язаний з розробкою програмного забезпечення. Проте, як показано в розділі 14, іноді повторне використання найефективніше реалізувати за допомогою колекцій об'єктів (компонентів або об'єктних структур), а не через окремі об'єкти.

У книгах [74, 295, 186, 54, 137, 13*, 32*, 34*] пропонуються різні методи об'єктно-орієнтованого проектування. У цих методах впродовж всього процесу проектування використовується одноманітна нотація, прийнята в UML [304]. У даному розділі не пропонуються які-небудь особливі методи проектування, а розглядаються лише загальні концепції об'єктно-орієнтованого проектування. У розділі 12.2 розглянуті етапи процесу проектування. По всьому розділу використовується система позначень, прийнята в UML.

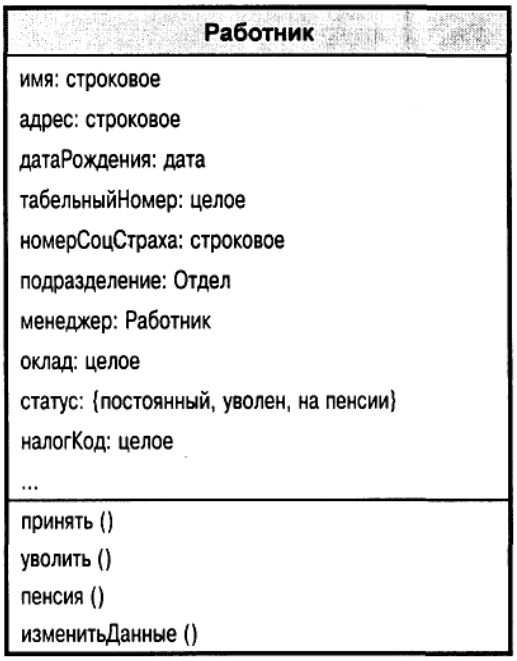
## 12.1. Об'єкти і класи об'єктів

В даний час широко використовуються поняття *об'єкт* і *об'єктно-орієнтований.* Ці терміни застосовуються до різних типів об'єктів, методів проектування, систем і мов програмування. У всіх випадках застосовується загальне правило, згідно якому об'єкт інкапсулює дані про свою внутрішню будову. Це правило відбите в моєму визначенні об'єкту і класу об'єктів.

> Об'єкт–ето щось, здатне перебувати в різних станах і певна безліч операцій, що має. Стан визначається як набір атрибутів об'єкту. Операції, пов'язані з об'єктом, надають сервіси (функціональні можливості) іншим об'єктам (клієнтам) для виконання певних обчислень. Об'єкти створюються відповідно до визначення класу об'єктів, яке служить шаблоном для створення об'єктів. У нього включені оголошення всіх атрибутів і операцій, пов'язаних з об'єктом даного класу.

Нотація, яка використовується тут для позначення класів об'єктів, визначена в UML. Клас об'єктів представляється як прямокутник з назвою класу, розділений на дві секції. У верхній секції перераховані атрибути об'єктів. Операції, пов'язані з даним об'єктом, розташовані в нижній секції. Приклад такої нотації представлений на мал. 12.2, де показаний клас об'єктів, що моделює службовця якоїсь організації. У UML термін *операція* є специфікацією деякої дії, а термін *метод* зазвичай відноситься до реалізації даної операції.

Клас **Працівник** визначається рядом атрибутів, в яких містяться дані про службовців, зокрема їх імена і адресу, коди соціального забезпечення, податкові коди і так далі Звичайно, насправді атрибутів, що асоціюються з класом, більше, ніж зображено на малюнку. Визначені також операції, пов'язані з об'єктами: **прийняти** (виконується під час вступу на роботу), **звільнити** (виконується при звільненні службовця з організації), **пенсія** (виконується, якщо службовець стає пенсіонером організації) і **изменитьДанные** (виконується у випадках, якщо потрібно внести зміни до наявних даних про працівника).




*Мал. 12.2. Об'єкт Працівник*

Взаємодія між об'єктами здійснюється за допомогою запитів до сервісів (виклик методів) з інших об'єктів і при необхідності шляхом обміну даними, потрібними для підтримки сервісу. Копії даних, необхідних для роботи сервісу, і результати роботи сервісу передаються як параметри. Ось декілька прикладів такого стилю взаємодії.

```
// Виклик методу, що асоціюється з об'єктом Buffer (Буфер)
// який повертає наступне значення в буфер
v = circularBuffer.Get () ;
// Виклик методу, пов'язаного з об'єктом thermostat (термостат)
// який підтримує потрібну температуру
thermostat.setTemp (20);
```

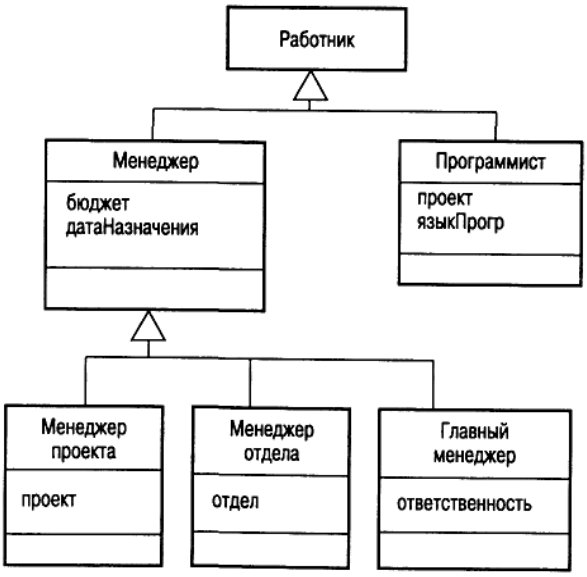
У деяких розподілених системах взаємодія між об'єктами реалізована безпосередньо у вигляді текстових повідомлень, якими обмінюються об'єкти. Об'єкт, що отримав повідомлення, виконує його граматичний розбір, ідентифікує сервіс і пов'язані з ним дані і запускає запрошуваний сервіс. Проте, якщо об'єкти співіснують в одній програмі, виклики методів реалізовані аналогічно викликам процедур або функцій в мовах програмування, наприклад таких, як З або Ada.

Якщо запити до сервісу реалізовані саме таким чином, взаємодія між об'єктами синхронно. Це означає, що об'єкт, що відправив запит до сервісу, чекає закінчення виконання запиту. Проте, якщо об'єкти реалізовані як паралельні процеси або потоки, взаємодія об'єктів може бути асинхронною. Відправивши запит до сервісу, об'єкт може продовжити роботу і не чекати, поки сервіс виконає його запит. Нижче в цьому розділі показано, яким чином можна реалізувати об'єкти як паралельні процеси.

Як наголошувалося в розділі 7, в якій описаний ряд об'єктних моделей, класи об'єктів можна упорядкувати або у вигляді ієрархії узагальнення або у вигляді ієрархії спадкоємство, яке показує відношення між основними і приватними класами об'єктів. Ці приватні класи об'єктів повністю сумісні з основними класами, але містять більше інформації. У системі позначень UML напрям узагальнення указується стрілками, направленими на батьківський клас. У об'єктно-орієнтованих мовах програмування узагальнення зазвичай реалізується через механізм спадкоємства. Похідний клас (клас-нащадок) успадковує атрибути і операції від батьківського класу.

Приклад такої ієрархії зображений на мал. 12.3, де показані різні класи працівників. Класи, розташовані внизу ієрархії, мають ті ж атрибути і операції, що і батьківські класи, але

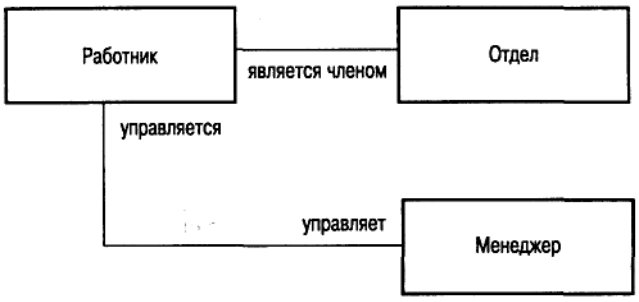
можуть містити нові атрибути і операції або ж змінювати наявні в батьківських класах. Якщо в моделі використовується ім'я батьківського класу, значить, об'єкт в системі може бути визначений або самим класом, або будь-яким з його нащадків.




*Мал. 12.3. Ієрархія узагальнення*

На мал. 12.3 видно, що клас **Менеджер** володіє всіма атрибутами і операціями класу **Працівник** і, крім того, має два нові атрибути: ресурси, якими управляє менеджер (**бюджет**), і дата призначення його на посаду менеджера (**датаНазначения**). Також додані нові атрибути в клас **Програміст**. Один з них визначає проект, над яким працює програміст, інший характеризує рівень його професіоналізму при використанні певної мови програмування (**языкПрогр**). Таким чином, об'єкти класу **Менеджер** і **Програміст** можна використовувати замість об'єктів класу **Працівник**.

Об'єкти, що є членами класу об'єктів, взаємодіють з іншими об'єктами. Ці взаємини моделюються за допомогою опису зв'язків (асоціацій) між класами об'єктів. У UML зв'язок позначається лінією, яка сполучає класи об'єктів, причому лінія може бути забезпечена інформацією про даний зв'язок. На мал. 12.4 показані зв'язки між об'єктами класів **Працівник** і **Відділ** і між об'єктами класів **Працівник** і **Менеджер**.




*Мал. 12.4. Модель зв'язків*

Зв'язки представляють найзагальніші відносини і часто використовуються в UML там, де потрібно вказати, що якась властивість об'єкту є пов'язаною з об'єктом або ж реалізація методу об'єкту покладається на зв'язаний об'єкт. Проте в принципі тип зв'язку може бути яким завгодно. Одним з найбільш поширених типів зв'язку, який служить для створення нових об'єктів з що вже є, є агрегація. Цей тип зв'язку розглянутий в розділі 7.

### 12.1.1. Паралельні об'єкти

У загальному випадку об'єкти запрошують сервіс від будь-якого об'єкту за допомогою передачі йому повідомлення "запит до сервісу". Зазвичай немає необхідності в послідовному виконанні, при якому один об'єкт чекає завершення роботи сервісу по зробленому запиту. Загальна модель взаємодії об'єктів дозволяє їх одночасне виконання у вигляді паралельних процесів. Такі об'єкти можуть виконуватися на одному комп'ютері або на різних машинах як розподілені об'єкти.

На практиці в більшості об'єктно-орієнтованих мов програмування за умовчанням реалізована модель послідовного виконання, в якій запити до сервісів об'єктів і виклики функцій реалізовані одним і тим же способом. Наприклад, на мові Java, коли об'єкт, що викликав об'єкт **theList** (Список), створюється із звичайного класу об'єктів, це запишеться так:

```
theList.append(17)
```

Тут викликається метод **append** (додати), пов'язаний з об'єктом **theList**, який додає елемент 17 в список **theList**, а виконання об'єкту, що зробив виклик, припиняється до тих пір, поки не завершиться операція додавання. Проте в Java існує дуже простий механізм потоків (threads), який дозволяє створювати об'єкти, що паралельно виконуються. Тому об'єктно-орієнтовану архітектуру програмної системи можна перетворити так, щоб об'єкти стали паралельними процесами.

Існує два типи паралельних об'єктів.

1. *Сервери,* в яких об'єкт реалізований як паралельний процес з методами, відповідними певним операціям об'єкту. Методи запускаються у відповідь на зовнішнє повідомлення і можуть виконуватися паралельно з методами, пов'язаними з іншими об'єктами. Після закінчення всіх дій виконання об'єкту припиняється і він чекає подальших запитів до сервісу.
2. *Активні об'єкти,* у яких стан може змінюватися за допомогою операцій, що виконуються усередині самого об'єкту. Процес, що представляє об'єкт, постійно виконує ці операції, а отже, ніколи не зупиняється.

Сервери найбільш корисні в розподілених середовищах, де зухвалі об'єкти, що викликаються, виконуються на різних комп'ютерах. Час відповіді, який потрібний сервісу, заздалегідь не відомий, тому де тільки можна слід спроектувати систему так, щоб об'єкт, що відправив запит до сервісу, не чекав, поки сервіс виконає запит. Також сервери можуть використовуватися на одній машині, де їм вимагається якийсь час для виконання запиту (наприклад, друк документа) і де є вірогідність відправки запитів до сервісу від декількох різних об'єктів.

Активні об'єкти використовуються там, де об'єктам необхідно оновлювати свій стан через певні інтервали часу. Такі об'єкти характерні для систем реального часу, в яких об'єкти пов'язані з апаратними пристроями, що збирають інформацію з оточення середовища. Методи об'єктів дозволяють іншим об'єктам дістати доступ до інформації, що визначає стан об'єкту.

У лістингу 12.1 показано, як на мові Java можна визначити і реалізувати активний об'єкт. Даний клас об'єктів представляє бортовий радіомаяк-відповідач (transponder) літака. За допомогою супутникової навігаційної системи радіомаяк-відповідач відстежує положення літака. Він може відповідати на повідомлення, що приходять від комп'ютерів, керівників повітряними польотами. У відповідь на запит метод **givePosition** повідомляє поточне положення літака.

**Лістинг 12.1. Реалізація активного об'єкту, що використовує потоки мови Java**

```
class Transponder extends Thread {
       Position currentPosition ;
       Coords cl, c2 ;
       Satellite satl, sat2 ;
       Navigator theNavigator ;
```

```
        public Position givePosition ()
        {
            return currentPosition;
        }
        public void run ()
        {
            while (true)
            {
                c1 = sat1.position ();
                c2 = sat2.position ();
                currentPosition = theNavigator.compute (c1, c2) ;
            }
        }
}//Transponder
```

Даний об'єкт реалізований як потік, де в безперервному циклі методу **run** міститься код, що обчислює положення літака за допомогою сигналів, отриманих від супутників. У Java потоки створюються за допомогою вбудованого класу **Thread** (Потік), промовця в оголошенні класів як базовий.

## 12.2. Процес об'єктно-орієнтованого проектування

У цьому розділі процес об'єктно-орієнтованого проектування показаний на прикладі розробки структури програмної системи, що управляє, вбудованої в автоматизовану метеостанцію. Як наголошувалося вище, є декілька методів об'єктно-орієнтованого проектування, причому якого-небудь переважного методу або процесу проектування не існує. Процес, що розглядається тут, є достатньо загальним, тобто складається з операцій, характерних для більшості процесів об'єктно-орієнтованого проектування. В цьому відношенні він порівнянний з процесом, пропонованою мовою UML [304], проте я значно спростив його.

Загальний процес об'єктно-орієнтованого проектування складається з декількох етапів.

1. Визначення робочого оточення системи і розробка моделей її використання.
2. Проектування архітектури системи.
3. Визначення основних об'єктів системи.
4. Розробка моделей архітектури системи.
5. Визначення інтерфейсів об'єкту.

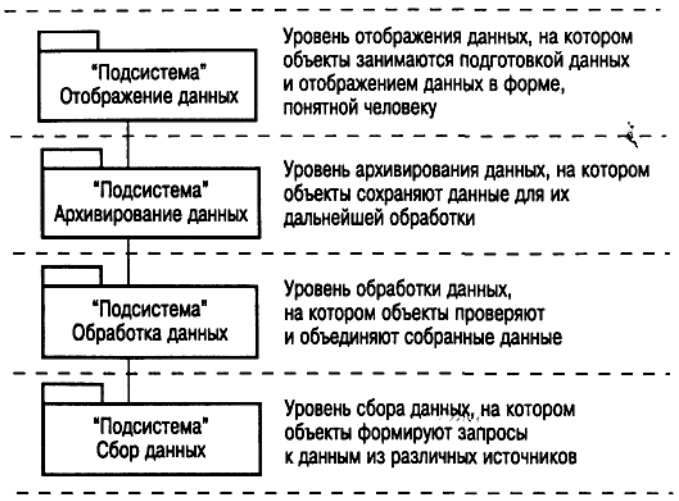
Процес проектування не можна представити у вигляді простої схеми, в якій передбачається чітка послідовність етапів. Фактично всі перераховані етапи значною мірою можна виконувати паралельно, з взаємним впливом один на одного. Як тільки розроблена архітектура системи, визначаються об'єкти і (частково або повністю) інтерфейси. Після створення моделей об'єктів окремі об'єкти можна перевизначити, а це може привести до змін в архітектурі системи. Далі в цьому розділі кожен етап процесу проектування обговорюється окремо.

Приклад ПО, яким я скористаюся для ілюстрації об'єктно-орієнтованого проектування, є частина системи, що створює метеорологічні карти на основі автоматично зібраних метеорологічних даних. Докладне перерахування вимог для такої системи займе багато сторінок. Проте, навіть обмежившись коротким описом системи, можна розробити її загальну архітектуру.

> Однією з вимог системи побудови карти погоди є регулярне оновлення метеорологічних карт на основі даних, отриманих від видалених метеостанцій і інших джерел, наприклад спостерігачів, метеозондов і супутників. У відповідь на запит регіонального комп'ютера системи обслуговування метеостанцій передають йому свої дані.
> Регіональна комп'ютерна система об'єднує дані з різних джерел. Зібрані дані архівуються і за допомогою даних з цього архіву і бази даних цифрових карт створюється на-

бір локальних метеорологічних карт. Карти можна роздрукувати, направивши їх на спеціальний принтер, або ж відобразити в різних форматах.

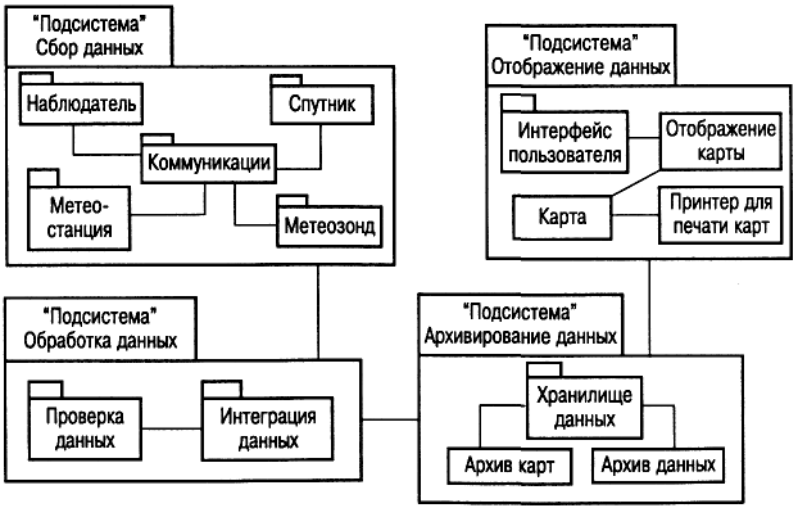
З даного опису видно, що одна частина загальної системи займається збором даних, інша узагальнює дані, отримані з різних джерел, третя виконує архівацію даних і нарешті четверта створює метеорологічні карти. На мал. 12.5 зображена одна з можливої архітектури системи, яку можна побудувати на основі запропонованого опису. Вона є багаторівневою архітектурою (обговорювану в розділі 10), в якій відбиті всі етапи обробки даних в системі, тобто збір даних, узагальнення даних, архівація даних і створення карт. Така багаторівнева архітектура цілком годиться для нашої системи, оскільки кожен етап грунтується тільки на обробці даних, виконаній на попередньому етапі.




*Мал. 12.5. Багаторівнева архітектура системи побудови карт погоди*

На мал. 12.5 показані всі рівні системи. Назви рівнів поміщені в прямокутники, що в нотації UML позначає підсистеми. Прямокутники UML (тобто підсистеми) – це набір об'єктів і інших підсистем. Я використовую тут це позначення, щоб показати, що кожен рівень включає безліч інших компонентів.

На мал. 12.6 зображена розширена модель архітектури, в якій показані компоненти підсистем. Ці компоненти також дуже абстрактні і побудовані на інформації, що міститься в описі системи. Продовжимо розглядати цей приклад, приділяючи особливу увагу підсистемі **Метеостанція**, яка є частиною рівня **Збір даних**.




*Мал. 12.6. Підсистеми в системі побудови карт погоди*

### 12.2.1. Оточення системи і моделі її використання

Перший етап в будь-якому процесі проектування полягає у виявленні взаємин між проектованим програмним забезпеченням і його оточенням. Виявлення цих взаємин допомагає вирішити, як забезпечити необхідну функціональність системи і як структурувати систему, щоб вона могла ефективно взаємодіяти зі своїм оточенням.

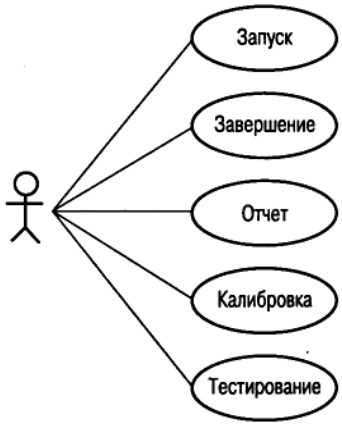
Моделлю оточення системи і моделлю використання системи є два доповнюючий один один моделі взаємин між даною системою і її оточенням.

1. Модель оточення системи – це статична модель, яка описує інші системи з оточення того, що розробляється ПО.
2. Модель використання системи – динамічна модель, яка показує взаємодію даної системи зі своїм оточенням.

Модель оточення системи можна представити за допомогою схеми зв'язків (див. мал. 12.4), яка дає просту блок-схему загальної архітектури системи. За допомогою *пакетів* мови UML її можна представити в розгорненому вигляді як сукупність підсистем (див. мал. 12.6). Таке уявлення показує, що робоче оточення системи Метеостанція знаходиться усередині підсистеми, що займається збором даних. Там же показані інші підсистеми, які утворюють систему побудови карт погоди.

При моделюванні взаємодії системи з її оточенням застосовується абстрактний підхід, який не вимагає великих об'ємів даних для опису цих взаємодій. Підхід, пропонований UML, полягає в тому, щоб розробити модель варіантів використання, в якій кожним варіантом є певна взаємодія з системою (див. розділ 6). У моделі варіантів використання кожна можлива взаємодія зображається у вигляді еліпса, а зовнішня суть, включена у взаємодію, представлена стилізованою фігуркою людини. У нашому прикладі зовнішня суть, хоча і представлена фігуркою людини, є системою обробки метеорологічних даних.

Модель варіантів використання для метеостанції показана на мал. 12.7. У цій моделі метеостанція взаємодіє із зовнішніми об'єктами під час запуску і завершення роботи, при складанні звітів на основі зібраних даних, а також при тестуванні і калібруванні метеорологічних приладів.




*Мал. 12.7. Варіанти використання метеостанції*

Кожен з наявних варіантів використання можна описати за допомогою простої природної мови. Такий опис допомагає розробникам проекту ідентифікувати об'єкти в системі і зрозуміти, що система повинна робити. Я використовую стилізовану форму опису, яка чітко визначає, як відбувається обмін інформацією, як ініціюється взаємодія і так далі Ця форма опису показана в табл. 12.1, де представлений варіант використання **Звіт** (див. мал. 12.7).

**Таблиця 12.1. Опис варіанту використання Звіт**

| **Система** | **Метеостанція** |
|---|---|

| | |
|---|---|
| Варіант використання | Звіт |
| Учасники | Система збору метеорологічних даних, метеостанція |
| Дані | Метеостанція відправляє зведення з даними, знятими з різних приладів в певний часовий період системою збору метеорологічних даних. У повідомленні містяться максимальні, мінімальні і середні значення температури грунту і повітря, атмосферного тиску, швидкості вітру, загальна кількість випавших опадів і напрям вітру, узяті через п'ятихвилинні інтервали часу |
| Вхідні сигнали | Система збору метеорологічних даних встановлює модемний зв'язок з метеостанцією і відправляє запит на передачу даних |
| Відповідь | Підсумкові дані відправляються в систему збору метеорологічних даних |
| Коментарі | Зазвичай від метеостанцій запрошують звіт кожну годину, але ця частота запитів може відрізнятися для різних станцій, а також може змінитися в майбутнє |

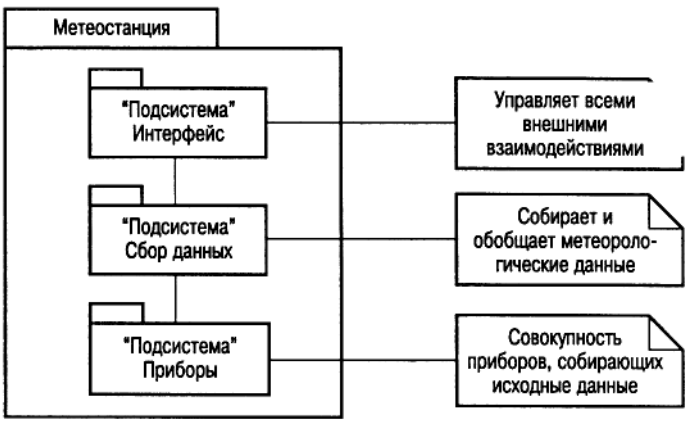
Звичайно, для опису варіантів використання можна удатися до будь-якої іншої методики за умови, що запропонований опис короткий і зрозумілий. Як правило, потрібно розробити описи для всіх варіантів використання, наявних в даній моделі.

Опис варіантів використання допомагає ідентифікувати об'єкти і операції в системі. З опису варіанту використання Звіт видно, що в системі повинні бути об'єкти, що представляють прилади для збору метеорологічних даних, а також об'єкти, що надають підсумкові метеорологічні дані. Повинні також бути операції, що формують запит, і операції, що пересилають метеорологічні дані.

### 12.2.2. Проектування архітектури

Коли взаємодії між проектованою системою ПО і її оточенням визначені, ці дані можна використовувати як основу для розробки архітектури системи. Звичайно, при цьому необхідно застосовувати знання про загальні принципи проектування системної архітектури і дані про конкретну наочну область.

Автоматизована метеостанція є відносно простою системою, тому її архітектуру можна знов представити як багаторівневу модель. На мал. 12.8 усередині великого прямокутника **Метеостанція** розташовано три прямокутники UML. Тут я використовував систему нотації UML (текст в прямокутниках із заломленими кутами) з тим, щоб представити додаткову інформацію.




*Мал. 12.8. Архітектура метеостанції*

У програмному забезпеченні метеостанції можна виділити три рівні.

1. Рівень інтерфейсів, який займається всіма взаємодіями з іншими частинами системи і наданням зовнішніх інтерфейсів системи.
2. Рівень збору даних, керівник збором даних з приладів і узагальнювальний метеорологічні дані перед відправкою їх в систему побудови карт погоди.
3. Рівень приладів, в якому представлені всі прилади, використовувані в процесі збору початкових метеорологічних даних.

У загальному випадку слід спробувати розкласти систему на частини так, щоб архітектура була якомога простіша. Згідно хорошому практичному правилу, модель архітектури повинна складатися не більше ніж з семи основних об'єктів. Кожен такий об'єкт можна описати окремо, проте для того, щоб відобразити структуру цих об'єктів і їх взаємозв'язку, можна скористатися схемою, подібною показаною на мал. 12.6.

### 12.2.3. Визначення об'єктів

Перед виконанням даного етапу проектування вже повинні бути сформовані уявлення щодо основних об'єктів проектованої системи. У системі метеостанції очевидно, що прилади є об'єктами і потрібний принаймні один об'єкт на кожному рівні архітектури. Це прояв основного принципу, згідно якому об'єкти зазвичай з'являються в процесі проектування. Разом з тим потрібно визначити і документувати всі інші об'єкти системи.

Хоча цей розділ названий "Визначення *об'єктів",* насправді на даному етапі проектування визначаються *класи* об'єктів. Структура системи описується в термінах цих класів. Класи об'єктів, визначені раніше, неминуче отримують детальніший опис, тому іноді доводиться повертатися на даний етап проектування для перевизначення класів.

Існує безліч підходів до визначення класів об'єктів.

1. Використання граматичного аналізу природного мовного опису системи. Об'єкти і атрибути – це іменники, операції і сервіси – дієслова [1]. Такий підхід реалізований в ієрархічному методі об'єктно-орієнтованого проектування [295], який широко використовується в аерокосмічній промисловості Європи.
2. Використання як об'єкти ПО подій, об'єктів і ситуацій реального миру з області додатку, наприклад літаків, ролевих ситуацій менеджера, взаємодій, подібних до інтерактивного спілкування на наукових конференціях і так далі [313, 74, 343, 13*, 33*]. Для реалізації таких об'єктів можуть потрібно спеціальні структури зберігання даних (абстрактні структури даних).
3. Застосування підходу, при якому розробник спочатку повністю визначає поведінку системи. Потім визначаються компоненти системи, що відповідають за різні поведінкові акти (режими роботи системи), при цьому основна увага приділяється тому, хто ініціює і хто здійснює дані режими. Компоненти системи, що відповідають за основні режими роботи, вважаються об'єктами [301].
4. Застосування підходу, заснованого на сценаріях, в якому по черзі визначаються і аналізуються різні сценарії використання системи. Оскільки аналізується кожен сценарій, група, що відповідає за аналіз, повинна ідентифікувати необхідні об'єкти, атрибути і операції. Метод аналізу, при якому аналітики і розробники привласнюють ролі об'єктам, показує ефективність підходу, заснованого на сценаріях [33].

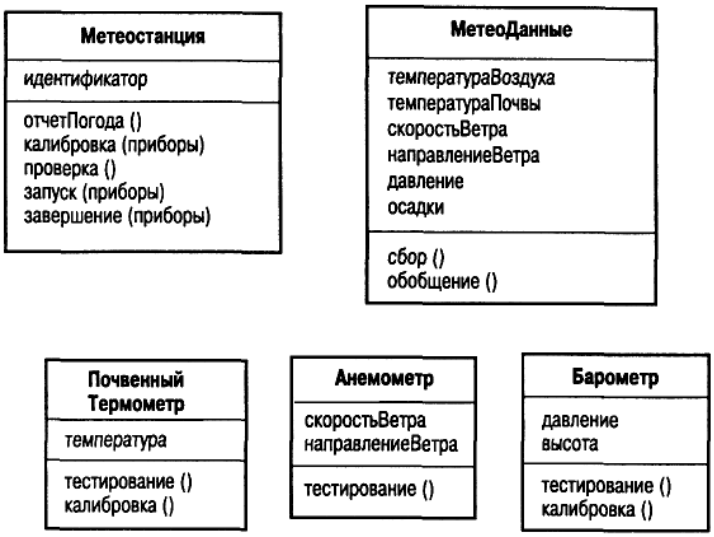
Кожен з описаних підходів допомагає почати процес визначення об'єктів. Але для опису об'єктів і класів об'єктів необхідно використовувати інформацію, отриману з різних джерел. Об'єкти і операції, спочатку визначені на основі неформального опису системи, цілком можуть послужити відправною крапкою при проектуванні. Потім для удосконалення і розширення опису первинних об'єктів можна використовувати додаткову інформацію, отриману з області застосування ПО або аналізу сценаріїв. Додаткову інформацію також можна отрима-

ти в ході обговорення з користувачами системи, що розробляється, або аналізу наявних систем.

При визначенні об'єктів метеостанції я використовую змішаний підхід. Щоб описати всі об'єкти, буде потрібно багато місця, тому на мал. 12.9 я показав тільки п'ять класів об'єктів. **Почвеннийтермометр**, **Анемометр** і **Барометр** є об'єктами області додатку, а об'єкти **Метеостанція** і **Метеоданниє** визначені на основі опису системи і опису сценаріїв (варіантів використання).

Всі об'єкти пов'язані з різними рівнями в архітектурі системи.

1. Клас об'єктів **Метеостанція** надає основний інтерфейс метеостанції при роботі із зовнішнім оточенням. Тому операції класу відповідають взаємодіям, показаним на мал. 12.7. В даному випадку, щоб описати всі ці взаємодії, я використовую один клас об'єктів, але в інших проектах для представлення інтерфейсу системи, можливо, потрібно буде використовувати декілька класів.
2. Клас об'єктів **Метеоданниє** інкапсулює підсумкові дані від різних приладів метеостанції. Пов'язані з ним операції збирають і узагальнюють дані.
3. Класи об'єктів **Почвеннийтермометр**, **Анемометр** і **Барометр** відображають реальні апаратні засоби метеостанції, відповідні операції цих класів повинні управляти даними приладами.




*Мал. 12.9. Приклади класів об'єктів системи метеостанції*

На цьому етапі проектування знання з області додатку ПО можна використовувати для ідентифікації майбутніх об'єктів і сервісів. У нашому прикладі відомо, що метеостанції зазвичай розташовані у видалених місцях. Вони оснащені різними приладами, які іноді дають збій в роботі. Звіт про неполадки в приладах повинен відправлятися автоматично. А це означає, що необхідні атрибути і операції, які перевіряли б правильність функціонування приладів. Крім того, необхідно ідентифікувати дані, зібрані зі всіх станцій; таким чином, кожна метеостанція повинна мати власний ідентифікатор.

Я вирішив не створювати об'єкти, що асоціюються з активними об'єктами кожного приладу. Щоб зняти дані в потрібний час, об'єкти приладів викликають операцію **збір** об'єкту **Метеоданниє**. Активні об'єкти мають власне управління, в нашому прикладі передбачається, що кожен прилад сам визначає, коли потрібно проводити виміри. Проте такий підхід має недолік: якщо ухвалено рішення змінити часовий інтервал збору даних або якщо різні метеостанції збирають дані через різні проміжки часу, тоді необхідно вводити нові класи об'єктів. Створюючи об'єкти приладів, що знімають свідчення за запитом, будь-які зміни в стратегії збору даних можна легко реалізувати без зміни об'єктів, пов'язаних з приладами.

### 12.2.4. Моделі архітектури

Моделі системної архітектури показують об'єкти і класи об'єктів, складових систему, і при необхідності типи взаємин між цими об'єктами. Такі моделі служать мостом між вимогами до системи і її реалізацією. А це означає, що до даних моделей пред'явлені суперечливі вимоги. Вони повинні бути абстрактними настільки, щоб зайві дані не приховували відношення між моделлю архітектури і вимогами до системи. Проте, щоб програміст міг ухвалювати рішення по реалізації, модель повинна містити достатню кількість інформації.

Ці суперечність можна обійти, розробивши декілька моделей різного рівня деталізації. Там, де існують тісні робочі зв'язки між розробниками вимог, проектувальниками і програмістами, можна обійтися однією узагальненою моделлю. В цьому випадку конкретні рішення по архітектурі системи можна приймати в процесі реалізації системи. Коли зв'язки між розробниками вимог, проектувальниками і програмістами не такі тісні (наприклад, якщо система проектується в одному підрозділі організації, а реалізується в іншому), потрібна більш деталізована модель.

Отже, в процесі проектування важливо вирішити, які вимагається моделі і яким повинен бути ступінь їх деталізації. Це рішення залежить також від типу системи, що розробляється. Систему послідовної обробки даних можна спроектувати на основі вбудованої системи реального часу різними способами з використанням різних моделей архітектури. Існує безліч систем, яким потрібні всі види моделей. Але зменшення числа створених моделей скорочує витрати і час проектування.

Існує два типи об'єктно-орієнтованих моделей системної архітектури.

1. Статичні моделі, які описують статичну структуру системи в термінах класів об'єктів і взаємин між ними. Основними взаєминами, які документуються на даному етапі, є відносини узагальнення, відносини "используют-используются" і структурні відносини.
2. Динамічні моделі, які описують динамічну структуру системи і показують взаємодії між об'єктами системи (але не класами об'єктів). Документовані взаємодії містять послідовність складених об'єктами запитів до сервісів і описують реакцію системи на взаємодії між об'єктами.

У мові моделювання UML підтримується величезна кількість можливих статичних і динамічних моделей. Буч (Booch [55]) пропонує дев'ять різних типів схем для представлення моделей. Щоб показати всі моделі, не вистачить місця, та і не всі з них придатно для прикладу з метеостанцією. Тут розглядаються три типи моделей.

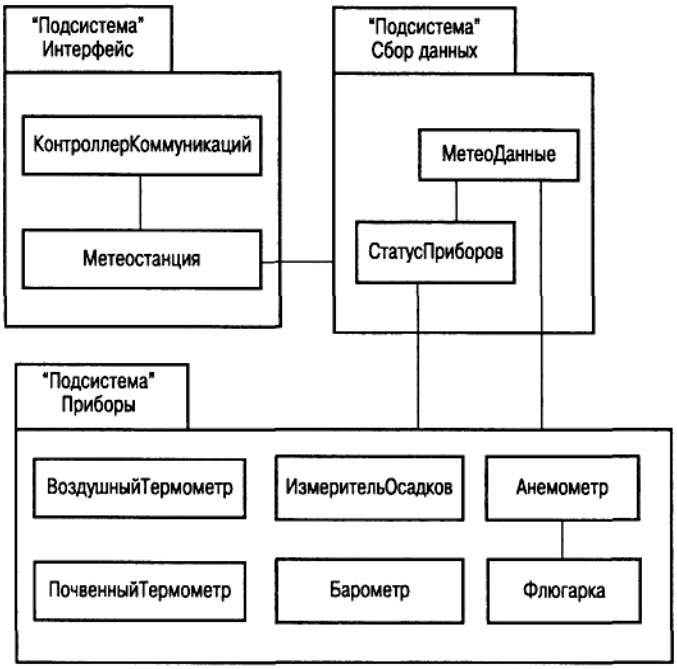
1. Моделі підсистем, які показують логічно згруповані об'єкти. Вони представлені за допомогою діаграми класів, в якій кожна підсистема позначається як пакет. Моделі підсистем є статичними.
2. Моделі послідовностей, які показують послідовність взаємодій між об'єктами. Вони представляються в UML за допомогою діаграм послідовності або кооперативних діаграм. Це динамічні моделі.
3. Моделі кінцевого автомата, які показують зміну стану окремих об'єктів у відповідь на певні події. У UML вони представлені у вигляді діаграм стану. Моделі кінцевого автомата є динамічними.

Інші типи моделей розглянуті раніше в цій і попередніх розділах. Моделі варіантів використання показують взаємодії з системою (див. мал. 12.7, 6.11 і 6.12), моделі об'єктів дають опис класів об'єктів (див. мал. 12.2), моделі узагальнення і спадкоємства (див. мал. 7.8-7.10) показують, які класи є узагальненнями інших класів, модель агрегації (див. мал. 7.11) виявляє взаємозв'язку між колекціями об'єктів.

На мій погляд, модель підсистем є одній з найбільш важливих і корисних статичних моделей, оскільки показує, як можна організувати систему у вигляді логічно зв'язаних груп об'єктів. Ми вже зустрічали приклади такого типу моделі на мал. 12.6, де зображені підсистеми системи побудови карт погоди. У UML пакети є структурами інкапсуляції і не відобража-

ються безпосередньо в об'єктах системи, що розробляється. Проте вони можуть відображатися, наприклад, у вигляді бібліотек Java.

На мал. 12.10 показані об'єкти підсистем метеостанції. У даній моделі також представлені деякі зв'язки. Наприклад, об'єкт **Контроллеркоммуникаций** пов'язаний з об'єктом **Метеостанція**, а об'єкт **Метеостанція** пов'язаний з пакетом **Збір даних**. Сумісна модель пакетів і класів об'єктів дозволяє показати логічно згруповані системні елементи.




*Мал. 12.10. Пакети системи метеостанції*

Модель послідовностей – одна з найбільш корисних і наочних динамічних моделей, яка в кожному вузлі взаємодії документує послідовність взаємодій, що відбуваються між об'єктами. Опишемо основні властивості моделі послідовності.

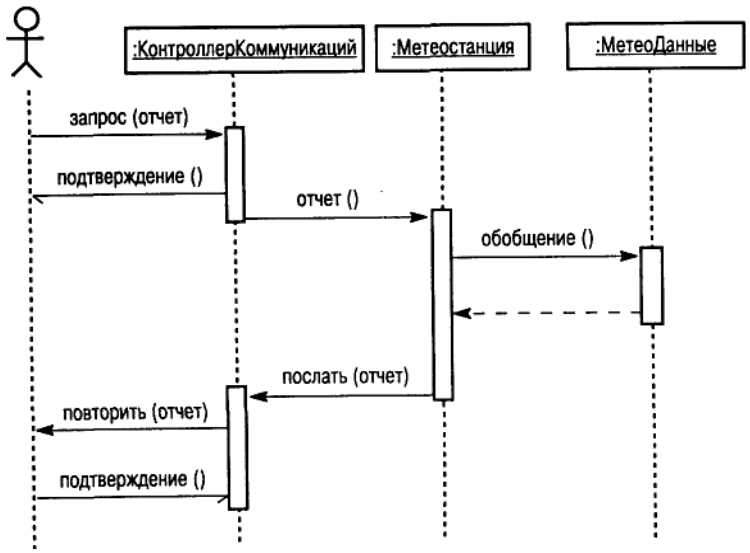
1. Об'єкти, що беруть участь у взаємодії, розташовуються горизонтально вверху діаграми. Від кожного об'єкту виходить пунктирна вертикальна лінія – лінія життя об'єкту.
2. Час направлений зверху вниз по пунктирних вертикальних лініях. Тому в даній моделі легко побачити послідовність операцій.
3. Взаємодії між об'єктами представлені маркірованими стрілками, що зв'язують вертикальні лінії. Це не потік даних, а представлення повідомлень або подій, основних в даній взаємодії.
4. Тонкий прямокутник на лінії життя об'єкту позначає інтервал часу, протягом якого даний об'єкт був об'єктом системи, що управляє. Об'єкт бере на себе управління у верхній частині прямокутника і передає управління іншому об'єкту внизу прямокутника. Якщо в системі є ієрархія викликів, то управління не передається до тих пір, поки не завершиться останнє повернення у виклику первинного методу.

Сказане вище проілюстроване на мал. 12.11, де зображена послідовність взаємодій в той момент, коли зовнішня система посилає метеостанції запит на отримання даних. Діаграму можна прокоментувати таким чином.

1. Об'єкт:контроллеркоммуникаций, що є екземпляром однойменного класу, отримує зовнішній запит "відправити звіт про погоду". Він підтверджує отримання запиту. Половинна стрілка показує, що, відправивши повідомлення, об'єкт не чекає відповіді.
2. Цей об'єкт відправляє повідомлення об'єкту, який є екземпляром класу **Метеостанція**, щоб створити метеорологічний звіт. Об'єкт:контроллеркоммуникаций потім припиняє

роботу (його прямокутник управління закінчується). Використовуваний стиль стрілок показує, що объекты:КонтроллерКоммуникаций и:Метеостанция можуть виконуватися паралельно.

3. Об'єкт, який є екземпляром класу **Метеостанція**, відправляє повідомлення объекту:МетеоДанные, щоб підвести підсумки за метеорологічними даними. Тут інший стиль стрілок указує на те, що объект:Метеостанция чекає відповіді.
4. Після складання зведення, управління передається объекту:Метеостанция. Пунктирна стрілка позначає повернення управління.
5. Цей об'єкт передає повідомлення объекту:КонтроллерКоммуникаций, з якого був присланий запит, щоб передати дані у видалену систему. Потім объект:Метеостанция припиняє роботу.
6. Об'єкт:контроллеркоммуникаций передає звідні дані у видалену систему, отримує підтвердження і потім переходить в стан очікування наступного запиту.



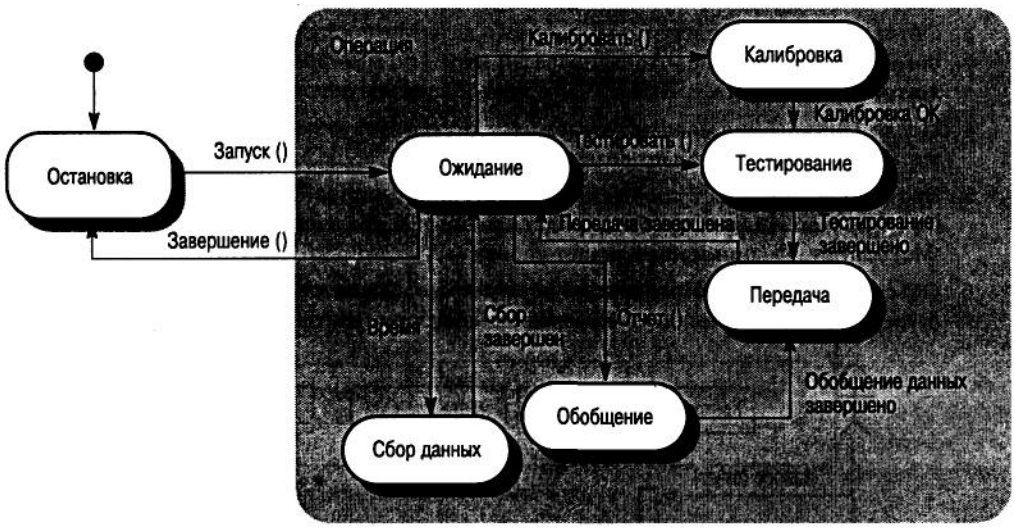

*Мал. 12.11. Послідовність операцій під час збору даних*

З діаграми послідовностей видно, що об'єкти **Контроллеркоммуникаций** і **Метеостанція** насправді є паралельними процесами, виконання яких може припинятися і знову поновлюватися. Тут істотно, що екземпляр об'єкту **Контроллеркоммуникаций** отримує повідомлення від зовнішньої системи, розшифровує отримані повідомлення і ініціалізував дії метеостанції.

При документуванні проекту для кожної значної взаємодії необхідно створювати діаграму послідовностей. Якщо розробляється модель варіантів використання, то діаграму послідовності потрібно створювати для кожного заданого варіанту.

Діаграми послідовностей зазвичай застосовуються при моделюванні комбінованої поведінки груп об'єктів, проте за бажання можна також показати поведінку одного об'єкту у відповідь на оброблювані ним повідомлення. У UML для опису моделей кінцевого автомата використовуються діаграми станів.

На мал. 12.12 представлена діаграма стану об'єкту **Метеостанція**, яка показує реакцію об'єкту на запити від різних сервісів.




*Мал. 12.12. Діаграма станів об'єкту Метеостанція*

Цю діаграму можна прокоментувати таким чином.

1. Якщо об'єкт знаходиться в змозі **Останов**, він може відреагувати тільки повідомленням **запуск().** Потім він переходить в стан очікування подальших повідомлень. Немаркірована стрілка з чорним кружком указує на те, що цей стан є початковим.
2. В змозі **Очікування** система чекає подальших повідомлень. При отриманні повідомлення **завершення()** об'єкт повертається в стан завершення роботи **Останов.**
3. Якщо отримано повідомлення **звіт(),** то система переходить в стан узагальнення даних **Узагальнення**, а потім в стан передачі даних **Передача**, в якому інформація передається через об'єкт **Контроллеркоммуникаций**. Потім система повертається в стан очікування.
4. Отримавши повідомлення **калібрувати(),** система послідовно проходить через стани калібрування, тестування, передачі і лише після цього переходить в стан очікування. У разі отримання повідомлення **тестувати(),** система відразу переходить в стан тестування.
5. Якщо отриманий сигнал **час**, система переходить в стан збору даних, в якому вона збирає дані від приладів. Кожен прилад по черзі також отримує інструкцію "зняти свої дані".

Зазвичай не потрібно створювати діаграми станів для всіх визначених в системі об'єктів. Більшість об'єктів відносно прості, і модель кінцевого автомата просто зайва.

### 12.2.5. Специфікація інтерфейсів об'єктів

Важливою частиною будь-якого процесу проектування є специфікація інтерфейсів між різними компонентами системи. Інтерфейси необхідно визначити так, щоб об'єкти і інші компоненти можна було проектувати паралельно. Визначивши інтерфейс, розробники інших об'єктів можуть вважати, що інтерфейс вже реалізований.

Одному об'єкту не обов'язково повинен відповідати один інтерфейс. Один і той же об'єкт може мати декілька інтерфейсів, причому кожен з них пропонує свій спосіб підтримки методів. Така підтримка є безпосередньо в Java, де інтерфейси оголошуються окремо від об'єктів і об'єкти "реалізують" інтерфейси. Іншими словами, через один інтерфейс можна дістати доступ до набору об'єктів.

Проектування інтерфейсів об'єктів пов'язане із специфікацією інтерфейсу в об'єкті або групі об'єктів. Під цим мається на увазі визначення сигнатур і семантик сервісів, які підтримуються цим об'єктом або групою об'єктів. У UML інтерфейси можна визначити подібно до діаграми класів. Проте розділу властивостей там немає, тому в стандарті UML шаблон «інтерфейс» слід включати в іменну частину.

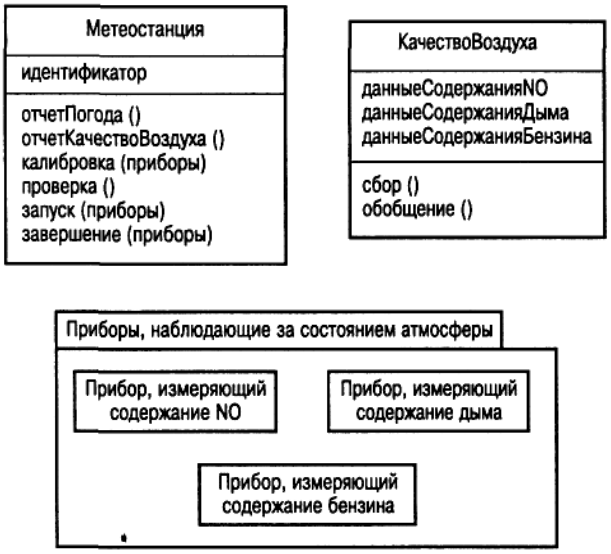
Я віддаю перевагу альтернативному підходу, в якому при визначенні інтерфейсу застосовується мова програмування. У лістингу 12.2 показана специфікація інтерфейсу на мові Java для метеостанції. У міру ускладнення інтерфейсів такий підхід виявляється ефективнішим, оскільки для виявлення помилок і суперечностей в описі інтерфейсу можна скористатися засобами перевірки синтаксису, наявними в компіляторі мови програмування. З представленого опису видно, що деякі методи можуть використовувати різну кількість параметрів. Наприклад, метод завершення без параметрів застосовується до цілої станції, а той же метод з параметрами може відключити один прилад.

**Лістинг 12.2. Опис інтерфейсу метеостанції**

```
interface Метеостанція {

    public void Метеостанція();

    public void запуск();
    public void запуск (Прилад і);

    public void завершення();
    public void завершення(Прилад і);

    public void отчетПогода();

    public void тестувати();
    public void тестувати(Прилад і);

    public void калібрувати(Прилад і);

    public int получитьИдНомер();

    }//Метеостанція
```

## 12.3. Модифікація системної архітектури

Головна перевага об'єктно-орієнтованого підходу до проектування системи полягає в тому, що він спрощує завдання внесення змін в системну архітектуру, оскільки представлення стану об'єкту не робить на неї ніякого впливу. Зміна внутрішніх даних об'єкту не повинна впливати на інші об'єкти системи. Більш того, оскільки об'єкти слабо зв'язані між собою, зазвичай нові об'єкти просто вставляються без значних дій на решту компонентів системи.



*Мал. 12.13. Нові об'єкти для спостереження за забрудненням повітря*

Щоб проілюструвати стабільність об'єктно-орієнтованого підходу, припустимо, що в кожну метеостанцію потрібно було додати можливість спостереження за ступенем забруднення навколишнього середовища, тобто необхідно додати прилади, що вимірюють склад повітря, щоб обчислити кількість різних забруднювачів. Зняті вимірювання по забрудненню повітря передаються з таким же інтервалом часу, що і решта метеорологічних даних. Для модифікації проекту необхідно внести ряд змін.

1. Клас об'єктів, іменований **Качествовоздуха** слід вставити як частина об'єкту **Метеостанція** на одному рівні з об'єктом **Метеоданниє.**
2. У об'єкт **Метеостанція** необхідно додати метод **отчетКачествоВоздуха**, щоб інформація про стан повітря відправлялася на центральний комп'ютер. Програму управління метеостанцією необхідно змінити так, щоб при отриманні запиту з верхнього рівня об'єкту **Метеостанція** здійснювався автоматичний збір даних по забрудненню повітря.
3. Необхідно додати об'єкти, які представляють типи приладів що вимірюють ступінь забруднення повітря. У нашому прикладі можна додати прилади, які вимірювали б рівень оксиду натрію, диму і пари бензину.

На мал. 12.13 показаний об'єкт **Метеостанція** і нові об'єкти, додані в систему. За винятком самого верхнього рівня системи (об'єкт **Метеостанція**) в наявні об'єкти не потрібно було вносити змін. Додавання в систему збору даних про забруднення повітря не зробило ніякого впливу на збір метеорологічних даних.

## КЛЮЧОВІ ПОНЯТТЯ

- При об'єктно-орієнтованому проектуванні основні компоненти програмної системи представляються як об'єкти зі своїми станами і операціями.
- Об'єкти надають сервіси (методи) іншим об'єктам і створюються в реальному часі на основі визначення класу об'єктів.
- Об'єкти можуть бути реалізовані послідовно і паралельно. Паралельний об'єкт може бути пасивним, у якого стан змінюється тільки через його інтерфейс, або активним, який може змінювати свій стан без втручання ззовні.
- Уніфікована мова моделювання UML створена для підтримки систем нотацій, які застосовуються при документуванні об'єктно-орієнтованих проектів.
- Процес об'єктно-орієнтованого проектування складається з наступних етапів: проектування архітектури системи, ідентифікація об'єктів системи, опис архітектури різними моделями об'єктів і документування інтерфейсів об'єктів.
- В процесі об'єктно-орієнтованого проектування можливе створення ряду різних моделей. Всі моделі можна розділити на статичних (моделі класів, моделі узагальнення, моделі агрегації) і динамічних (моделі послідовностей, моделі кінцевого автомата).
- Слід чітко визначати інтерфейси об'єктів, оскільки вони використовуються іншими об'єктами. Для документування інтерфейсів об'єктів можна використовувати мови програмування, наприклад Java.
- Важливою перевагою об'єктно-орієнтованого проектування є те, що він спрощує процес модифікації системи.

### Вправи

**12.1.** Поясните, чому в проектуванні систем застосування підходу, який покладається на слабо зв'язані об'єкти, що приховують інформацію про своє уявлення, приводить до створення системної архітектури, яку потім можна легко модифікувати.

**12.2.** Покажіть на прикладах різницю між об'єктом і класом об'єктів.

**12.3.** За яких умов можна розробляти систему, в якій об'єкти виконуються паралельно?

**12.4.** За допомогою графічної системи нотації UML спроектуйте наступні класи об'єктів з певними атрибутами і операціями:

• телефон;

• принтер персонального комп'ютера;
• персональна стереосистема;
• банківські розрахунки;
• каталог бібліотеки.

**12.5.** Розробіть детальніший проект метеостанції, додавши опису інтерфейсів об'єктів, зображених на мал. 12.9. Вони можуть бути записані за допомогою мов Java, C++ або UML.

**12.6.** Розробіть проект метеостанції, що показує взаємодію між підсистемою збору даних і приладами, що збирають дані. Скористайтеся діаграмою послідовностей.

**12.7.** Визначите можливі об'єкти в наступних системах, застосовуючи при цьому об'єктно-орієнтований підхід.

- Система "Щоденник групи" підтримує розклад зборів і зустрічей в групі співробітників. Для організації зустрічі, в якій бере участь група людей, система знаходить загальні для всіх особистих щоденників вільні "вікна" і призначає цю зустріч на певний час. Якщо система не знаходить загальних "вікон", то починає взаємодіяти з користувачами, щоб реорганізувати особисті щоденники і тим самим створити "вікно" для зустрічі.
- Встановлена повністю автоматизована бензоколонка. Водій вставляє кредитну картку в прочитуючий пристрій, пов'язаний з насосом; картка по лініях комунікацій перевіряється кредитною компанією, встановлюється необхідна кількість бензину. Потім автомобіль заправляється горючим. Коли подача припиняється, з кредитної картки водія знімається вартість отриманого бензину. Кредитна картка повертається після вирахування водієві. Якщо картка невірна, вона повертається водієві перед подачею палива.

**12.8.** Запишіть точні визначення інтерфейсів на мові Java або C++ для об'єктів, визначених у вправі 12.7.

**12.9.** Намалюйте діаграму послідовностей, в якій відображені взаємодії між об'єктами в системі "Щоденник групи".

**12.10.** Намалюйте діаграму станів, на якій відображені можливі зміни станів для одного або більш за об'єкти, визначені у вправі 12.7.

# 13. Проектування систем реального часу

## Цілі

Мета справжнього розділу – познайомити з технологією проектування систем реального часу і з деякою загальною архітектурою таких систем. Прочитавши цей розділ, ви винні:

- ❑ знати основні концепції систем реального часу і розуміти, чому ці системи зазвичай реалізовані у вигляді паралельних процесів;
- ❑ освоїти основні етапи процесу проектування систем реального часу;
- ❑ знати призначення програми системи реального часу, що управляє;
- ❑ познайомитися із загальною архітектурою процесів систем спостереження і управління, а також систем збору даних.

В даний час комп'ютери застосовуються для управління широким спектром різноманітних систем, починаючи від простих домашніх пристроїв і закінчуючи крупними промисловими комплексами. Ці комп'ютери безпосередньо взаємодіють з апаратними пристроями. Програмне забезпечення таких систем (комп'ютер, що управляє, плюс керовані об'єкти) є вбудованою системою реального часу, завдання якої – реагувати на події, що генеруються устаткуванням, тобто у відповідь на ці події виробляти сигнали, що управляють. Таке ПО *вбудовується* у великі апаратні системи і повинне забезпечувати реакцію на події, що відбуваються в оточенні системи, в режимі *реального часу.*

Системи реального часу відрізняються від інших типів програмних систем. Їх коректне функціонування залежить від здатності системи реагувати на події через заданий (як правило, короткий) інтервал часу. От як я визначаю систему реального часу.

> Система реального часу – це програмна система, правильне функціонування якої залежить від результатів її роботи і від періоду часу, протягом якого отриманий результат. "М'яка" система реального часу – це система, в якій операції *віддаляються,* якщо протягом певного інтервалу часу не виданий результат. "Жорстка" система реального часу – це система, операції якої стають *некоректними,* тобто виробляється сигнал про помилку, якщо протягом певного інтервалу часу результат не виданий.

Систему реального часу можна розглядати як систему "стимул-відгук". При отриманні певного вхідного стимулу (вхідного сигналу) система генерує пов'язаний з ним відгук (у відповідь дія або у відповідь сигнал). Отже, поведінку системи реального часу можна визначити за допомогою списку вхідних сигналів, що отримуються системою, пов'язаних з ними у відповідь сигналів (відгуків) і інтервалу часу, протягом якого система повинна відреагувати на вхідний сигнал.

Вхідні сигнали діляться на два класи.

1. *Періодичні сигнали, відбуваються* через зумовлені інтервали часу. Наприклад, система перевіряє датчик кожні 50 мілісекунд, і робить дії (реагує) залежно від значень, отриманих від датчика (стимулу).
2. *Аперіодичні сигнали* відбуваються нерегулярно. Зазвичай вони "повідомляють про себе" за допомогою механізму переривань. Прикладом аперіодичного сигналу може бути переривання, яке виробляється після закінчення передачі вхід/вихід і розміщення даних в буфері обміну.

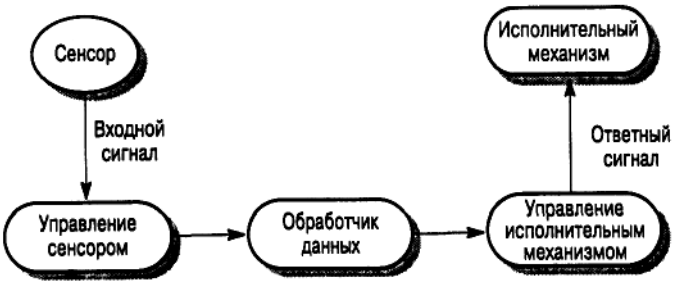
У системах реального часу періодичні вхідні сигнали зазвичай генеруються сенсорами (датчиками), що взаємодіють з системою. Вони надають інформацію про стан зовнішнього оточення системи. Системні відгуки (у відповідь сигнали) прямують групі виконавчих механізмів, керівників апаратними пристроями, які потім впливають на оточення системи. Аперіодичні вхідні сигнали можуть генеруватися і сенсорами, і виконавчими механізмами. Як

правило, аперіодичні сигнали означають виняткові ситуації, наприклад помилки в роботі апаратури. На мал. 13.1 показана модель "сенсор-система-исполнительный механізм" для вбудованої системи реального часу.


*Мал. 13.1. Загальна модель системи реального часу*

Системи реального часу повинні реагувати на вхідні сигнали, що відбуваються в різні моменти часу. Отже, архітектуру такої системи необхідно організувати так, щоб управління переходило до відповідного обробника щонайшвидше після отримання вхідного сигналу. У послідовних програмах такий механізм передачі управління неможливий. Тому зазвичай системи реального часу проектують як безліч паралельних взаємодіючих процесів. Частина системи - програма, що управляє, часто звана диспетчером, - управляє всіма процесами.

Більшість моделей "стимул-відгук" систем реального часу зводяться до узагальненої архітектурної моделі, що складається з трьох типів процесів (мал. 13.2). Для кожного типу сенсора є процес управління сенсором; обчислювальний процес визначає необхідний у відповідь сигнал на отриманий системою вхідний сигнал; процеси управління виконавчими механізмами управляють діями цих механізмів. Така модель дозволяє швидко зібрати дані зі всіх наявних сенсорів (до того, як відбудеться наступне введення даних), обробити їх і отримати у відповідь сигнал від відповідного виконавчого механізму.




*Мал. 13.2. Процеси управління сенсорами і виконавчими механізмами*

## 13.1. Проектування систем

Як указувалося в розділі 2, в процесі проектування системи ухвалюються рішення про те, які властивості будуть реалізовані програмною частиною системи, а які - апаратними засобами. Тимчасові обмеження і інші вимоги припускають, що деякі функції системи, наприклад обробку сигналів, необхідно реалізувати на спеціально розробленому устаткуванні. Таким чином, процес проектування систем реального часу включає проектування устаткування (апаратура) спеціального призначення і проектування програмного забезпечення.

Апаратні компоненти забезпечують вищу продуктивність, ніж еквівалентне їм (по виконуваних функціях) програмне забезпечення. Апаратним засобам можна доручити "вузькі" місця системної обробки сигналів і, таким чином, уникнути дорогої оптимізації ПО. Якщо за продуктивність системи відповідають апаратні компоненти, при проектуванні ПО основну

увагу можна приділити його переносимості, а питання, пов'язані з продуктивністю, відходять на другий план.

Вирішення про розподіл функцій по апаратних і програмних компонентах слід приймати якомога пізніше, оскільки архітектура системи повинна складатися з автономних компонентів, які можна реалізувати як апаратний, так і програмно. Саме така структура відповідає цілям розробника, що проектує зручну в обслуговуванні систему. Отже, результатом процесу проектування високоякісної системи повинна бути система, яку можна реалізувати і апаратними і програмними засобами.

Відмінність процесу проектування систем реального часу від інших систем полягає в тому, що вже на перших етапах проектування необхідно враховувати час реакції системи. В центрі процесу проектування системи реального часу – події (вхідні сигнали), а не об'єкти або функції. Процес проектування таких систем складається з декількох етапів.

1. Визначення безлічі вхідних сигналів, які оброблятимуться системою, і відповідних їм системних реакцій, тобто у відповідь сигналів.
2. Для кожного вхідного сигналу і відповідного йому у відповідь сигналу обчислюються тимчасові обмеження. Вони застосовуються до обробки як вхідних, так і у відповідь сигналів.
3. Об'єднання процесів обробки вхідних і у відповідь сигналів у вигляді сукупності паралельних процесів. У коректній моделі системної архітектури кожен процес пов'язаний з певним класом вхідних і у відповідь сигналів (як показано на мал. 13.2).
4. Розробка алгоритмів, що виконують необхідні обчислення для всіх вхідних і у відповідь сигналів. Щоб отримати уявлення про об'єми обчислювальних і тимчасових витрат в процесі обробки сигналів, розробка алгоритмів зазвичай проводиться на ранніх етапах процесу проектування.
5. Розробка тимчасового графіка роботи системи.
6. Збірка системи, що працює під управлінням диспетчера, – програми, що управляє.

Звичайно, описаний процес проектування є ітераційним. Як тільки визначена структура обчислювальних процесів і часовий графік роботи, необхідно зробити всесторонній аналіз і провести імітацію роботи системи, щоб упевнитися в тому, що вона задовольняє тимчасовим обмеженням. В результаті аналізу може виявитися, що система не відповідає тимчасовим вимогам. У такому разі для підвищення продуктивності системи необхідно змінити структуру обчислювальних процесів, алгоритм управління, програму, що управляє, або всі ці компоненти разом.

У системах реального часу складно аналізувати тимчасові залежності. Із-за непередбачуваної природи аперіодичних вхідних сигналів розробники вимушені робити деякі попередні припущення щодо вірогідності поява аперіодичних сигналів. Зроблені припущення можуть виявитися невірними, і після розробки системи її показники продуктивності не задовольнятимуть тимчасовим вимогам. У роботах [86, 62] обговорюються загальні проблеми перевірки тимчасових параметрів систем. У книзі [132] всесторонньо розглянуті методи, використовувані при аналізі продуктивності систем реального часу.

Всі процеси в системі реального часу повинні бути скоординовані. Механізм координації процесів забезпечує виключення конфліктів при використанні загальних ресурсів. Коли один процес використовує загальний ресурс (або об'єкт), інші процеси не повинні мати доступу до цього ресурсу. До механізмів, що забезпечують взаємне виключення процесів, відносяться семафори [97], монітори [162] і метод критичних областей [59]. Тут я не розглядатиму ці механізми, оскільки всі вони добре документовані в описах операційних систем [332, 318].

### 13.1.1. Моделювання систем реального часу

Системи реального часу повинні реагувати на події, що відбуваються через нерегулярні інтервали часу. Такі події (або вхідні сигнали) часто приводять до переходу системи з одного

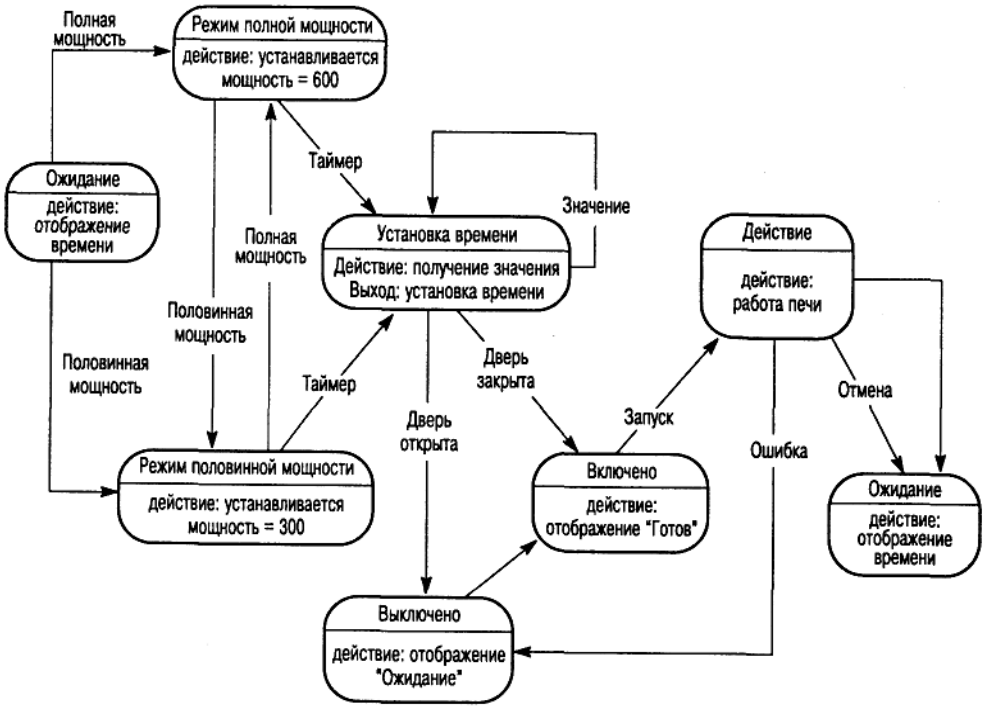
стану в інше. Тому одним із способів опису систем реального часу може бути модель кінцевого автомата і відповідна діаграма станів, розглянуті в розділі 7.

У моделі кінцевого автомата в кожен момент часу система знаходиться в одному зі своїх станів. Отримавши вхідний сигнал, вона переходить в інший стан. Наприклад, система управління клапаном може перейти із стану "Клапан відкритий" в стан "Клапан закритий" після отримання певної команди оператора (вхідний сигнал).

Описаний вище підхід до моделювання системи я проілюструю на розглянутому в розділі 7 прикладі мікрохвильової печі. На мал. 13.3 показана модель кінцевого автомата для звичайної мікрохвильової печі, обладнаної кнопками включення живлення, таймера і запуску системи. Стани системи позначені прямокутниками, що округляють, вхідні сигнали, що викликають перехід системи з одного стану в інше, показані стрілками. На діаграмі показані всі стани печі, також названі дії виконавчих механізмів системи або дії з виводу інформації.

Проглядати послідовність роботи системи потрібно зліва направо. У початковому стані **Очікування**, користувач може вибрати режим повної або половинної потужності. Наступний стан наступає при натисненні на кнопку таймера і установці часу роботи печі. Якщо двері печі закриті, система переходить в стан **Дію**. У цьому стані йде процес приготування їжі, після завершення якого пекти повертається в стан **Очікування**.

Моделі кінцевого автомата – хороший спосіб представлення структури систем реального часу. Тому такі моделі є невід'ємною частиною методів проектування систем реального часу [338]. Метод Харела (Harel) [115], що базується на *діаграмах станів,* направлений на вирішення проблеми внутрішньої складності моделей кінцевого автомата. Діаграма станів структурує моделі таким чином, що групи стану можна було б розглядати як єдина суть. Крім того, за допомогою діаграм станів паралельні системи можна представити у вигляді моделі станів. Моделі станів підтримуються також UML [304, 30*]. У цій книзі я також використовую систему нотації, прийняту в UML.




*Мал. 13.3. Модель кінцевого автомата для мікрохвильової печі*

### 13.1.2. Програмування систем реального часу

На архітектуру системи реального часу робить вплив мова програмування, яка використовується для реалізації системи. До цих пір жорсткі системи реального часу часто програмуються на асемблерних мовах. Мови більш високого рівня також дають можливість згенерувати ефективний програмний код. Наприклад, мова З дозволяє писати вельми ефективні про-

грами. Проте в нім немає конструкцій, що підтримують паралельність процесів і управління спільно використовуваними ресурсами. Крім того, програми на З часто складні для розуміння.

Мова Ada спочатку розроблялася для реалізації вбудованих систем, а тому має в своєму розпорядженні такі засоби, як управління процесами, виключення і правила уявлення. Його засіб *рандеву* (rendezvous) - відмінний механізм для синхронізації завдань (процесів) [63, 24, 4*]. На жаль, перша версія мови Ada (Ada 83) виявилася непридатною для реалізації жорстких систем реального часу. У ній були відсутні засоби, завершення завдань, що дозволяють встановити граничні терміни, не було вбудованих виключень для випадку перевищення граничних термінів і пропонувався строгий алгоритм обслуговування черги "першим прийшов - першим вийшов". При перегляді стандартів мови Ada [23] головна увага приділялася саме цим моментам. У переглянутій версії мови підтримуються захищені типи, що дозволило простіше реалізовувати захищені структури даних, що розділялися, і забезпечувати повніший контроль при виконанні і синхронізації завдань. Проте при програмуванні систем реального часу покращувана версія мови Ada все-таки не забезпечує достатнього контролю над жорсткими системами реального часу.

Перші версії мови Java розроблялися для створення невеликих вбудовуваних систем, таких, наприклад, як контроллери пристроїв і приладів. Розробники Java включили декілька засобів для підтримки паралельних процесів у вигляді паралельних об'єктів (потоків) і синхронізованих методів. Але оскільки в подібних системах немає строгих тимчасових обмежень, то і в мові Java не передбачені засоби, що дозволяють управляти плануванням потоків або запускати потоки в конкретні моменти часу.

Тому Java не підходить для програмування жорстких систем реального часу або систем, в яких є строгий часовий графік процесів. Перерахуємо основні проблеми Java як мови програмування систем реального часу.

1. Не можна вказати час, протягом якого повинен виконуватися потік.
2. Неконтрольований процес очищення пам'яті – він може початися у будь-який час. Тому неможливо передбачити поведінку потоків в часі.
3. Не можна визначити розміри черги, пов'язаної з ресурсами, що розділяються.
4. Реалізація віртуальної машини Java відрізняється для різних комп'ютерів.
5. У мові немає засобів для детального аналізу розподілу часу роботи процесорів.

В даний час ведеться робота за рішенням деяких з цих проблем і формується нова версія мови Java для програмування систем реального часу [256]. Проте не зовсім зрозуміло, яким чином цю версію можна відокремити від лежачої в її основі віртуальної машини Java: властивість переносимості мови завжди конфліктувала з характеристиками режиму реального часу.

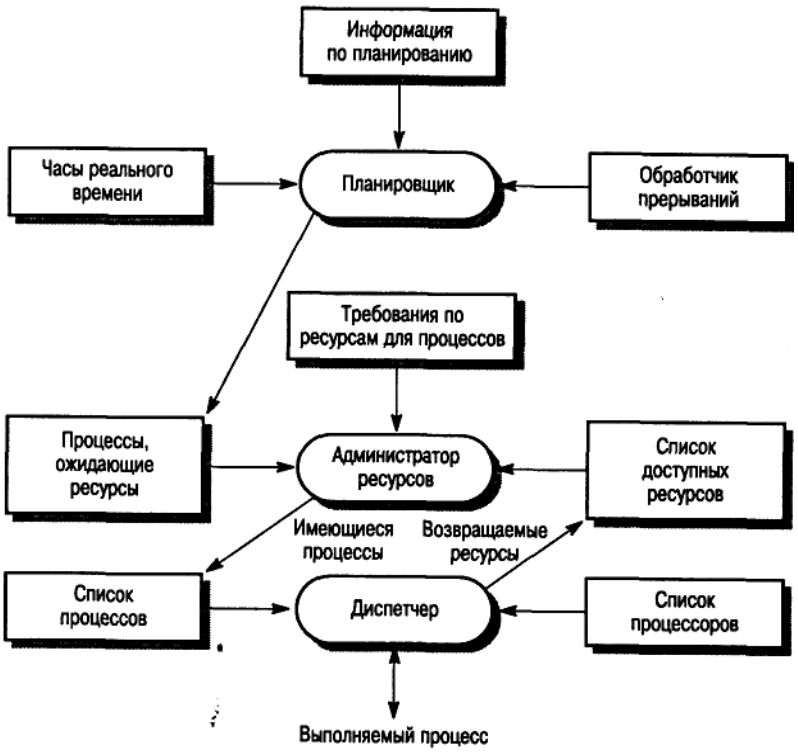
## 13.2. Програми, що управляють

Програма (диспетчер) системи реального часу, що управляє, є аналогом операційної системи комп'ютера. Вона управляє процесами і розподілом ресурсів в системах реального часу, запускає і зупиняє відповідні процеси для обробки вхідних сигналів і розподіляє ресурси пам'яті і процесора. Проте зазвичай в програмах, що управляють, відсутні складніші засоби, властиві операційним системам, наприклад засоби управління файлами.

У роботі [17] представлений повний огляд засобів, необхідних програмам систем реального часу, що управляють. Дана тема обговорюється в монографії [80], де також коротко розглянуті комерційні розробки програм, що управляють, для систем реального часу. Не дивлячись на те що на ринку програмних продуктів існує декілька програм систем реального часу, що управляють, їх часто проектують самостійно як частини систем із-за спеціальних вимог, що пред'являються до конкретних систем реального часу.

Компоненти програми (мал. 13.4), що управляє, залежать від розмірів і складності проектованої системи реального часу. Програми, що зазвичай управляють, за винятком найпростіших, складаються з наступних компонентів.

1. *Годинник реального часу* періодично надає інформацію для планування процесів.
2. *Обробник переривань управляє* аперіодичними запитами до сервісів.
3. *Планувальник* проглядає список процесів, які призначені на виконання, і вибирає один з них.
4. *Адміністратор ресурсів,* отримавши процес, запланований на виконання, виділяє необхідні ресурси пам'яті і процесора.
5. *Диспетчер* запускає на виконання який-небудь процес.




*Мал. 13.4. Компоненти програми реального часу, що управляє*

Програми систем, що надають сервіси на постійній основі, що управляють, наприклад телекомунікаційних або моніторингових систем з високими вимогами до надійності, можуть мати ще декілька компонентів.

- *Конфігуратор* відповідає за динамічну переконфігурацію апаратних засобів [205]. Не припиняючи роботу системи, з неї можна витягувати апаратні модулі і змінити систему за допомогою додавання нових апаратних засобів.
- *Менеджер несправностей* відповідає за виявлення апаратних і програмних несправностей і робить відповідні дії з їх виправлення. Питання відмовостійкої і відновлення систем розглядаються в розділі 18.

Вхідні сигнали, що обробляються системою реального часу, зазвичай мають декілька рівнів пріоритетів. Для одних сигналів, наприклад пов'язаних з винятковими ситуаціями, важливо, щоб їх обробка завершувалася протягом певного інтервалу часу. Якщо процес з вищим пріоритетом запрошує сервіс, то виконання інших процесів повинне бути припинене. Внаслідок цього адміністратор системи повинен уміти управляти принаймні двома рівнями пріоритетів системних процесів.

1. *Рівень переривань* є найвищим рівнем пріоритетів. Він привласнюється тим процесам, на які необхідно швидко відреагувати. Прикладом такого процесу може бути процес годинника реального часу.
2. *Тактовий рівень* пріоритетів привласнюється періодичним процесам.

Ще один рівень пріоритетів може бути у фонових процесів, на виконання яких не накладаються жорсткі тимчасові обмеження (наприклад, процес самотестирования). Ці процеси виконуються тоді, коли є вільні ресурси процесора.

Усередині кожного рівня пріоритетів різним класам процесів можна призначити інші пріоритети. Наприклад, може бути декілька рівнів переривань. Щоб уникнути втрати даних переривання від швидшого пристрою повинне витісняти обробку переривань від повільнішого пристрою.

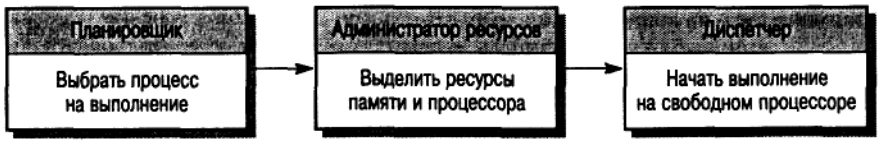
### 13.2.1. Управління процесами

Управління процесами – це вибір процесу на виконання, виділення для нього ресурсів пам'яті і процесора і запуск процесу.

Періодичними називаються процеси, які повинні виконуватися через фіксований зумовлений проміжок часу (наприклад, при зборі даних або управлінні виконавчими механізмами). Програма системи реального часу, що управляє, для визначення моменту запуску процесу використовує свій годинник реального часу. У більшості систем реального часу є декілька класів періодичних процесів з різними періодами (інтервалами часу між виконанням процесів) і тривалістю виконання. Програма, що управляє, повинна бути здатна у будь-який момент часу вибрати процес, призначений на виконання.

Годинник реального часу конфігурується так, щоб періодично подавати тактовий сигнал, період між сигналами складає зазвичай декілька мілісекунд. Сигнал годинника ініціює процес на рівні переривань, який запускає планувальник процесів для управління періодичними процесами. Процес на рівні переривань зазвичай сам не управляє періодичними процесами, оскільки обробка переривань повинна завершуватися щонайшвидше.

Дії, що виконуються програмою, що управляє, при управлінні періодичними процесами, показані на мал. 13.5. Планувальник проглядає список періодичних процесів і вибирає з нього на виконання один процес. Вибір залежить від пріоритету процесу, періоду процесу, передбачуваної тривалості виконання і кінцевих термінів завершення процесу. Іноді за один період між тактовими сигналами годинника необхідно виконати два процеси з різною тривалістю виконання. У такій ситуації один процес необхідно припинити на якийсь час, відповідне його тривалості.




*Мал. 13.5. Дії програми, що управляє, при запуску процесу*

Якщо програмою, що управляє, зареєстровано переривання, це означає, що до одного з сервісів зроблений запит. Механізм переривань передає управління зумовленому елементу пам'яті, в якій міститься команда перемикання на програму обслуговування переривань. Ця програма повинна бути простій, короткою і швидко виконуватися. Під час обслуговування переривань всі інші переривання системою ігноруються. Щоб зменшити вірогідність втрати даних, час перебування системи в такому стані повинен бути мінімальним.

Програма, що виконує сервісну функцію, повинна перекрити доступ наступним перериванням, щоб не перервати саму себе. Вона повинна виявити причину переривання і ініціювати процес з високим пріоритетом для обробки сигналу, що викликав переривання. У деяких системах високошвидкісного збору даних обробник переривань зберігає для подальшої об-

робки дані, які у момент отримання переривання знаходилися в буфері. Після обробки переривання управління знов переходить до програми, що управляє.

У будь-який момент часу може бути декілька призначених на виконання процесів з різними рівнями пріоритетів. Планувальник встановлює порядок виконання процесів. Ефективне планування грає важливу роль, якщо необхідно відповідати вимогам, які пред'являються до системи реального часу. Існує дві основні стратегії планування процесів.

1. *Що не витісняє планування.* Один процес планується на виконання, він запускається і виконується до кінця або блокується по яких-небудь причинах, наприклад при очікуванні введення даних. При такому плануванні можуть виникнути проблеми, пов'язані з тим, що у разі декількох процесів з різними пріоритетами процес з високим пріоритетом повинен чекати завершення процесу з низьким пріоритетом.
2. *Витісняюче планування.* Виконання процесу може бути припинене, якщо до сервісу поступили запити від процесів з вищим пріоритетом. Процес з вищим пріоритетом має перевагу перед процесом з нижчим рівнем пріоритету, і тому йому виділяється процесор.

В рамках цих стратегій розроблена безліч різних алгоритмів планування. До них відноситься циклічне планування, при якому кожен процес виконується по черзі, і планування за швидкістю, коли при першому виконанні отримують вищий пріоритет процеси з коротким періодом виконання [64]. Кожен з алгоритмів планування має певні переваги і недоліки, проте тут ми їх розглядати не будемо.

Інформація про призначений на виконання процес передається адміністраторові ресурсів. Він виділяє для вибраного процесу необхідну пам'ять, а в багатопроцесорній системі – ще і процесор. Потім процес поміщається в "список призначень", тобто в список процесів, призначених на виконання. Коли процесор завершує виконання якого-небудь процесу і стає вільним, викликається диспетчер. Він проглядає наявний список, вибирає процес, який можна виконувати на вільному процесорі, і запускає його на виконання.

## 13.3. Системи спостереження і управління

В даний час можна виділити декілька класів стандартних систем реального часу: моніторингові системи (системи спостереження), системи збору даних, системи управління і ін. Кожному типу систем відповідає особлива структура процесів, тому при проектуванні системи, як правило, архітектуру створюють по одному з існуючих стандартних типів. Таким чином, замість обговорення загальних проблем проектування систем реального часу тут краще розглянути проектування за допомогою узагальнених моделей.

Системи спостереження і управління – важливий клас систем реального часу. Їх основним призначенням є перевірка сенсорів (датчиків), що надають інформацію про оточення системи, і виконання відповідних дій залежно від інформації, що поступила від сенсорів. Системи спостереження виконують дії після реєстрації особливого значення сенсора. Системи управління безперервно управляють апаратними виконавчими механізмами на підставі значень, що отримуються від сенсорів.

Розглянемо наступний приклад.

Хай в будівлі встановлена система охоронної сигналізації. У системі використовується декілька типів сенсорів: датчики рухи, встановлені в окремих кімнатах; датчики на вікнах першого поверху, які подають сигнал, якщо розбивається вікно; дверні датчики, що фіксують відкриття дверей. Всього в системі 50 датчиків на вікнах, 30 на дверях і 200 датчиків руху.

Коли який-небудь датчик фіксує присутність стороннього, система автоматично викликає місцеву поліцію і, використовуючи звуковий синтезатор, повідомляє місцеположення датчика (номер кімнати), від якого йде сигнал. У кімнатах, розташованих біля активного датчика, включається світлова сигналізація і звуковий аварійний сигнал. Система сигналізації

зазвичай включається через мережу, але може працювати і від батарей. Проблеми з електроживленням реєструються спеціальною програмою, контролюючою напругу електромережі. Якщо в мережі реєструється падіння напруги, програма перемикає систему сигналізації на резервне живлення від батарей.

В даному прикладі описана "м'яка" система реального часу, оскільки тут немає жорстких тимчасових вимог. У такій системі не *потрібно* реєструвати події, що відбуваються з високою швидкістю, тому опит датчиків може проводитися двічі в секунду.

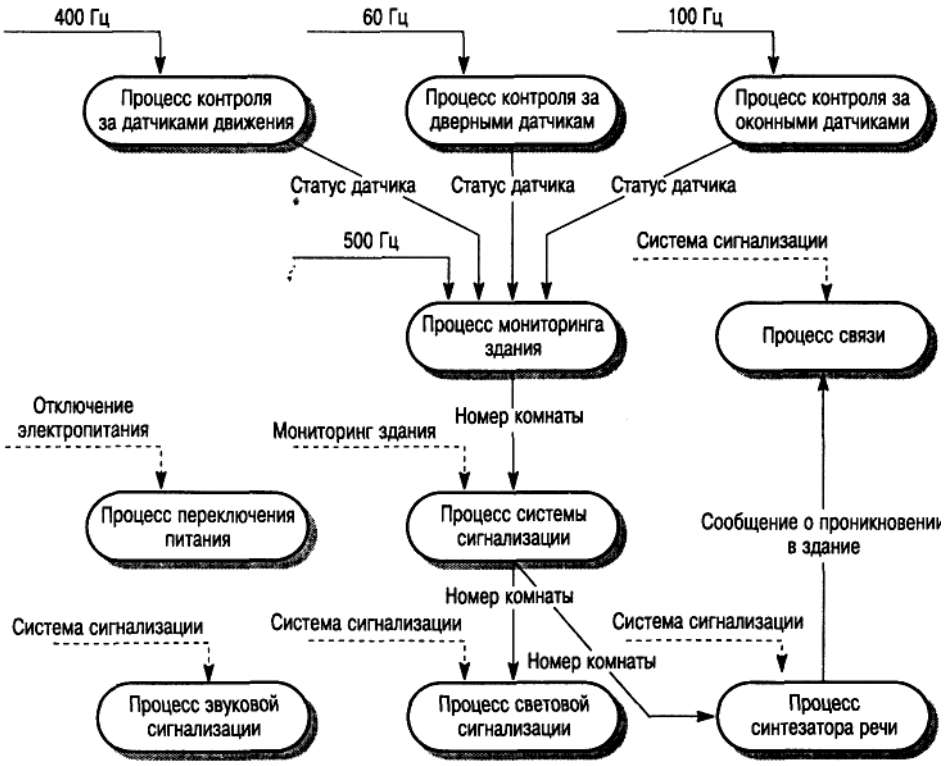
Процес проектування починається з опису аперіодичних вхідних сигналів, що отримуються системою, і пов'язаних з ними реакцій системи. Тут ми обмежимося спрощеним проектом, в якому не враховуються сигнали, породжувані процедурами самоперевірки, і зовнішні сигнали, що генеруються при тестуванні системи або при її виключенні у разі помилкової тривоги. Кінець кінцем система обробляє тільки два типи вхідних сигналів.

1. *Відключення електроживлення* генерується програмою, контролюючою електричний ланцюг. У відповідь на цей сигнал система перемикає мережу на резервне живлення за допомогою подачі сигналу електронному приладу перемикання живлення.
2. *Сигнал про вторгнення* є вхідним і генерується одним з датчиків системи. У відповідь на нього система визначає номер кімнати, в якій знаходиться активний датчик, викликає поліцію, ініціюючи звуковий синтезатор, і включає звуковий сигнал тривоги і світлову сигналізацію будівлі в місці порушення.

На наступному кроці процесу проектування визначаються тимчасові обмеження для кожного вхідного і у відповідь сигналів системи. У табл. 13.1 перераховані ці тимчасові обмеження. До різних типів датчиків, що генерують вхідні сигнали, пред'явлені різні тимчасові вимоги.

**Таблиця 13.1. Тимчасові обмеження на вхідні і у відповідь сигнали системи**

| Сигнал | Тимчасові обмеження |
|---|---|
| Відключення електроживлення | Перемикання на живлення від батарей повинне відбутися протягом 50 мс |
| Сигналізація на дверях | Кожен сигнальний датчик на дверях перевіряється двічі в секунду |
| Сигналізація на вікнах | Кожен датчик на вікні перевіряється двічі в секунду |
| Датчик руху | Кожен датчик руху опитується двічі в секунду |
| Звуковий сигнал | Звуковий сигнал повинен прозвучати через півсекунди після сигналу датчика |
| Включення світлової сигналізації | Світлова сигналізація повинна включитися через півсекунди після сигналу датчика |
| Зв'язок | Виклик в поліцію повинен початися протягом 2 з після сигналу датчика |
| Синтезатор мови | Синтезоване повідомлення повинне бути готове через 4 з після сигналу датчика |




*Мал. 13.6. Архітектура процесів системи охоронної сигналізації*

На наступному етапі проектування розподіляються системні функції по паралельних процесах. Періодично потрібно опитувати три типи датчиків, тому у кожного типу датчиків є пов'язаний з ними процес. Окрім цього, є система управління перериваннями, контролююча електроживлення, система зв'язку з поліцією, синтезатор мови, система включення звукової сигналізації і система, що включає світлову сигналізацію біля датчика. Кожною системою управляє незалежний процес. На мал. 13.6 показана архітектура процесів системи.

На схемі, представленій на мал. 13.6, стрілки (з примітками), що сполучають окремі процеси, позначають потоки даних між процесами з вказівкою типу даних. Написи над стрілками справа над кожним процесом указують систему, що управляє даним процесом. На стрілках вгорі вказано мінімальне значення частоти виконання процесу.

Частота виконання процесів визначається кількістю датчиків і тимчасовими вимогами, що пред'являються системі. Припустимо, що в системі є 30 дверних датчиків, які потрібно перевіряти двічі в секунду. Отже, пов'язаний з дверним датчиком процес повинен виконуватися 60 разів в секунду (частота 60 Гц). Також 400 разів в секунду виконується процес, контролюючий датчик руху.

Аперіодичні процеси позначені стрілками з пунктирними лініями. На цих лініях вказані події, які викликають даний процес. Всі основні виконавчі процеси (звуковій і світловій сигналізації і ін.) починаються командою з процесу **Система безпеки**; їм не потрібні дані з інших процесів. Процесу, керівникові електроживленням, також не потрібні дані з інших частин системи.

Всі представлені процеси можна реалізувати на мові Java як потоки. Лістинг 13.1 містить код Java реалізації процесу **BuildingMonitor** (моніторинг будівлі), що опитує датчики системи. У разі сигналу тривоги програма активізує систему сигналізації. Хай в нашому прикладі система відповідає тимчасовим вимогам. Як вже наголошувалося, в мові Java 2.0 немає засобів для завдання частоти виконання потоків.

**Лістинг 13.1. Реалізація процесу моніторингу будівлі**

```
//Див. Web-страницу http://www.software-engin.com/_
//де представлений повний Java-код цього прикладу
class BuildingMonitor extends Thread {
```

```
    BuildingSensor win, door, move;
    Siren siren = new SIRENO;
    Lights lights = new Lights{};
    Synthesizer synthesizer = new Synthesizer();
    DoorSensors doors = new DoorSensors (30);
    WindowSensors windows = new WindowSensors(50);
    MovementSensors movements = new MovementSensors(200);
    PowerMonitor pm = new PowerMonitor();

    BuildingMonitor()
    {
         //ініціалізація датчиків і запуск процесів
         siren.start();lights.start();
         synthesizer.start();windows.start();
         doors.start()/movements.start();pm.start();
    }
    public void run ()
    {
         int room = 0;
         while (true)
         {
            //перевірка датчиків руху двічі в секунду (400 Гц) move =
            movements.getVal();
            //перевірка віконних датчиків двічі в секунду (100 Гц) win =
            windows.getVal();
            //перевірка дверних датчиків двічі в секунду (60 Гц) door =
            doors.getVal();
            if(move.sensorVal==l|door.sensorVal==l|win.sensorVal==l)
                 {
                      //датчик зареєстрував порушення
                      if(move.sensorVal == 1) room = move.room;
                      if(door.sensorVal == 1) room = door.room;
                      if(win.sensorVal == 1) room = win.room;

                      lights.on(room);siren.on();synthesizer.on(room);
                      break;
                 }
         }
         lights.shutdown() ; siren.shutdown();synthesizer.shutdown();
         windows.shutdown();doors.shutdown();movements.shutdown();
    }//run
)//BuildingMonitor
```

Оскільки дана система не містить строгих тимчасових вимог, її можна реалізувати на мові Java. Звичайно, в Java 2.0 немає ніякої гарантії відповідності тимчасовим специфікаціям. У нашій системі всі датчики опитуються однакова кількість разів, чого не буває в реальних системах.

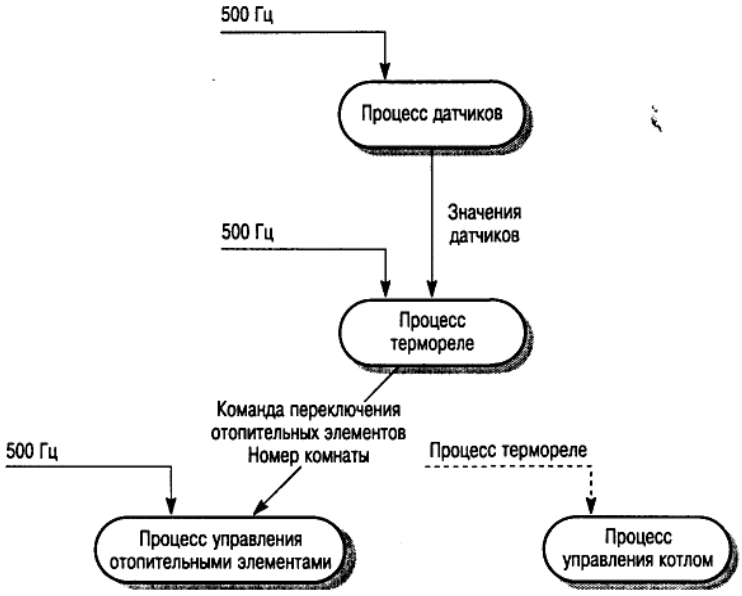
Після визначення архітектури процесів системи починається розробка алгоритмів обробки вхідних сигналів і генерації у відповідь сигналів. Як вже наголошувалося в розділі 3.1, цей етап проектування необхідно виконувати якомога раніше, щоб упевнитися, що система відповідатиме тимчасовим вимогам. Якщо відповідні алгоритми виявляються складними, може виникнути необхідність в зміні тимчасових обмежень. Зазвичай алгоритми систем реального часу порівняно прості. Вони перевіряють елементи пам'яті, виконують деякі прості розрахунки і управляють передачею сигналу. У прикладі системи сигналізації проектування алгоритмів не розглядається.

Останнім етапом процесу проектування є складання тимчасового графіка виконання процесів. У нашому прикладі немає процесів із строгими термінами виконання. Розглянемо пріоритети процесів. Всі процеси, що опитують датчики, мають один і той же пріоритет.

Процесу, керівникові електроживленням, необхідно призначити вищий пріоритет. Пріоритети процесів, керівників системою сигналізації, і процесів, що опитують датчики, повинні бути однакові.

Систему охоронної сигналізації можна віднести швидше до систем спостереження, чим до систем управління, оскільки в ній немає виконавчих механізмів, безпосередньо залежних від значень датчиків. Прикладом системи управління може служити система управління опалюванням будівлі. Система спостерігає за температурними датчиками, встановленими в різних кімнатах будівлі і перемикає нагрівальні прилади залежно від реальної температури і температури, встановленої в термореле. Термореле, у свою чергу, контролює опалювальний казан.

Архітектура процесів такої системи показана на мал. 13.7; в загальному вигляді вона виглядає подібно до системи сигналізації. Подальший розгляд цього прикладу пропонується читачеві як вправа.

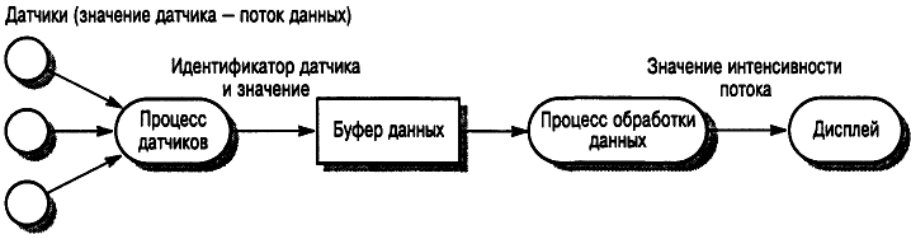



*Мал. 13.7. Архітектура процесів системи управління опалюванням*

## 13.4. Системи збору даних

Ці системи представляють інший клас систем реального часу, які зазвичай базуються на узагальненій архітектурній моделі. Такі системи збирають дані з сенсорів в цілях їх подальшої обробки і аналізу.

Для ілюстрації цього класу систем розглянемо модель, представлену на мал. 13.8. Тут зображена система, що збирає дані з датчиків, які вимірюють потік нейтронів в ядерному реакторі. Дані, зібрані з різних датчиків, поміщаються в буфер, з якого потім витягуються і обробляються. На моніторі оператора відображається середнє значення інтенсивності потоку нейтронів.



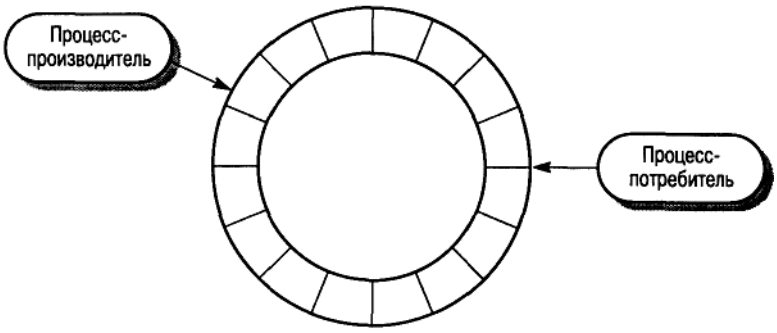

*Мал. 13.8. Архітектура системи спостереження за інтенсивністю потоку нейтронів*

Кожен датчик пов'язаний з процесом, який перетворить аналоговий сигнал, що показує інтенсивність вхідного потоку, в цифровій. Сигнал спільно з ідентифікатором датчика запису-

ється в буфер, де зберігаються дані. Процес, що відповідає за обробку даних, бере їх з буфера, обробляє і передає процесу відображення для виводу на операторну консоль.

У системах реального часу, ведучих збір і обробку даних, швидкості виконання і періоди процесу збору і процесу обробки можуть не співпадати. Якщо обробляються великі об'єми даних, збір даних може виконаються швидше, ніж їх обробка. Якщо ж виконуються тільки прості обчислення, швидше відбувається обробка даних, а не їх збір.

Щоб згладити різницю в швидкостях збору і обробки даних, в більшості подібних систем для зберігання вхідних даних використовується кільцевий буфер. Процеси, що створюють дані (процеси-виробники), поставляють інформацію в буфер. Процеси, оброблювальні дані (процеси-споживачі), беруть дані з буфера (мал. 13.9).




*Мал. 13.9. Кільцевий буфер в системі збору даних*

Очевидно, необхідно запобігти одночасному доступу процесу-виробника і процесу-споживача до одних і тих же елементів буфера. Крім того, система повинна відстежувати, щоб процес-виробник не додавав дані в повний буфер, а процес-споживач не забирав дані з порожнього буфера.

У лістингу 13.2 показана можлива реалізація буфера даних як об'єкту Java. Значення в буфері мають тип SensorRecord (запис даних датчика). Визначені два метода– get і put: метод get бере елементи з буфера, метод put додає елемент в буфер. При оголошенні типу CircularBuffer (кільцевий буфер) програмний конструктор задає розмір буфера.

**Лістинг 13.2. Реалізація кільцевого буфера**

```
class CircularBuffer
{
   int bufsize;
   SensorRecord [] std're;
   int numberOfEntries = 0;
   int front = 0, back = 0;

   CircularBuffer (int n) {
      bufsize = n;
      store = new SensorRecord [bufsize] ;
   }//CircularBuffer

   synchronized void put (SensorRecord rec) throws
InterruptedException
   {
      if(numberOfEntries == bufsize)
         waitt) ;
      store [back] = new SensorRecord (rec.sensorld,rec.sensorVal);
      back = back + 1;
      if(back == bufsize)
         back = 0;
      numberOfEntries = numberOfEntries + 1;
      notify();
```

```
    }//put
    synchronized SensorRecord get() throws InterruptedException
    {
        SensorRecord result = new SensorRecord(-1,-1);
        if(numberOfEntries == 0)
            wait () ;
        result = store [front];
        front = front + 1;
        if(front == bufsize)
            front = 0;
        numberOfEntries = numberOfEntries - 1;
        notify();
        return result;
    }// get
}//CircularBuffer
```

Модифікатор **synchronized**, пов'язаний з методами **get** і **put**, указує на те, що дані методи не повинні виконуватися паралельно. При виклику одного з цих методів система реального часу блокує екземпляр об'єкту, щоб в цей же час не відбувся виклик іншого методу і відповідно не проводилися маніпуляції на тій же ділянці буфера. Виклики методів **wait** і **notify** з методів **get** і **put** гарантують, що вхідні дані не можна покласти в повний буфер або узяти з порожнього буфера. Метод **wait** викликає потік і припиняється, поки інший потік за допомогою методу **notify** не відправить йому повідомлення про зняття очікування. При виклику методу **wait** блокування на захищені дані об'єкту знімається. Метод **notify** відновлює виконання одне з чекаючих потоків.

## КЛЮЧОВІ ПОНЯТЬ

- Система реального часу – це програмна система, яка повинна реагувати на події в реальному масштабі часу. Її коректне функціонування залежить не тільки від отриманих результатів, але і від часу, протягом якого вони отримані.
- Загальна модель архітектури систем реального часу складається з процесів, пов'язаних з кожним класом сенсорів (датчиків) і з кожним виконавчим механізмом. Можуть бути також присутніми інші координуючі процеси.
- Архітектура системи реального часу зазвичай організована як безліч паралельних процесів, що взаємодіють між собою.
- Програма системи реального часу, що управляє, управляє процесами і апаратними ресурсами. Обов'язковим компонентом програми, що управляє, є планувальник, який запускає процес на виконання в заданий час. Планувальник враховує пріоритети процесів.
- Системи спостереження і управління періодично опитують групу сенсорів, що збирають інформацію з оточення системи. За допомогою команд і виконавчих механізмів система реагує на дані, отримані від сенсорів.
- Системи збору даних зазвичай організовуються відповідно до моделі «виробник-споживач». Процес-виробник поміщає дані в кільцевий буфер, де вони використовуються процесом-споживачем. Щоб виключити конфлікти між процесом-виробником і процессом- споживачем, буфер зазвичай реалізують як процес.

## Вправи

**13.1.** Чому системи реального часу зазвичай реалізовані як безліч паралельних процесів? Проілюструйте свою відповідь прикладами.

**13.2.** Поясните, чому об'єктно-орієнтовані методи розробки ПО не завжди підходять до систем реального часу.

**13.3.** Намалюйте діаграми станів керівника ПО для наступних систем.

- Автоматична пральна машина з різними програмами для різних типів білизни.
- Програмне забезпечення для програвача компакт-дисків.
- Телефонний автовідповідач, який записує вхідні повідомлення і відображає кількість отриманих повідомлень на рідкокристалічному екрані. Система повинна визначити телефон що

дзвонив, вивести на екран послідовність чисел (ідентифікованих як тоновий набір) і зберігати записані повідомлення, які потім можна прослуховувати.

- Автомат по видачі напоїв, який може налити каву з молоком і цукром або без них. Користувач кидає монету і за допомогою натиснення кнопок на автоматі вибирає потрібний режим. Автомат видає чашку з розчинною кавою. Користувач потім ставить чашку під кран, натискає іншу кнопку і автомат наливає в чашку гарячу воду.

**13.4.** Використовуючи методи проектування систем реального часу спроектуйте наново систему збору даних від метеостанцій, розглянуту в розділі 12, у вигляді системи "стимул-відповідь".

**13.5.** Спроектуйте архітектуру процесів для системи спостереження, що збирає дані з групи датчиків, що вимірюють склад повітря і розташованих навколо міста. У системі 5000 датчиків, організованих в групи по 100 штук. Кожен датчик повинен перевірятися 4 рази в секунду. Якщо більше 30% датчиків в групі зафіксують, що якість повітря нижча за допустимий рівень, активізується застережливий світловий сигнал. Всі датчики передають зібрані дані центральному комп'ютеру, який кожні 15 мін генерує звіт про склад повітря в місті.

**13.6.** Обговорите сильні і слабкі сторони Java як мови програмування для реалізації систем реального часу.

**13.7.** Система безпеки поїзда автоматично закриває двері, якщо швидкість поїзда перевищує граничну для даної ділянки траси або якщо при виході на ділянку шляху горить червоне світло (тобто в'їзд на ділянку заборонений). Решта подробиць перерахована у врізанні 13.1. Ідентифікуйте вхідні сигнали, які повинні обробляти бортова система управління поїздом, і пов'язані з ними у відповідь сигнали.

**13.8.** Припустите вірогідну архітектуру процесів такої системи.

**13.9.** Якщо в бортовій системі безпеки поїзда при зборі даних з путніх передавачів використовуються періодичні процеси, яку частоту збору даних слід запланувати, щоб система гарантовано отримувала інформацію від передавачів? Обгрунтуйте свою відповідь.

**Врізання 13.1. Опис системи безпеки поїзда**

- Система збирає дані про швидкість на ділянці від путнього передавача, який безперервно передає ідентифікатор ділянки і значення швидкості на цій ділянці. Цей же передавач передає інформацію про статус сигналу управління на ділянці траси. Час передачі всій інформації не повинен перевищувати 50 мс.
- Щоб отримати дані від передавача, відстань між поїздом і передавачем не повинна перевищувати 10 м.
- Максимальна швидкість поїзда 180 км/ч.
- Датчики на поїзді надають інформацію про поточну швидкість поїзда (оновлювану кожні 250 мс), статус поїзда оновлюється кожні 100 мс.
- Якщо на поточній ділянці швидкість поїзда перевищує граничну більш ніж на 5 км/ч, в кабіні. машиніста лунає застережливий сигнал. При перевищенні граничної швидкості більш ніж на 10 км/ч починається автоматичне гальмування поїзда, яке продовжується до тих пір, поки швидкість поїзда не буде рівна граничною на даній ділянці. Гальмування поїзда повинне починатися через 100 мс після реєстрації підвищеної швидкості поїзда.
- Якщо поїзд виїхав на ділянку, перед якою горить червоне світло семафора, система безпеки повинна загальмувати поїзд до повної зупинки. Гальмування повинне початися через 100 мс після реєстрації червоного світлового сигналу від семафора.
- Система постійно оновлює інформацію на екрані в кабіні машиніста.

# 14. Проектування з повторним використанням компонентів

## Цілі

Мета справжнього розділу – показати різні способи повторного використання наявного програмного забезпечення в процесі проектування програмних систем. Прочитавши цей розділ, ви винні:

- ❑ знати основні переваги повторного використання компонентів ПО і проблеми, які можуть виникнути при цьому;
- ❑ познайомитися з різними типами повторно використовуваних компонентів і знати основ-

ні етапи процесу їх проектування;

- ❑ засвоїти, що таке сімейства додатків і чому вони служать ефективним способом повторного використання ПО;
- ❑ знати, що патерни – це абстракції високого рівня, які забезпечують повторне використання компонентів в процесі об'єктно-орієнтованого проектування.

У більшості інженерних розробок процес проектування заснований на повторному використанні вже наявних компонентів. У таких сферах, як механіка або електротехніка, інженери ніколи не розробляють проект "З нуля". Їх проекти базуються на компонентах, вже перевірених і протестованих в інших системах. Як правило, це не тільки малі компоненти, наприклад фланці і клапани, але також цілі підсистеми, наприклад двигуни, компресори або турбіни.

В даний час не викликає сумнівів той факт, що необхідно порівнювати різні підходи до розробки програмного забезпечення. Якщо програмне забезпечення розглядати як актив, то повторне використання цих активів дозволить істотно скоротити витрати на його розробку. Тільки за допомогою систематичного повторного використання ПО можна зменшити витрати на його створення і обслуговування, скоротити терміни розробки систем і підвищити якість програмних продуктів.

Щоб повторне використання ПО було ефективним, його необхідно враховувати на всіх етапах процесу проектування ПО або процесу розробки вимог. Під час програмування можливе повторне використання на етапі підбору компонентів, відповідних вимогам. Проте для систематичного *повторного* використання необхідний такий процес проектування, в ході якого постійно розглядалася б можливість повторного використання вже існуючої архітектури, де система була б явно організована з доступних наявних компонентів ПО.

Метод проектування ПО, заснований на повторному використанні, припускає максимальне використання вже наявних програмних об'єктів. Такі об'єкти можуть радикально розрізнятися розмірами.

1. *Повторно використовувані застосування.* Можна повторно використовувати цілі застосування або шляхом включення їх в систему без зміни інших підсистем (наприклад, комерційні готові продукти, див. розділ 14.1.2), або за допомогою розробки сімейств додатків, що працюють на різних платформах і адаптованих до вимог конкретних замовників (див. розділ 14.2).
2. *Повторно використовувані, компоненти.* Можна повторно використовувати компоненти додатків - від підсистем до окремих об'єктів. Наприклад, система розпізнавання тексту, розроблена як частина системи обробки текстів, може повторно використовуватися в системах управління базами даних. Цей вид повторного використання розглядається в розділі 14.3.
3. *Повторно використовувані функції.* Можна повторно використовувати програмні компоненти, які реалізують окремі функції, наприклад математичні. Заснований на стандартних бібліотеках метод повторного використання застосовується в програмуванні останні 40 років.

Повторне використання цілих застосувань практикується досить широко; при цьому компанії, що займаються розробкою ПО, адаптують свої системи для різних платформ і для роботи в різних умовах. Також добре відоме повторне використання функцій через стандартні бібліотеки, наприклад графічні і математичні. Інтерес до повторного використання компонентів виник ще в початку 1980-х років, проте на практиці такий підхід до розробки систем ПО застосовується лише останні декілька років.

Очевидною перевагою повторного використання ПО є зниження загальної вартості проекту, оскільки в цілому потрібно специфікувати, спроектувати, реалізувати і перевірити меншу кількість системних компонентів. Але зниження вартості проекту – це тільки потенційна перевага повторного використання. Як видно з табл. 14.1, повторне використання ПО має ряд інших переваг.

**Таблиця 14.1. Переваги повторного використання ПО**

| Перевага | Опис |
|---|---|
| Підвищення надійності | Компоненти, повторно використовувані в інших системах, виявляються значно надійнішими за нові компоненти. Вони протестовані і перевірені в різних умовах роботи. Помилки, допущені при їх проектуванні і реалізації, виявлені і усунені ще при першому їх застосуванні. Тому повторне використання компонентів скорочує загальна кількість помилок в системі |
| Зменшення проектних рисок | Для вже існуючих компонентів можна точніше прогнозувати витрати, пов'язані з їх повторним використанням, чим витрати, необхідні на їх розробку. Такий прогноз – важливий чинник адміністрування проекту, оскільки дозволяє зменшити неточності при попередній оцінці кошторису проекту |
| Ефективне використання фахівців | Частина фахівців, що виконують однакову роботу в різних проектах, може займатися розробкою компонентів для їх подальшого повторного використання, ефективно застосовуючи накопичені раніше знання |
| Дотримання стандартів | Деякі стандарти, такі як стандарти інтерфейсу користувача, можна реалізувати у вигляді набору стандартних компонентів. Наприклад, можна розробити повторно використовувані компоненти для реалізації різних меню призначеного для користувача інтерфейсу. Всі застосування надають меню користувачам в одному форматі. Використання стандартного призначеного для користувача інтерфейсу підвищує надійність систем, оскільки, працюючи із знайомим інтерфейсом, користувачі здійснюють менше помилок |
| Прискорення розробки | Часто для успішного просування системи на ринку необхідна якомога раніша її поява, причому незалежно від повної вартості її створення. Повторне використання компонентів прискорює створення систем, оскільки скорочується час на їх розробку і тестування |

Для успішного проектування і розробки ПО з повторним використанням компонентів повинні виконуватися три основні умови.

1. Можливість пошуку необхідних системних компонентів. У організаціях повинен бути каталог документованих компонентів, призначених для повторного використання, який забезпечував би швидкий пошук потрібних компонентів.
2. При повторному використанні необхідно упевнитися, що поведінка компонентів передбачена і надійно. У ідеалі всі компоненти, представлені в каталозі, повинні бути сертифіковані, щоб підтвердити відповідність певним стандартам якості.
3. На кожен компонент повинна бути відповідна документація, мета якої – допомогти розробникові отримати потрібну інформацію про компонент і адаптувати його до нового застосування. У документації повинна міститися інформація про те, де використовується даний компонент, і інші питання, які можуть виникнути при повторному використанні компоненту.

Успішне використання компонентів в додатках Visual Basic, Visual C++ і Java продемонструвало важливість повторного використання. Розробка ПО, заснована на повторному використанні компонентів, стає широко поширеним рентабельним підходом до розробки програмних продуктів [331, 3*].

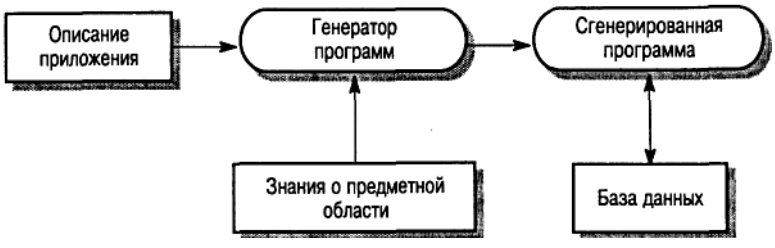
Разом з тим підходу до розробки ПО з повторним використанням компонентів властивий ряд недоліків і проблем (табл. 14.2), які перешкоджають запланованому скороченню витрат на розробку проекту.

**Таблиця 14.2. Проблеми повторного використання**

| Проблема | Опис |
|---|---|
| Підвищення вартості супроводу системи | Недоступність початкової коди компоненту може привести до збільшення витрат на супровід системи, оскільки повторно використовувані системні елементи можуть з часом виявитися не сумісними із змінами, вироблюваними в системі |
| Недостатня інструментальна підтримка | CASE-средства не підтримують розробку ПО з повторним використанням компонентів. Інтеграція цих засобів з системою бібліотек компонентів скрутно або навіть неможливо. Якщо процес розробки ПО здійснюється за допомогою CASE-средств, повторне використання компонентів можна повністю виключити |
| Синдром "винаходу велосипеда" | Деякі розробники ПО вважають за краще переписати компоненти, оскільки вважають, що зможуть при цьому їх удосконалити. Крім того, багато хто вважає, що створення програм "З нуля" перспективнее і "благородніше" за повторне використання написаних іншими програм |
| Зміст бібліотеки компонентів | Заповнення бібліотеки компонентів і її супровід може коштувати дорого. В даний час ще недостатньо добре продумані методи класифікації, каталогізації і витягання інформації про програмні компоненти |
| Пошук і адаптація компонентів | Компоненти ПО потрібно знайти в бібліотеці, вивчити і адаптувати до роботи в нових умовах, що "не укладається" в звичайний процес розробки ПО |

З перерахованого вище виходить, що повторне використання компонентів повинне бути систематичним, плановим і включеним у всі організаційні програми організації-розробника. У Японії повторне використання відоме багато років [231] і є невід'ємною частиною "японського" методу розробки ПО [85]. Багато компаній, наприклад Hewlett-Packard, успішно застосовують повторне використання в своїх розробках [139]. Досвід цієї компанії представлений у фундаментальній книзі [187].

Альтернативою повторному використанню програмних компонентів є застосування програмних генераторів. Згідно цьому підходу інформація, необхідна для повторного використання, записується в систему генератора програм з урахуванням знань про ту наочну область, де експлуатуватиметься система, що розробляється. В даному випадку в системній специфікації повинно бути точно вказано, які саме компоненти вибрані для повторного використання, а також описані їх інтерфейси і те, як вони повинні компонуватися. На основі такої інформації генерується система ПО (мал. 14.1).




*Мал. 14.1. Генерування програм*

Повторне використання, засноване на генераторах програм, можливо тільки тоді, коли можна ідентифікувати наочні абстракції і їх відображення у виконуваний код. Тому для компоновки і управління наочними абстракціями використовуються, як правило, проблемнозависимые мови (наприклад, мови четвертого покоління). Ось наочні області, в яких застосування такого підходу може бути успішним.

1. *Генератори додатків для обробки економічних даних.* На вході генератора – опис додатку на мові четвертого покоління або діалогова система, де користувач визначає екранні форми і способи обробки даних. На виході – програма на якій-небудь мові програмування, наприклад COBOL або SQL.
2. *Генератори програм синтаксичного аналізатора.* На вході генератора – граматичний опис мови, на виході – програма граматичного розбору мовних конструкцій.
3. *Генератори код CASE-средств.* На вході генераторів – архітектура ПО, а на виході - програмна реалізація проектованої системи.

Розробка ПО з використанням програмних генераторів економічно вигідна, проте істотно залежить від повноти і коректності визначення абстракцій наочної області. Даний підхід можна широко використовувати в перерахованих вище наочних областях і у меншій мірі при розробці систем управління і контролю [261]. Головна перевага цього підходу полягає у відносній легкості розробки програм за допомогою генераторів. Проте необхідність глибокого розуміння наочної області і її моделей обмежує застосовність даного методу.

## 14.1. Покомпонентна розробка

Метод покомпонентної розробки ПО з повторним використанням компонентів з'явився в кінці 1990-х років як альтернатива об'єктно-орієнтованому підходу до розробки систем, який не привів до повсюдного повторного використання програмних компонентів, як передбачалося спочатку. Окремі класи об'єктів виявилися дуже деталізованими і специфічними: їх потрібно було пов'язувати з додатком або під час компіляції, або при компоновці системи. Використання класів зазвичай припускає наявність детальних даних про класи, що робить доступним початковий код, але для комерційних продуктів початковий код відкритий дуже рідко. Не дивлячись на ранні оптимістичні прогнози, значний розвиток ринку окремих програмних об'єктів і компонентів так і не відбувся.

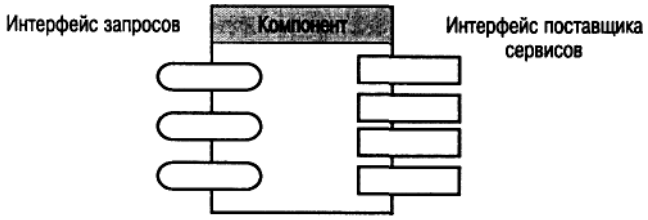
Компоненти абстрактніші, ніж класи об'єктів. Тому їх можна вважати незалежними постачальниками сервісів. Якщо система запрошує який-небудь сервіс, викликається компонент, що надає цей сервіс незалежно від того, де виконується компонент і на якій мові написаний. Прикладом простого компоненту може бути окрема математична функція, що обчислює, наприклад, квадратний корінь числа. Для обчислення квадратного кореня програма викликає компонент, який може виконати дане обчислення. На іншому кінці масштабної лінійки компонентів знаходяться системи, які надають повний обчислювальний сервіс.

Погляд на компонент як на постачальника сервісів визначається двома основними характеристиками компонентів, що допускають їх повторне використання.

1. Компонент – це незалежно виконуваний програмний об'єкт. Початковий код компоненту може бути недоступний, тому такий компонент не компілюється спільно з іншими компонентами системи.
2. Компоненти оголошують свій інтерфейс і всі взаємодії з ними здійснюються з його допомогою. Інтерфейс компоненту описується в термінах операцій, що параметризуються, а внутрішній стан компоненту завжди приховано.

Компоненти визначаються через свої інтерфейси. В більшості випадків компоненти можна описати у вигляді двох взаємозв'язаних інтерфейсів, як показано на мал. 14.2.

- *Інтерфейс постачальника сервісів,* який визначає сервіси, що надаються компонентом.
- *Інтерфейс запитів,* який визначає, які сервіси доступні компоненту з системи, що використовує цей компонент.




*Мал. 14.2. Інтерфейси компоненту*

Як приклад розглянемо компонент (мал. 14.3), який надає сервіси виведення документів на друк. У нашому прикладі підтримуються наступні сервіси: друк документів, проглядання стану черги на конкретному принтері, реєстрація і видалення принтерів з системи, передача документа з одного принтера на інший і видалення документа з черги на друк. Дуже важливо, щоб комп'ютерна платформа, на якій виконується компонент, надавала сервіс (назвемо його **Файлоппрінтер**), що дозволяє витягувати файл опису принтера, і сервіс **Компрінтер**, передавальний команди на конкретний принтер.


*Мал. 14.3. Компонент служби друку*

Компоненти можуть існувати на різних рівнях абстракції – від простий бібліотечної підпрограми до цілих застосувань, таких як Microsoft Excel. У роботі [236] визначено п'ять рівнів абстракції компонентів.

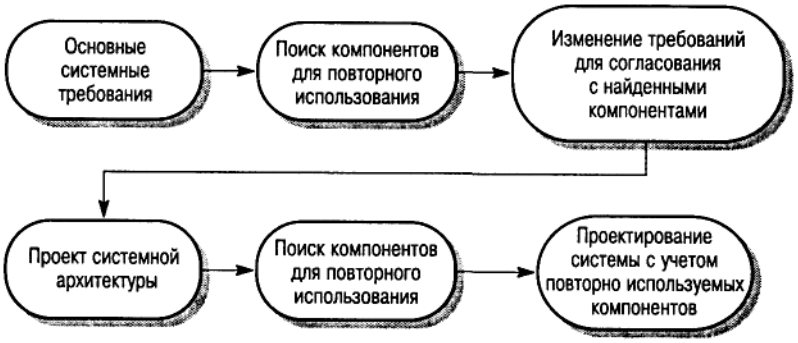
1. *Функціональна абстракція.* Компонент реалізує окрему функцію, наприклад математичну. По суті, інтерфейсом постачальника сервісів тут є сама функція.
2. *Безсистемне угрупування.* В даному випадку компонент – це набір слабо зв'язаних між собою програмних об'єктів і підпрограм, наприклад оголошень даних, функцій і тому подібне Інтерфейс постачальника сервісів складається з назв всіх об'єктів в угрупуванні.
3. *Абстракції даних.* Компонент є абстракцією даних або класом, описаним на об'єктно-орієнтованій мові. Інтерфейс постачальника сервісів складається з методів (операцій), що забезпечують створення, зміну і діставання доступу до абстракції даних.
4. *Абстракції кластерів.* Тут компонент – це група зв'язаних класів, що працюють спільно. Такі компоненти іноді називають структурою. Інтерфейс постачальника сервісів є композицією всіх інтерфейсів об'єктів, складових структуру (див. розділ 14.1.1).
5. *Системні абстракції.* Компонент є повністю автономною системою. Повторне використання абстракцій системного рівня іноді називають повторним використанням комерційних продуктів. Інтерфейсом постачальника сервісів на цьому рівні є так званий програмний інтерфейс додатків (Application Programming Interface – API), який надає

доступ до системних команд і методів. Повторне використання комерційних продуктів розглядається в розділі 14.1.2.

Підхід покомпонентної розробки систем можна інтегрувати в загальний процес створення ПО шляхом додавання спеціальних етапів, на яких відбираються і адаптуються повторно використовувані компоненти (мал. 14.4). Проектувальники системи розробляють системну архітектуру на високому рівні абстракції, складаючи специфікації системних компонентів. Надалі ці специфікації використовуються для пошуку повторно використовуваних компонентів. Вони включаються в систему або на рівні системної архітектури, або на нижчих деталізованих рівнях.


*Мал. 14.4. Інтеграція повторного використання компонентів в процес розробки ПО*

Хоча такий підхід може привести до значного збільшення кількості повторно використовуваних компонентів, він, по суті, протилежний підходу, вживаному в інших інженерних дисциплінах, де процес проектування підпорядкований ідеї повторного використання. Перед початком етапу проектування розробники виконують пошук компонентів, відповідних для повторного використання. Системна архітектура будується на основі вже наявних (готових) компонентів (мал. 14.5).




*Мал. 14.5. Процес проектування з повторним використанням компонентів*

В даному випадку вимоги до системи змінюються з урахуванням наявних компонентів, вибраних для повторного використання. Системна архітектура також базується на наявних компонентах. Звичайно, такий підхід припускає певні компроміси в реалізації вимог. Хоча така система може виявитися менш ефективною, чим система, розроблена без повторного використання компонентів, цей недолік компенсується нижчою вартістю розробки, вищими темпами створення системи і її підвищеною надійністю.

Зазвичай покомпонентний процес реалізації системи є або процесом макетування (прототипирования), або процесом покрокової розробки. Разом з бібліотеками компонентів можна використовувати стандартні мови програмування, наприклад Java. Альтернативою йому (і поширенішою мовою) є мова сценаріїв, яка спеціально розроблена для інтеграції повторно використовуваних компонентів і забезпечує швидку розробку програм.

Першою мовою сценаріїв, розробленою для інтеграції повторно використовуваних компонентів, був Unix shell [56]. Після нього розроблена безліч інших мов сценаріїв, наприклад Visual Basic і TCL/TK. У роботі [268] обговорюються переваги мов сценаріїв, зокрема відсутність визначення типів даних, і той факт, що вони не компілюються, а інтерпретуються.

Мабуть, головною проблемою, пов'язаною з покомпонентною розробкою систем, є їх супровід і модернізація. При зміні вимог до системи часто в компоненти необхідно внести зміни, відповідні новим вимогам, проте в більшості випадків це неможливо, оскільки початко-

вий код компонентів недоступний. Також, як правило, не підходить альтернативний варіант: заміна одного компоненту іншим. Таким чином, необхідна додаткова робота по адаптації повторно використовуваних компонентів, що приводить до підвищення вартості обслуговування системи. Проте, оскільки розробка з повторним використанням компонентів дозволяє швидше створювати ПО, організації згодні оплачувати додаткові витрати на супровід і модернізацію систем.

### 14.1.1. Об'єктні структури додатків

Перші прихильники об'єктно-орієнтованої розробки ПО вважали, що найбільш відповідними абстракціями для повторного використання є об'єкти. Проте, як випливає з попереднього розділу, об'єкти зазвичай дуже дрібні структурні одиниці, занадто "прив'язані" в конкретному застосуванню. Очевидно, що в процесі об'єктно-орієнтованого проектування замість повторного використання об'єктів набагато ефективніше повторно використовувати крупномодульные абстракції, так звані об'єктні структури додатку.

Об'єктними структурами додатку є структури підсистем, що складаються з безлічі абстрактних і конкретних класів об'єктів і інтерфейсів між ними [343]. Окремі деталі підсистем реалізуються за допомогою компонентів і забезпечують конкретні реалізації абстрактних класів. Об'єктні структури, як правило, немає самі по собі додатками. Зазвичай додатки будуються за допомогою інтеграції декількох об'єктних структур.

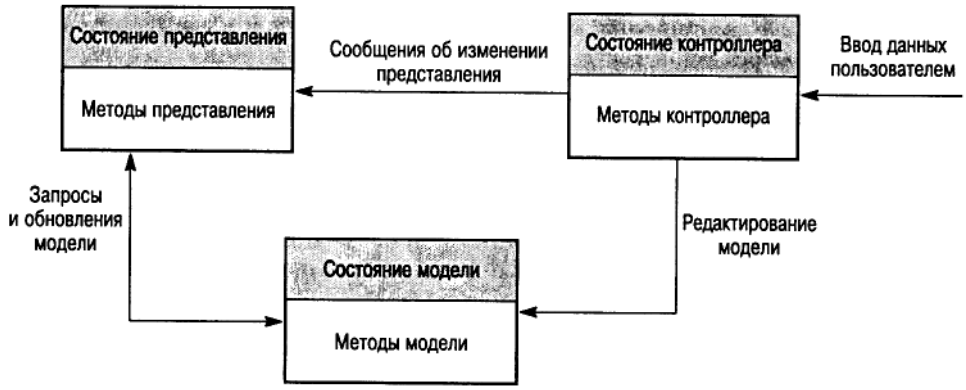
Існує три основні класи об'єктних структур [113].

1. *Інфраструктури систем.* Забезпечують розробку інфраструктур для систем зв'язку (комунікаційних систем), призначених для користувача інтерфейсів і компіляторів [306].
2. *Інтеграційні структури.* Як правило, складаються з набору стандартів і пов'язаних з ними класів об'єктів, забезпечуючу взаємодію і обмін даними між компонентами. До цього типу структур відносяться CORBA, COM і DCOM від Microsoft, а також Java Beans [264]. Даний тип об'єктних структур розглядався в розділі 11 при обговоренні архітектури розподілених об'єктів.
3. *Структури інструментальних середовищ розробки додатків.* Пов'язані з окремими прикладними областями, такими як телекомунікації або фінанси [30]. Вони вбудовуються в систему знань області додатку і підтримують розробку додатків кінцевого користувача. Дані структури пов'язані з сімействами додатків, які розглядаються в розділі 14.2.

Об'єктна структура – це узагальнена структура, яку можна деталізувати і розширити при створенні конкретної підсистеми або додатку. Деталізація об'єктної структури зазвичай припускає додавання конкретних класів, що успадковують методи від абстрактних класів об'єктної структури. Крім того, визначаються методи, які викликаються у відповідь на події, визначені об'єктною структурою.

На момент написання книги були досить добре розроблені інфраструктури систем, особливо ті з них, які пов'язані з графічними інтерфейсами користувача. Їх поступово витісняють структури інструментальних середовищ розробки додатків [77]. Спробуємо розібрати найбільш ефективні уявлення і організацію даних об'єктних структур.

Одній з найвідоміших і поширеніших об'єктних структур для графічних інтерфейсів користувача (GUI) є об'єктна структура "модель-представление-контроллер" (мал. 14.6). Ця модель з'явилася в 1980-х роках як метод проектування графічних інтерфейсів користувача, який підтримує різні представлення об'єкту і розрізняє взаємодії з кожним з цих уявлень.




*Мал. 14.6. Об'єктна структура "модель-представление-контроллер"*

Об'єктні структури часто реалізуються у вигляді патернів (див. розділ 14.2). Наприклад, об'єктна структура "модель-представление-контроллер" включена в патерн Оглядач, описаний у врізанні 14.1, а також в ряд інших патернів, детально розглянутих в роботі [124].

Основні недоліки об'єктних структур – їх складність і час, необхідне для того, щоб навчитися працювати з ними. Для повного вивчення об'єктних структур може знадобитися декілька місяців. Саме тому у великих організаціях деякі розробники ПО спеціалізуються по об'єктних структурах. Немає ніяких сумнівів в ефективності даного підходу до повторного використання, проте високі витрати на вивчення об'єктних структур обмежують його повсюдне розповсюдження.

### 14.1.2. Повторне використання комерційних програмних продуктів

Термін "комерційні програмні продукти" можна застосувати до будь-якого компоненту, створеного незалежним виробником. Разом з тим під цим терміном часто мається на увазі програмне забезпечення системного рівня. Я також вважаю за краще говорити про комерційні системи. Функціональність, пропонована цими системами, набагато ширше за функціональність більш спеціалізованих компонентів, і тому збільшується потенційний виграш, отриманий від повторного використання.

Деякі типи комерційних систем використовуються повторно впродовж багатьох років. Кращим прикладом тому, мабуть, служать бази даних. Дуже небагато розробників створюють власні системи управління базами даних. Але до недавнього часу існувало лише декілька великих комерційних систем, таких як системи управління базами даних і монітори телеобробки, використовувані повторно.

Нові підходи до проектування систем, що надають програмам доступ до системних функцій, показують, що створення великих систем (наприклад, систем електронної комерції) за допомогою інтеграції ряду комерційних систем сьогодні розглядається як один з прийнятних варіантів проектування. Завдяки функціональності, пропонованій цими системами, скорочення фінансових і тимчасових витрат може досягти величини, порівнянної з розробкою нового ПО "с нуля". Більш того, зменшуються ризики, оскільки комерційні системи вже існують і розробники можуть побачити, чи задовольняють вони вимогам, що пред'являються до них.

В принципі використання крупномодульных комерційних систем не відрізняється від використання будь-якого іншого більш спеціалізованого компоненту. Для цього необхідно вивчити інтерфейси системи і використовувати їх для організації взаємодії з іншими системними компонентами, також необхідно розробити системну архітектуру, яка підтримувала б комерційні системи при спільній роботі.

Проте той факт, що комерційними програмними продуктами є крупні системи і часто продаються як окремі автономні системи, вносить додаткові проблеми. При інтеграції таких систем можуть виникнути, як мінімум, чотири проблеми [42].

1. *Недостатній контроль над функціональністю і продуктивністю комерційних продуктів.* Хоча вважається, що їх інтерфейси відомі, не виключена вірогідність наявності прихованих операцій, які "перетинатимуться" з системними операціями. Вирішення цієї проблеми може стати пріоритетом для системних розробників, що використовують комерційні продукти, причому ця проблема, очевидно, не пов'язана з виробником даного продукту.
2. *Проблеми, пов'язані з організацією, взаємодії комерційних систем.* Іноді складно підібрати комерційні продукти для спільної роботи, оскільки кожен продукт розробляється на основі різних припущень з приводу його використання. У [127] приведені результати експерименту по інтеграції чотирьох різних комерційних продуктів, при цьому було встановлено, що робота трьох продуктів грунтувалася на одних і тих же подіях, проте всі вони використовували різні моделі подій і кожен з них вважав, що має першочерговий доступ до черги подій. Як наслідок, на розробку системи було витрачено в п'ять разів більше зусиль, чим передбачалося, а на складання тимчасового графіка роботи системи пішли два роки замість шести місяців, що передбачалися.
3. *Відсутність контролю за модифікацією комерційних продуктів.* Виробники комерційних продуктів приймають вирішення по зміні своїх систем під тиском ринку. Зокрема, нові версії програмних продуктів, розроблених для персональних комп'ютерів, створюються дуже часто і можуть виявитися не сумісними з попередніми версіями. Нові версії можуть володіти додатковою функціональністю, непідтримуваною попередніми версіями.
4. *Підтримка виробниками комерційних продуктів.* Рівень підтримки, що надається виробниками комерційних продуктів, варіюється в широких межах, оскільки ці системи розповсюджуються вільно. Підтримка виробників особливо важлива в тих *випадках,* коли у розробників виникають проблеми, пов'язані з діставанням доступу до початкового коду і до докладної документації системи. Не дивлячись на те що виробник бере на себе зобов'язання по підтримці своїх систем, зміна ситуації на ринку і економічних умов може привести до того, що йому почне важко продовжувати виконання узятих зобов'язань. Наприклад, виробник комерційної системи вирішив більше не підтримувати розвиток якого-небудь продукту із-за обмеженого попиту або, можливо, передав його іншій компанії, яка не хоче підтримувати всі його продукти.

Звичайно, маловірогідно, що всі ці проблеми виникнуть в кожному випадку використання комерційних продуктів. По моїх приблизних підрахунках, в багатьох програмних проектах, інтегруючих комерційні системи, може з'явитися принаймні одна з перерахованих проблем. Відповідно переваги у вартості і часі виконання робіт по використанню комерційних продуктів опиняться менше, ніж передбачалося в первинному оптимістичному варіанті.

Всі перераховані проблеми є проблемами життєвого циклу ПО і впливають тільки на початкову розробку системи. Але у багатьох випадках при використанні комерційних продуктів витрати на супровід і модернізацію систем також можуть зрости [42], оскільки люди, що беруть участь в обслуговуванні системи, з часом все більше віддаляються від розробників початкової системи.

Не дивлячись на всі ці проблеми, переваги, що отримуються при використанні комерційних продуктів, вельми істотні, оскільки в цьому випадку можна заощадити місяці, а іноді і роки на розробці системи. Оскільки швидке створення систем є одним з ключових чинників для більшості програмних проектів, даний вид повторного використання компонентів, ймовірно, отримає з часом широке практичне застосування.

### 14.1.3. Розробка повторно використовуваних компонентів

Розробка ідеального компоненту для повторного використання повинна бути процесом (заснованим на досвіді і знаннях про проблеми повторного використання) створення узагальнених компонентів, які можна адаптувати для різних варіантів їх використання.

Програмний компонент, призначений для повторного використання, має ряд особливостей.

1. Повинен відображати стабільні абстракції наочної області, тобто фундаментальні поняття області додатки, які міняються поволі. Наприклад, в банківській системі абстракціями наочної області можуть бути рахунки, форма вкладника, бюлетені і тому подібне
2. Повинен приховувати спосіб представлення свого стану і надавати операції, які дозволяють оновлювати стани і діставати до нього доступ. Наприклад, в компоненті, який представляє рахунок в банці, повинні бути операції, що дозволяють виконати запити по залишках на рахівницях, по змінах в залишках рахунку, записати операції (транзакції) на рахівницях і тому подібне
3. Повинен бути максимально незалежним. У ідеалі компонент повинен бути настільки автономним, щоб не потребувати інших компонентів. Насправді таке здійсниме тільки для зовсім простих компонентів, складніші завжди залежать від інших компонентів. Краще всього наявні залежності звести до мінімуму, особливо якщо вони пов'язані з такими компонентами, як змінні функції операційної системи.
4. Всі виняткові ситуації повинні бути частиною інтерфейсу компоненту. Компоненти не винні самі обробляти виключення, оскільки в різних застосуваннях існують різні вимоги для обробки виняткових ситуацій. Краще визначити ті виключення, які необхідно обробляти, і оголосити їх як частина інтерфейсу компоненту. Наприклад, простий компонент, що реалізовує структуру даних стека, повинен визначати і оголосити виключеннями переповнювання і обнулення стека.

У більшості існуючих систем є великі сегменти коди, які реалізують абстракції наочної області, проте їх не можна безпосередньо використовувати як компоненти. Причина в невідповідності програмної коди моделі, показаної на мал. 14.2, чітко певному інтерфейсу запитів і постачальників сервісів. Щоб повторно використовувати такі компоненти, як правило, необхідно побудувати пакувальник (програмний засіб для створення оболонки і стандартизації зовнішніх звернень). Пакувальник приховує початковий код і надає інтерфейс для зовнішніх компонентів, що відкриває доступ до сервісів, що надаються.

При створенні компонентів, призначених для повторного використання, передбачається надання дуже загального інтерфейсу з операціями, які забезпечують різні способи використання компонентів. Щоб зробити компоненти практичними у використанні, потрібний мінімальний інтерфейс, простій для розуміння. З іншого боку, передбачувана можливість повторного використання ускладнює компоненти і тому зменшує їх зрозумілість. Тому розробники компонентів, призначених для повторного використання, повинні прийти до деякого компромісу між узагальненістю і зрозумілістю компонентів.

## 14.2. Сімейства додатків

Один з найбільш ефективних підходів до повторного використання базується на понятті сімейства додатків. Сімейство додатків, або серія програмних продуктів, – це набір додатків, що мають загальну архітектуру, що відображає специфіку конкретної наочної області (див. розділ 10). Разом з тим всі додатки однієї серії різні. Кожного разу при створенні нового застосування повторно використовується загальне ядро сімейства додатків. Далі в процесі розробки створюється декілька додаткових компонентів, а деякі компоненти адаптуються згідно новим вимогам.

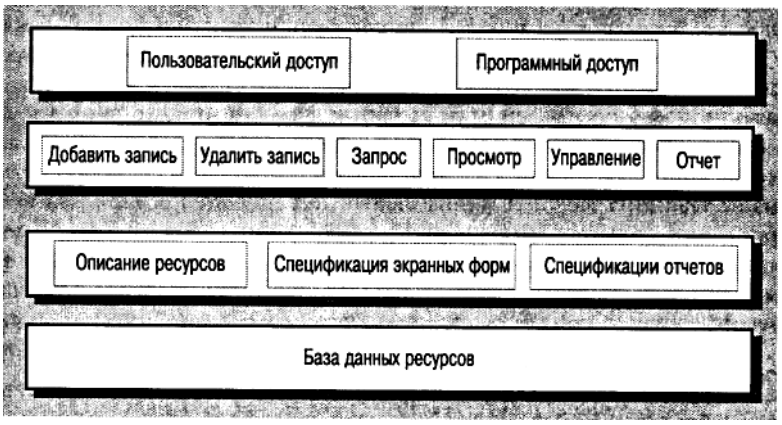
Існують різні спеціалізації сімейств додатків, приведемо деякі з них.

1. *Платформена спеціалізація,* при якій для різних платформ розробляються свої версії додатку. Наприклад, додаток може мати версії для платформ Windows NT, Solaris або Linux. В даному випадку функціональність додатку зазвичай не міняється; піддаються

змінам тільки ті компоненти, які відповідають за взаємодію з апаратними засобами і операційною системою.

2. *Конфігураційна спеціалізація,* при якій різні версії додатку створюються для управління різними периферійними пристроями. Наприклад, різні версії системи безпеки можуть залежати від типу використовуваної радіосистеми. В цьому випадку змінюється функціональність додатку для того, щоб відповідати периферійним пристроям, і необхідно змінити ті компоненти, які пов'язані з периферійними пристроями.
3. *Функціональна спеціалізація,* при якій створюються різні версії додатку для замовників з різними вимогами. Наприклад, система автоматизації бібліотек може мати декілька модифікацій залежно від того, де вона застосовується – в публічній, довідковій або університетській бібліотеці. В цьому випадку змінюються компоненти, що реалізовують функціональність системи, і додаються нові компоненти.

Щоб наочно представити цю технологію повторного використання, розглянемо архітектуру системи управління ресурсами, зображену на мал. 14.7. Подібні системи використовуються в організаціях для відстежування активів (ресурсів, запасів) і управління ними. Наприклад, система управління ресурсами енергосистеми повинна відстежувати всі стаціонарні енергооб'єкти і відповідне устаткування. У університетах система управління ресурсами може відстежувати устаткування, використовуване в учбових лабораторіях.




*Мал. 14.7. Узагальнена система управління ресурсами*

Очевидно, системи управління ресурсами відрізнятимуться один від одного залежно від типу ресурсів і від інформації, необхідної для управління ними. Наприклад, в додатку для енергосистеми не потрібні засоби, що дозволяють змінювати розміщення ресурсів, оскільки всі об'єкти енергосистеми стаціонарні. Проте система обліку ресурсів для університету повинна мати таку можливість, оскільки устаткування може переходити з однієї лабораторії в іншу.

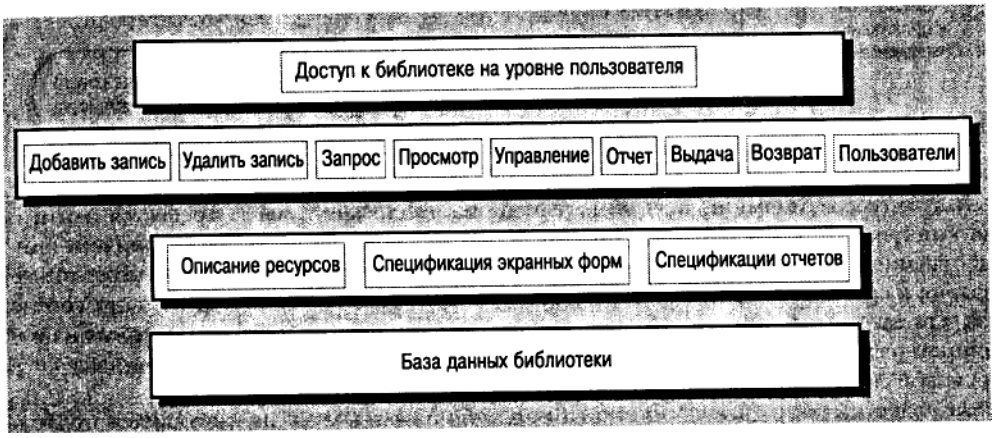
Проте всі ці системи повинні надавати основні засоби для управління ресурсами: можливість додавання і видалення ресурсів, формування запитів, проглядання бази даних ресурсів і формування звітів. Отже, архітектура систем управління ресурсами буде однакова для цілого сімейства додатків, в якому окремі застосування підтримують різні типи ресурсів.

Для того, щоб повторне використання систем було ефективним, на етапі створення архітектури необхідно відокремити основні засоби системи від конкретної інформації про керовані ресурси і від доступу користувачів до цієї інформації. На мал. 14.7 розділення досягнуте завдяки багаторівневій архітектурі, в якій на одному рівні вбудовані описи ресурсів, формування екранних форм і звітів. Верхні рівні системи використовують ці описи в своїх методах і не містять конкретної інформації про ресурси. За допомогою зміни рівня описів можна створювати різні додатки управління ресурсами.

Звичайно, такий тип систем можна виконати у вигляді об'єктно-орієнтованих, визначивши спочатку об'єкт абстрактного ресурсу, а потім за допомогою спадкоємства – об'єкт, залежний від типу керованого ресурсу. У результаті така архітектура буде небагато чим відрізнятися

від архітектури, зображеної на мал. 14.7. Проте для систем даного типу об'єктно-орієнтований підхід не годиться. Коли додатки розраховані на великі бази даних, що містять мільйони записів, але відносна мала кількість типів логічних модулів (відповідних об'єктам і суті), очевидно, що об'єктно-орієнтована система працює менш ефективно, чим системи з реляційними базами даних. На момент написання книги комерційні об'єктно-орієнтовані бази все ще залишаються відносно повільними і не здатними підтримувати сотні транзакцій в секунду.

Подібно до того як за допомогою описів нових ресурсів можна створювати нові члени сімейства додатків, за допомогою включення нових модулів на системному рівні в систему можна додати нові функціональні можливості. Для створення бібліотечної системи (мал. 14.8) я адаптував систему управління ресурсами, показану на мал. 14.7. В результаті в систему додані нові можливості для видачі і повернення ресурсів і реєстрації користувачів системою. На мал. 14.8 цих засобів розташовані справа. Оскільки програмний доступ тут не потрібний, самий верхній рівень системи підтримує доступ до ресурсів тільки на рівні користувача.



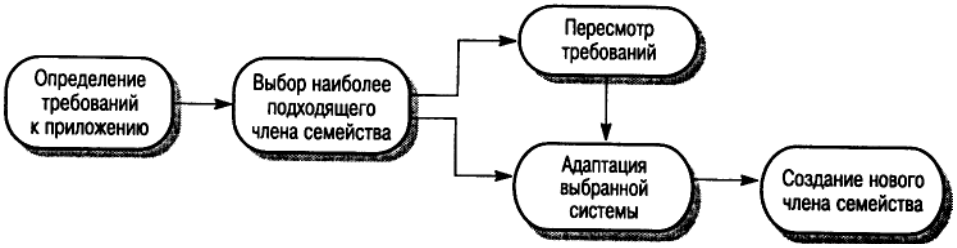

*Мал. 14.8. Бібліотечна система*

В цілому адаптація версії додатку в процесі його розробки приводить до того, що значна частина коди додатку використовується повторно. Більш того, накопичений досвід часто можна використовувати для розробки інших систем, тому об'єднання розробників ПО в окрему групу скорочує процес їх навчання. Оскільки тести для більшості компонентів додатку також можна використовувати повторно, то повний час, необхідне на розробку додатку, значно зменшується.

Процес адаптації сімейства додатків для створення нового застосування складається з декількох етапів, представлених на мал. 14.9. Деталі процесу можуть значно відрізнятися для різних прикладних областей і для різних організацій. Узагальнений процес створення нового застосування складається з наступних етапів.

1. *Визначення вимог для нового застосування.* Даний етап – звичайний процес розробки вимог. Але оскільки система вже існує, природно провести експериментування з нею і виявити ті системні вимоги, які необхідно змінити.
2. *Вибір найбільш відповідного члена сімейства додатків.* Виконується аналіз вимог, після чого вибирається найбільш відповідний член сімейства, що вимагає внесення мінімальних змін.
3. *Перегляд вимог.* Як правило, з'являється додаткова інформація, що вимагає внесення змін до існуючої системи, тому перегляд вимог на цьому етапі дозволяє зменшити кількість необхідних змін.
4. *Адаптація вибраної системи. Для* системи розробляються нові модулі, а ті, що існують адаптуються до нових вимог.
5. *Створення нового члена сімейства додатків. Для* замовника створюється новий член сімейства додатків. На цьому етапі виконується документування ключових особливо-

стей системи, щоб надалі її можна було використовувати як основу для розробки інших систем.




*Мал. 14.9. Процес розробки нового члена сімейства додатків*

В процесі створення нового члена сімейства додатків часто потрібно знайти якийсь компроміс між якнайповнішим використанням існуючих застосувань і виконанням конкретних вимог для нового застосування. Чим детальніші вимоги до системи, тим менше вірогідність, що наявні компоненти відповідатимуть цим вимогам. Але практично завжди можна досягти певного компромісу і обмежити об'єм змін, що вносяться до системи, тоді процес створення нової системи виконується швидко і з невеликими витратами.

За рідкісним виключенням, сімейства додатків створюються з наявних застосувань, тобто організація створює додатки і потім при необхідності використовує їх як основу для розробки нового застосування. Але разом з тим внесення змін порушує структуру додатку, тому рано чи пізно ухвалюється рішення про створення сімейства узагальнених застосувань. В цьому випадку активно використовуються знання, зібрані в процесі створення початкової групи додатків.

## 14.3. Проектні патерни

Спроби повторно використовувати компоненти, що діють, постійно обмежуються конкретними рішеннями, прийнятими системними розробниками. Рішення можуть відноситися до окремих алгоритмів, використовуваних при реалізації компонентів, до об'єктів і до типів даних в інтерфейсах компонентів. Якщо вирішення противоречат конкретним вимогам до компонентів, то повторне використання компонентів або неможливо, або робить систему неефективної.

Один із способів вирішення даної проблеми – повторне використання абстрактніших структур, що не містять деталей реалізації. Такі структури розробляються спеціально для того, щоб відповідати певному застосуванню. Перші реалізації цього підходу привели до документування і публікації фундаментальних алгоритмів [201], а потім до документування абстрактних типів даних, таких як стеки, дерева і списки [53]. Зовсім недавно такий спосіб повторного використання узагальнений в понятті патерну.

Проектні патерни (design patterns) [124] з'явилися з ідей, висунутих Крістофером Александером (Alexander [8]), який запропонував зручні і ефективні узагальнені патерни розробки конкретних проектів. Патерн – це опис проблеми і методу її рішення, що дозволяє надалі використовувати це рішення в різних умовах. Патерн не є детальною специфікацією. Швидше, він є описом, в якому закумульовані знання і досвід. Патерн – гарантоване вирішення загальної проблеми.

При створенні ПО проектні патерни завжди пов'язані з об'єктно-орієнтованим проектуванням. Щоб забезпечити загальність, патерни часто використовують такі об'єктно-орієнтовані поняття, як спадкоємство і поліморфізм. Проте загальний принцип використання патернів однаково застосовний при будь-якому підході до проектування ПО.

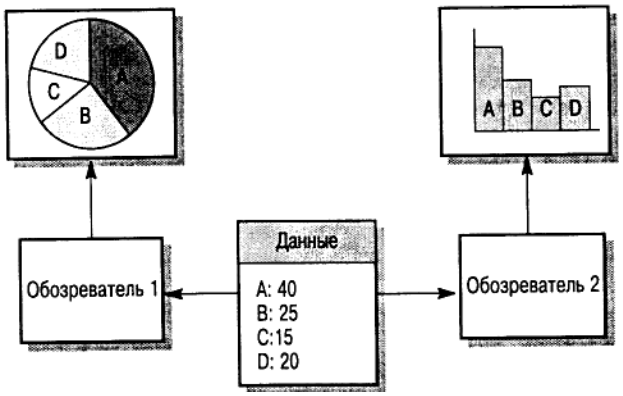
У [124] визначено чотири основні елементи проектного патерну.

1. Змістовне ім'я, яке є посиланням на патерн.

2. Опис проблемної області з перерахуванням всіх ситуацій, в яких можна використовувати патерн.
3. Опис рішень з окремим описом різних частин рішення і їх взаємин. Це не опис конкретного проекту, *а шаблон* проектних рішень, який можна використовувати різними способами. У описі рішень часто використовуються графічні уявлення, які показують взаємини між об'єктами і класами об'єктів в даному рішенні.
4. Опис "вихідних" результатів – це опис результатів і компромісів, необхідних для застосування патерну. Зазвичай використовується для того, щоб допомогти розробникам оцінити конкретну ситуацію і вибрати для неї найбільш відповідний патерн.

Часто в опис патерну вводяться також розділи *мотивації* (обгрунтування корисності патерну) і *застосовності* (опис ситуацій, в яких можна використовувати патерн).

Як приклад розглянемо один з найчастіше використовуваних патернів, запропонованих в роботі [124], а саме Оглядач (Observer) (див. врізання 14.1). Даний патерн використовується тоді, коли необхідні різні представлення стану об'єкту. Він виділяє потрібний об'єкт і представляє його в різних формах; на мал. 14.10 показано два графічні представлення одного і того ж набору даних.




*Мал. 14.10. Різні представлення даних*

---

**Врізання 14.1. Опис патерну Оглядач**

**Ім'я паттера.** Оглядач

**Опис**. Відокремлює відображення стану об'єкту від самого і пропонує різні способи представлення стану. При зміні стану об'єкту всі уявлення автоматично оновлюються, щоб відобразити зміни, що відбулися.

**Опис проблеми**. У багатьох ситуаціях потрібно уявити інформацію про стан деякого об'єкту декількома різними способами, наприклад використовуючи графічне і табличне уявлення. Всі уявлення взаємозв'язані і повинні оновлюватися при зміні станів.

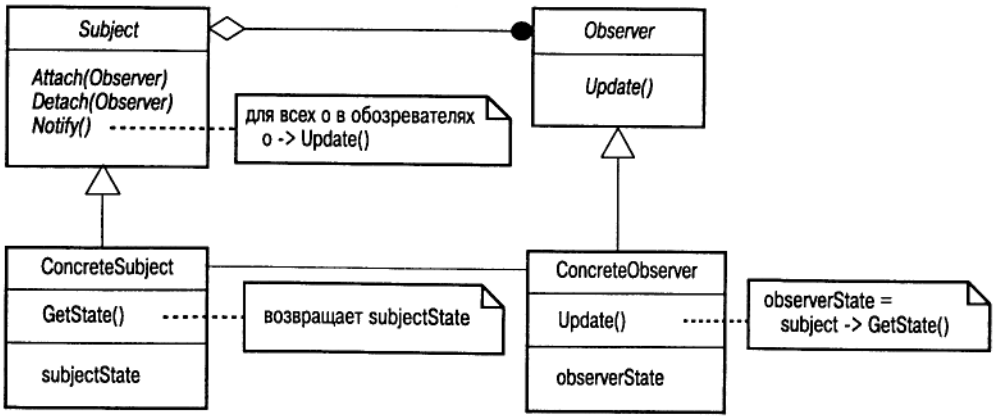
Даний патерн можна використовувати у всіх ситуаціях, де потрібно декілька різних уявлень інформації про станів об'єкту і де немає необхідності знати формати представлення даних про стан об'єкту.

**Опис рішення**. Структура патерну показана на мал. 14.11. У нім визначені два абстрактних об'єктам Subject (Дані) і Observer (Оглядач), а також два конкретні об'єкти: ConcreteSubject (Конкретні дані) і ConcreteObserver (Конкретний оглядач) які успадковують властивості відповідних абстрактних об'єктів. Стан, що відображається, підтримується об'єктом ConcreteSubject, який також успадковує методи від Subject, що дозволяють йому додавати і видаляти об'єкти Observer (методи Attach і Detach) і видавати сповіщення при зміні стану (метод Notify).

Об'єкт ConcreteObserver обробляє копію стану ConcreteSubject (копію subjectState, отриману за допомогою методу GetState (Отримати багатство)) і реалізує метод Update (Відновити) інтерфейсу Observer, який дозволяє зберігати копії стану. ConcreteObserver автоматично відображає цей стан.

**Результати**. Для оптимізації оглядача необхідна додаткова інформація про об'єкти. Зміни у форматі даних, що відображаються, викличуть серію зв'язаних змін в створених оглядачах.

Зазвичай в патернах класи об'єктів і взаємовідношення між ними зображаються за допомогою спеціальних графічних нотацій. На мал. 14.11 представлений патерн Оглядач в нотації мови UML.




*Мал. 14.11. Патерн Оглядач*

Застосування патернів є вельми ефективним способом повторного використання; проте, на мою думку, даний метод вимагає значних витрат на освоєння і може ефективно використовуватися в проектуванні систем тільки досвідченими програмістами. Причина криється в достатньо високій складності патернів. Патерн не схожий на виконуваний компонент, для використання якого досить вивчити тільки його інтерфейс. Очевидно, що на вивчення патерну потрібний певний час.

Використовувати патерни можуть тільки досвідчені програмісти, оскільки лише вони здатні розпізнати загальні ситуації, в яких можна застосувати той або інший патерн. Недосвідчені програмісти, навіть якщо вони прочитали декілька книг, що описують патерни, на практиці часто не можуть визначити, де слід використовувати патерн, а де необхідне нестандартне спеціальне рішення.

## КЛЮЧОВІ ПОНЯТТЯ

- Проектування з повторним використанням компонентів означає проектування програмних систем з урахуванням вже наявних компонентів ПО.
- До переваг повторного використання програмного забезпечення можна віднести нижчі витрати, швидшу розробку і знижені ризики. Також підвищується надійність систем і з'являється можливість ефективніше застосовувати досвід і знання фахівців, привертаючи їх до проектування повторно використовуваних компонентів.
- Покомпонентна розробка ПО грунтується на використанні компонентів з чітко певними інтерфейсами запитів і постачальників сервісів без конкретизації знань про внутрішній устрій компонентів. Можна повторно використовувати різні типи компонентів: функції, абстракції даних, структури і закінчені системи додатків.
- Під повторним використанням комерційних продуктів розуміється повторне використання крупномодульных готових програмних систем. Застосування комерційних продуктів може значно зменшити витрати і час на розробку нового ПО.
- Програмні компоненти, що створюються для повторного використання, повинні бути незалежними, відображати абстракції наочної області, надавати доступ до стану через методи інтерфейсу і не винні самі обробляти виняткові ситуації.
- Сімейство додатків – це група додатків, які розробляються на основі одного або декількох базових застосувань. Базова узагальнена система адаптується для відповідності вимогам системи, що розробляється.

- Проектні патерни – це абстракції високого рівня, які документують успішні проектні рішення. Вони є основою при повторному використанні проектних рішень в об'єктно-орієнтованих розробках. Опис патерну містить ім'я патерну, опис проблеми і рішення, а також результати і компроміси використання шаблону.

### Вправи

**14.1.** Які основні технічні і нетехнічні чинники, що утрудняють повторне використання програмного забезпечення?

**14.2.** Поясните, чому скорочення витрат при повторному використанні компонентів не прямо пропорціонально розмірам повторно використовуваних компонентів.

**14.3.** Приведіть чотири аргументи проти повторного використання компонентів.

**14.4.** Припустите можливі інтерфейси запитів і постачальників сервісів для наступних компонентів.
- Компонент, що реалізовує рахунок в банці.
- Компонент, що реалізовує не залежну від мови клавіатуру. Клавіатури в різних країнах мають різну організацію клавіш і різні набори символів.
- Компонент, що реалізовує засіб управління версіями, розглянуте в розділі 29.

**14.5.** Чим відрізняється повторне використання об'єктної структури додатку від повторного використання комерційних продуктів? Чому іноді простіше повторно використовувати комерційний продукт, чим об'єктну структуру додатку?

**14.6.** На прикладі метеорологічної станції, описаної в розділі 12, запропонуєте архітектуру сімейства додатків, які пов'язані з видаленим спостереженням і збором метеоданных.

**14.7.** На прикладі сімейства додатків по управлінню ресурсами (див. мал. 14.7) подумайте, які методи необхідно додати або змінити, щоб можна було робити повторне замовлення на окремі види ресурсів, якщо їх кількість стає менше деякої заданої величини.

**14.8.** Чому патерни – ефективний спосіб повторного використання в проектуванні? Які недоліки цього підходу?

**14.9.** Повторне використання збільшує кількість питань про власність, що охороняється авторським і інтелектуальним правом. Якщо замовник оплачує розробникові ПО замовлення на розробку якої-небудь системи, хто має право повторно використовувати створений код? Чи має право розробник використовувати цей код як основу для базового компоненту? Якими повинні бути механізми оплати праці розробника повторно використовуваних компонентів? Обговорите ці і інші етичні питання, пов'язані з повторним використанням програмного забезпечення.

## 15. Проектування інтерфейсу користувача

### Цілі

Мета справжнього розділу – познайомити з основними аспектами проектування інтерфейсу користувача, які повинні знати розробники ПО. Прочитавши цей розділ, ви винні:

- ❑ знати основні принципи проектування інтерфейсу користувача;
- ❑ освоїти п'ять різних стилів взаємодії користувача з програмними системами;
- ❑ знати різні стилі представлення інформації і те, в яких випадках доцільне графічне представлення даних;
- ❑ познайомитися з основними правилами проектування засобів підтримки користувача, вбудованих в програмне забезпечення;
- ❑ мати уявлення про основні показники зручності використання систем.

Проектування обчислювальних систем охоплює широкий спектр проектних дій – від проектування апаратних засобів до проектування інтерфейсу користувача. Організації-розробники часто наймають фахівців для проектування апаратних засобів і дуже рідко для проектування інтерфейсів. Таким чином, фахівцям з розробки ПО часто доводиться проектувати і інтерфейс користувача. Якщо у великих компаніях до цього процесу залучаються фахівці з інженерної психології, то в невеликих компаніях послугами таких фахівців практично не користуються.

Грамотно спроектований інтерфейс користувача украй важливий для успішної роботи системи. Складний в застосуванні інтерфейс, як мінімум, приводить до помилок користувача. Іноді вони просто відмовляються працювати з програмною системою, не дивлячись на її функціональні можливості. Якщо інформація представляється плутано або непослідовно, користувачі можуть зрозуміти її неправильно, внаслідок чого їх подальші дії можуть привести до пошкодження даних або навіть до збою в роботі системи.

У 1982 році, під час виходу першої редакції цієї книги, стандартним пристроєм взаємодії між користувачем і програмою був "беззвучний" буквено-цифровий (текстовий) термінал, що відображає на чорному полі символи зеленого або синього кольору. У той час інтерфейси користувача були текстовими або створювалися у вигляді спеціальних форм. Зараз майже всі користувачі працюють на персональних комп'ютерах. Всі сучасні персональні комп'ютери підтримують графічний інтерфейс користувача (graphical user interface – GUI), який має на увазі використання кольорового графічного екрану з високим дозволом і дозволяє працювати з мишею і з клавіатурою.

Хоча текстові інтерфейси ще достатньо широко застосовуються, особливо в успадкованих системах, у наш час користувачі вважають за краще працювати з графічним інтерфейсом. У табл. 15.1 перераховані основні елементи GUI.

**Таблиця 15.1. Елементи графічних інтерфейсів користувача**

| Елементи | Опис |
|---|---|
| Вікна | Дозволяють відображати на екрані інформацію різного роду |
| Піктограми | Представляють різні типи даних. У одних системах піктограми представляють файли, в інших – процеси |
| Меню | Введення команд замінюється вибором команд з меню |
| Покажчики | Миша використовується як пристрій вказівки для вибору команд з меню і для виділення окремих елементів у вікні |
| Графічні елементи | Можуть використовуватися спільно з текстовими |

Графічні інтерфейси володіють рядом переваг.

1. Їх відносно просто вивчити і використовувати. Користувачі, що не мають досвіду роботи з комп'ютером, можуть легко і швидко навчитися працювати з графічним інтерфейсом.
2. Кожна програма виконується в своєму вікні (екрані). Можна перемикатися з однієї програми в іншу, не втрачаючи при цьому дані, отримані в ході виконання програм.
3. Режим повноекранного відображення вікон дає можливість прямого доступу до будь-якого місця екрану.

Мета даного розділу - привернути увагу розробників ПО до деяких ключових проблем, лежачих в основі проектування інтерфейсів користувача. Розробники і програмісти зазвичай компетентні у використанні таких технологій, як класи Swing в мові Java [103] або HTML [249], реалізації інтерфейсів користувача, що є основою. Проте цю технологію далеко не завжди застосовують належним чином, внаслідок чого інтерфейси користувача виходять неелегантними, незручними і складними у використанні.

У цьому розділі я приведу декілька рекомендацій по проектуванню засобів кінцевого користувача, не розглядаючи весь процес проектування цих засобів. Із-за браку місця розглядаються тільки графічні інтерфейси. Спеціальні інтерфейси, наприклад для мобільних телефонів, телевізійних приймачів, копіювальної техніки або факсимільних апаратів, розглядати-

ся не будуть. Тут я зроблю тільки коротке введення в тему проектування інтерфейсів користувача. Додаткову інформацію по даній темі можна знайти в книгах [316, 99, 281].

На мал. 15.1 зображений ітераційний процес проектування призначеного для користувача інтерфейсу. Як наголошувалося в розділі 8, найбільш ефективним підходом до проектування інтерфейсу користувача є розробка із застосуванням моделювання призначених для користувача функцій. На початку процесу прототипирования створюються паперові макети інтерфейсу, потім розробляються екранні форми, що моделюють взаємодію з користувачем. Бажано, щоб кінцеві користувачі брали активну участь в процесі проектування інтерфейсу [258]. У одних випадках користувачі допоможуть оцінити інтерфейс; у інших будуть повноправними членами проектної групи [207, 138].


*Мал. 15,1. Процес проектування інтерфейсу користувача*

Важливим етапом процесу проектування інтерфейсу користувача є аналіз діяльності користувачів, яку повинна забезпечити обчислювальна система. Не вивчивши того, що, з погляду користувача, повинна робити система, неможливо сформувати реалістичний погляд на проектування ефективного інтерфейсу. Для аналізу потрібно (як правило, одночасно) застосовувати різні методики, а саме: аналіз завдань [94], етнографічний підхід (див. розділ 6) [328, 167], опити користувачів і спостереження за їх роботою.

## 15.1. Принципи проектування інтерфейсів користувача

Розробники інтерфейсів завжди повинні враховувати фізичні і розумові здібності людей, які працюватимуть з програмним забезпеченням. Люди на короткий час можуть запам'ятати вельми обмежений об'єм інформації (див. розділ 22) і здійснюють помилки, якщо доводиться вводити уручну великі об'єми даних або працювати в напружених умовах. Фізичні можливості людей можуть істотно розрізнятися, тому при проектуванні інтерфейсів користувача необхідно постійно пам'ятати про це.

Основою принципів проектування інтерфейсів користувача є людські можливості. У табл. 15.2 представлені основні принципи, застосовні при проектуванні будь-яких інтерфейсів користувача. Детальніший перелік "керівних" правил проектування інтерфейсів можна знайти в книзі [316].

**Таблиця 15.2. Принципи проектування інтерфейсів користувача**

| Принцип | Опис |
|---|---|
| Облік знань користувача | У інтерфейсі необхідно використовувати терміни і поняття, узяті з досвіду ма<br>бутніх користувачів системи |
| Узгодженість | Інтерфейс повинен бути узгодженим в тому сенсі, що однотипні (але різні) оп<br>рації повинні виконуватися одним і тим же способом |

| | |
|---|---|
| Мінімум несподіванок | Поведінка системи повинна бути прогнозованою |
| Здатність до відновлення | Інтерфейс повинен мати засоби, що дозволяють користувачам відновити да після помилкових дій |
| Керівництво користувача | Інтерфейс повинен надавати необхідну інформацію у разі помилок користува і підтримувати засоби контекстно-залежної довідки |
| Облік різнорідності користувачів | У інтерфейсі повинні бути засоби для зручної взаємодії з користувачами, м мають різний рівень кваліфікації і різні можливості |

Принцип обліку знань користувача припускає наступне: інтерфейс повинен бути настільки зручний при реалізації, щоб користувачам не знадобилося особливих зусиль, щоб звикнути до нього. У інтерфейсі повинні використовуватися терміни, зрозумілі користувачеві, а об'єкти, керовані системою, повинні бути безпосередньо пов'язані з робочим середовищем користувача. Наприклад, якщо розробляється система, призначена для авіадиспетчерів, то керованими об'єктами в ній повинні бути літаки, траєкторії польотів, сигнальні знаки і тому подібне Основну реалізацію інтерфейсу в термінах файлових структур і структур даних необхідно приховати від кінцевого користувача.

Принцип узгодженості інтерфейсу користувача припускає, що команди і меню системи повинні бути одного формату, параметри повинні передаватися у всі команди однаково і пунктуація команд повинна бути схожою. Такі інтерфейси скорочують час на навчання користувачів. Знання, отримані при вивченні якої-небудь команди або частини додатку, можна потім застосувати при роботі з іншими частинами системи.

В даному випадку мова йде про узгодженості низького рівня. І творці інтерфейсу завжди повинні прагнути до нього. Проте бажана узгодженість і більш високого рівня. Наприклад, зручно, коли для всіх типів об'єктів системи підтримуються однакові методи (такі, як друк, копіювання і тому подібне). Проте повна узгодженість неможлива і навіть небажана [140]. Наприклад, операцію видалення об'єктів робочого столу доцільно реалізувати за допомогою їх перетягання в корзину. Але в текстовому редакторові такий спосіб видалення фрагментів тексту здається неприродним.

Завжди потрібно дотримувати наступний принцип: кількість несподіванок повинна бути мінімальною, оскільки користувачів дратує, коли система раптом починає поводитися непередбачувано. При роботі з системою у користувачів формується певна модель її функціонування. Якщо його дія в одній ситуації викликає певну реакцію системи, природно чекати, що таке ж дія в іншій ситуації приведе до аналогічної реакції. Якщо ж відбувається зовсім не те, що очікувалося, користувач або дивується, або не знає, що робити. Тому розробники інтерфейсів повинні гарантувати, що схожі дії справлять схоже враження.

Дуже важливий принцип відновлюваності системи, оскільки користувачі завжди допускають помилки. Правильно спроектований інтерфейс може зменшити кількість помилок користувача (наприклад, використання меню дозволяє уникнути помилок, які виникають при введенні команд з клавіатури), проте всі помилки усунути неможливо. У інтерфейсах повинні бути засоби, що по можливості запобігають помилкам користувача, а також що дозволяють коректно відновити інформацію після помилок. Ці засоби бувають двох видів.

1. *Підтвердження деструктивних дій.* Якщо користувач вибрав потенційно деструктивну операцію, то він повинен ще раз підтвердити свій намір.
2. *Можливість відміни дій.* Відміна дії повертає систему в той стан, в якому вона знаходилася до їх виконання. Не зайвою буде підтримка багаторівневої відміни дій, оскільки користувачі не завжди відразу розуміють, що зробили помилку.

Наступний принцип – підтримка користувача. Засоби підтримки користувачів повинні бути вбудовані в інтерфейс і систему і забезпечувати різні рівні допомоги і довідкової інформації. Повинне бути декілька рівнів довідкової інформації – від основ для початкуючих до

повного опису можливостей системи. Довідкова система повинна бути структурованою і не перенавантажувати користувача зайвою інформацією при простих запитах до неї (див. розділ 15.4).

Принцип обліку різнорідності користувачів припускає, що з системою можуть працювати разные їх типи. Частина користувачів працює з системою нерегулярно, час від часу. Але існує і інший тип– "досвідчені користувачі", які працюють з додатком щодня по декілька годинників. Випадкові користувачі потребують такого інтерфейсу, який "керував" би їх роботою з системою, тоді як досвідченим користувачам потрібний інтерфейс, який дозволив би їм максимально швидко взаємодіяти з системою. Крім того, оскільки деякі користувачі можуть мати різні фізичні недоліки, в інтерфейсі повинні бути засоби, які допомогли б їм перенастроювати інтерфейс під себе. Це можуть бути засоби, що дозволяють відображати збільшений текст, заміщати звук текстом, створювати кнопки великих розмірів і тому подібне

Принцип визнання різноманіття категорій користувачів може суперечити іншим принципам проектування інтерфейсів, наприклад узгодженості інтерфейсу. Аналогічно, необхідний рівень довідкової інформації для різних типів користувачів може радикально відрізнятися. Неможливо створити таку довідкову систему, яка підійшла б всім користувачам. Розробник інтерфейсу повинен завжди бути готовим до компромісних рішень залежно від реальних користувачів системи.

## 15.2. Взаємодія з користувачем

Розробникові інтерфейсу користувача обчислювальних систем необхідно вирішити два головні завдання: яким чином користувач вводитиме дані в систему і як дані будуть представлені користувачеві. "Правильний" інтерфейс повинен забезпечувати і взаємодію з користувачем, і представлення інформації.

У цьому розділі обговорюються питання взаємодії системи з користувачем. Представлення даних розглядається в розділі 15.3. Інтерфейс користувача забезпечує введення команд і даних в обчислювальну систему. На перших обчислювальних машинах був тільки один спосіб введення даних – через інтерфейс командного рядка, причому для взаємодії з машиною використовувалася спеціальна командна мова. Такий спосіб годився тільки для досвідчених користувачів, тому пізніше були розроблені спрощеніші способи введення даних. Всі ці види взаємодії можна віднести до одного з п'яти основних стилів взаємодії [316].

1. *Безпосереднє маніпулювання.* Користувач взаємодіє з об'єктами на екрані. Наприклад, для видалення файлу користувач просто перетягує його в корзину.
2. *Вибір з меню.* Користувач вибирає команду із списку пунктів меню. Дуже часто вибрана команда впливає тільки на той об'єкт, який виділений (вибраний) на екрані. При такому підході для видалення файлу користувач спочатку вибирає файл, а потім команду на видалення.
3. *Заповнення форм.* Користувач заповнює поля екранної форми. Деякі поля можуть мати своє меню (випадне меню або списки). У формі можуть бути командні кнопки, при клацанні мишею на яких ініціюють деяку дію. Щоб видалити файл за допомогою інтерфейсу, заснованого на формі, треба ввести в поле форми ім'я файлу і потім клацнути на кнопці видалення, присутній у формі.
4. *Командна мова.* Користувач вводить конкретну команду з параметрами, щоб вказати системі, що вона повинна далі робити. Щоб видалити файл, користувач вводить команду видалення з ім'ям файлу як параметр цієї команди.
5. *Природна мова.* Користувач вводить команду на природній мові. Щоб видалити файл, користувач може ввести команду "Видалити файл з ім'ям XXX".

Кожен з цих стилів взаємодії має переваги і недоліки і найкращим чином підходить різним типам додатків і різним категоріям користувачів [316]. У табл. 15.3 перераховані основні переваги і недоліки перерахованих стилів взаємодії і вказані типи додатків, в яких вони зазвичай використовуються.

Звичайно, стилі взаємодії рідко використовуються в чистому вигляді, в одному застосуванні може використовуватися одночасно декілька різних стилів. Наприклад, в операційній системі Microsoft Window підтримується декілька стилів: пряме маніпулювання піктограмами, що представляють файли і теки, вибір команд з меню, ручне введення деяких команд, таких як команди конфігурації системи, використання форм (діалогових вікон).

**Таблиця 15.3. Переваги і недоліки стилів взаємодії користувача з системою**

| Стиль взаємодії | Основні переваги | Основні недоліки | Приклади додатків |
|---|---|---|---|
| Пряме маніпулювання | Швидке і інтуїтивно зрозуміле взаємодія. Легкий у вивченні | Складна реалізація. Підходить тільки там, де є зоровий образ завдань і об'єктів | Відеоігри; системи автоматичного проектування |
| Вибір з меню | Скорочення кількості помилок користувача.<br><br>Введення з клавіатури мінімальне | Повільний варіант для досвідчених користувачів. Може бути складним, якщо меню складається з великої кількості вкладених пунктів | Головним чином системи загального призначення |
| Заповнення форм | Простий введення даних.<br><br>Легкий у вивченні | Займає простір на екрані | Системи управління запасами; обробка фінансової інформації |
| Командна мова | Могутній і гнучкий | Важкий у вивченні. Складно запобігти помилкам введення | Операційні системи; бібліотечні системи |
| Природна мова | Підходить недосвідченим користувачам.<br><br>Легко настроюється | Вимагає великого ручного набору | Системи розкладу; системи зберігання даних WWW |

Призначені для користувача інтерфейси додатків World Wide Web базуються на засобах, що надаються мовою HTML (мова розмітки Web-страниц) разом з іншими мовами, наприклад Java, який пов'язує програми з компонентами Web-страниц. В основному інтерфейси Web-страниц проектуються для випадкових користувачів і є інтерфейсами у вигляді форм. У Web-приложениях можна створювати інтерфейси, в яких застосовувався б стиль прямого маніпулювання, проте до моменту написання книги проектування таких інтерфейсів представляло достатньо складне в аспекті програмування завдання.

В принципі необхідно застосовувати різні стилі взаємодії для управління різними системними об'єктами. Даний принцип складає основу моделі Сихейма (Seeheim) призначених для користувача інтерфейсів [278]. У цій моделі розділяються представлення інформації, управління діалоговими засобами і управління додатком. Насправді така модель є швидше ідеальною, чим практичною, проте майже завжди є можливість розділити інтерфейси для різних класів користувачів (наприклад, початкуючих і досвідчених). На мал. 15.2 зображена подібна модель з розділеними інтерфейсом командної мови і графічним інтерфейсом, лежача в основі деяких операційних систем, зокрема Linux.

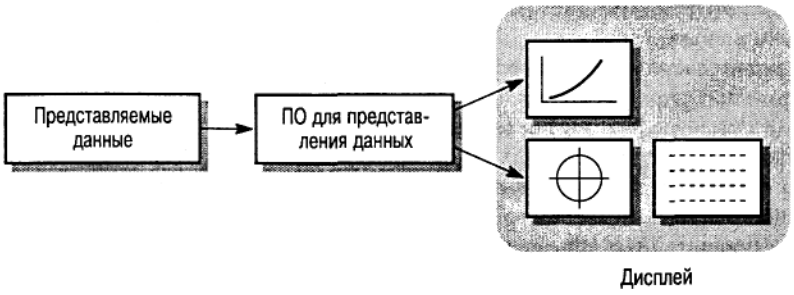

*Мал. 15.2. Множинний інтерфейс*

Розділення уявлення, взаємодії і об'єктів, включених в інтерфейс користувача, є основним принципом підходу "модель-представление-контроллер", який обговорюється в наступному розділі. Ця модель порівняння з моделлю Сихейма, проте використовується при реалізації окремих об'єктів інтерфейсу, а не всього застосування.

## 15.3. Представлення інформації

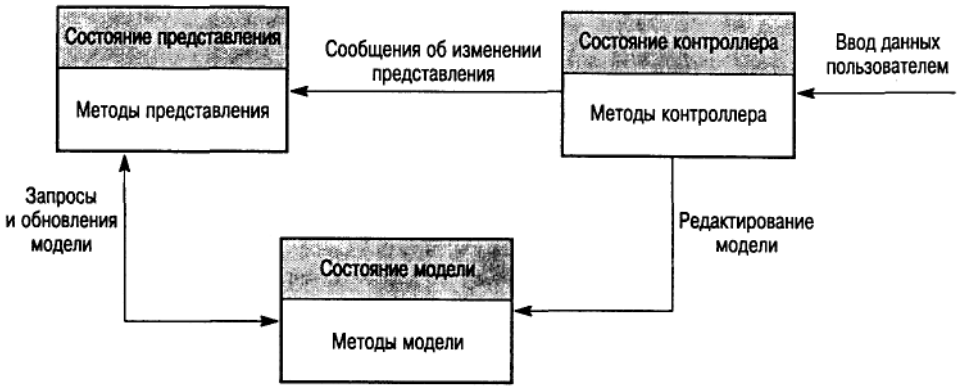
У будь-якій інтерактивній системі повинні бути засоби для представлення даних користувачам. Дані в системі можуть відображатися по-різному: наприклад, інформація, що вводиться, може відображатися безпосередньо на дисплеї (як, скажімо, текст в текстовому редакторові) або перетворюватися в графічну форму. Хорошим тоном при проектуванні систем вважається відділення представлення даних від самих даних. До деякої міри розробка такого ПО противоречит об'єктно-орієнтованому підходу, при якому методи, що виконуються над даними, повинні бути визначені самими даними. Проте в нащем випадку передбачається, що розробник об'єктів завжди знає якнайкращий спосіб представлення даних; хоча це, звичайно, не завжди так. Часто визначити якнайкращий спосіб представлення даних конкретного типу досить важко, у такому разі об'єктні структури не повинні бути "жорсткими".

Після того, як представлення даних в системі відокремлене від самих даних, зміни в уявленні даних на екрані користувача відбуваються без зміни самої системи (мал. 15.3).




*Мал. 15.3. Представлення даних*

Підхід "модель-представление-контроллер" (МПК), представлений на мал. 15.4, отримав первинне застосування в мові Smalltalk як ефективний спосіб підтримки різних представлень даних [130]. Користувач може взаємодіяти з кожним типом уявлення. Дані, що відображаються, інкапсульовані в об'єкти моделі. Кожен об'єкт моделі може мати декілька окремих об'єктів уявлень, де кожне уявлення – це різні відображення моделі. Я вже ілюстрував цей підхід в попередньому розділі, де розглядалися об'єктно-орієнтовані структури додатків.




*Мал. 15.4. Модель МПК взаємодії з користувачем*

Кожне уявлення має пов'язаний з ним об'єкт контроллера, який обробляє введені користувачем дані і забезпечує взаємодію з пристроями. Така модель може представити числові дані, наприклад, у вигляді діаграм або таблиць. Модель можна редагувати, змінюючи значення в таблиці або параметри діаграми. Такий підхід розглянутий в розділі 14 при описі патерну Оглядач (розділ 14.3).

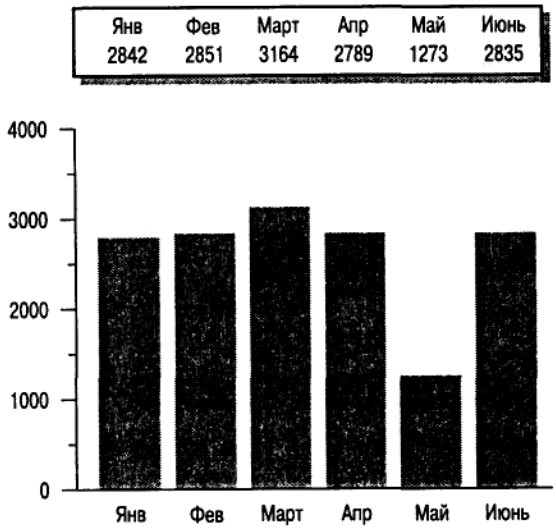
Щоб знайти якнайкраще представлення інформації, необхідно знати, з якими даними працюють користувачі і яким чином вони застосовуються в системі. Ухвалюючи рішення за уявленням даних, розробник повинен враховувати ряд чинників.

1. Що потрібне користувачеві – точні значення даних або співвідношення між значеннями?
2. Наскільки швидко відбуватимуться зміни значень даних? Чи потрібно негайно показувати користувачеві зміну значень?
3. Чи повинен користувач робити які-небудь дії у відповідь на зміну даних?
4. Чи потрібно користувачеві взаємодіяти з інформацією, що відображається, за допомогою інтерфейсу з прямим маніпулюванням?
5. Інформація повинна відображатися в текстовому (описово) або числовому форматі? Чи важливі відносні значення елементів даних?

Якщо дані не змінюються протягом сеансу роботи з системою, їх можна представити або в графічному, або в текстовому вигляді, залежно від типу додатку. Текстове представлення даних займає на екрані мало місця, але у такому разі дані не можна охопити одним поглядом. За допомогою різних стилів уявлення незмінні дані слід відокремити від даних, що динамічно змінюються. Наприклад, статичні дані можна виділити особливим шрифтом, підкреслити особливим кольором або позначити піктограмами.

Якщо потрібна точна цифрова інформація і дані змінюються відносно поволі, їх можна відображати в текстовому вигляді. Там, де дані змінюються швидко, зазвичай використовується графічне уявлення.

Як приклад розглянемо систему, яка щомісячний записує і підбиває підсумки за даними продажів якійсь компанії. На мал. 15.5 видно, що одні і ті ж дані можна представити у вигляді тексту і в графічному вигляді.

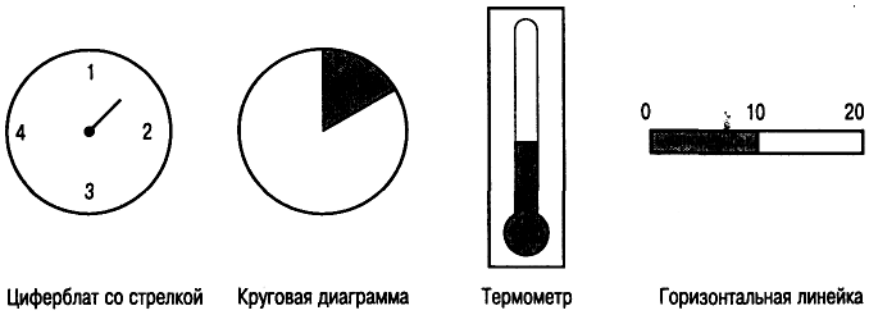



*Мал. 15.5. Альтернативні представлення даних*

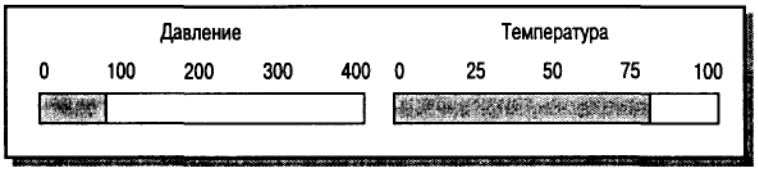
Менеджерам, що вивчають дані про продажі, зазвичай більше потрібні тенденції зміни або аномальні дані, чим їх точні значення. Графічне представлення цієї інформації у вигляді гістограми дозволяє виділити аномальні дані за березень і травень, що значно відрізняються від решти даних. Як видно з мал. 15.5, дані в текстовому уявленні займають менше місця, чим в графічному.

Динамічні зміни числових даних краще відображати графічно, використовуючи аналогові уявлення. Цифрові екрани, що постійно змінюються, збивають користувачів з пантелику, оскільки точні значення даних швидко не сприймаються. Графічне відображення даних при необхідності можна доповнити точними значеннями. Різні способи представлення числових даних, що змінюються, показані на мал. 15.6.

Безперервні аналогові відображення допомагають спостерігачеві оцінити відносні значення даних. На мал. 15.7 числових значень температури і тиску приблизно однакові. Але при графічному відображенні видно, що значення температури близьке до максимального, тоді як значення тиску не досягло навіть 25% від максимуму. Зазвичай, окрім поточного значення, спостерігачеві потрібно знати максимальні (або мінімальні) можливі значення. Він винен в думці обчислювати відносний стан прочитуваних даних. Додатковий час, необхідний для розрахунків, може привести до помилок оператора в стресових ситуаціях, коли виникають проблеми і на дисплеї відображаються аномальні дані.

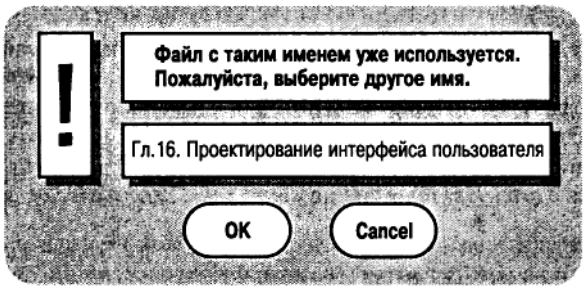



*Мал. 15.6. Способи представлення числових даних, що динамічно змінюються*




*Мал. 15.7. Графічне представлення даних, що показує значення по відношенню до максимальних*

При представленні точних буквено-цифрових даних для виділення особливій інформації можна використовувати графічні елементи. Замість звичайного рядка дані краще помістити в прямокутник або відзначити піктограмою (мал. 15.8). Прямокутник з повідомленням поміщається поверх поточного екрану, тим самим привертаючи до нього увагу користувача.




*Мал. 15.8. Виділення буквено-цифрових даних*

Виділення інформації за допомогою графічних елементів можна також використовувати для залучення уваги до змін, що відбуваються в різних частинах екрану. Але, якщо зміни відбуваються дуже швидко, не слід використовувати графічні елементи, оскільки швидкі зміни можуть привести до накладення екранів, що збиває з пантелику і дратує користувачів.

При представленні великих об'ємів даних можна використовувати різні прийоми візуалізація, яка указує на споріднені елементи даних. Розробники інтерфейсів повинні пам'ятати про можливості візуалізацію, особливо якщо інтерфейс системи повинен відображати фізичну суть (об'єкти). Ось декілька прикладів візуалізації даних.

1. Відображення метеорологічних даних, зібраних з різних джерел, у вигляді метеорологічних карт з ізобарами, повітряними фронтами і тому подібне
2. Графічне відображення стану телефонної мережі у вигляді зв'язаної безлічі вузлів.
3. Візуалізація стану хімічного процесу з показом тиску і температур в групі зв'язаних між собою резервуарів і труб.
4. Модель молекули і маніпулювання нею в тривимірному просторі за допомогою системи віртуальної реальності.
5. Відображення безлічі Web-страниц у вигляді дерева гіпертекстових посилань [208].

У книзі [316] міститься хороший опис різних підходів до візуалізації. Там же представлені різні класи візуалізація, часто використовувана на практиці. Зокрема, до них відноситься візуалізація даних з використанням двух- і тривимірних дерев і мереж. Більшість з них пов'язані з відображенням великих об'ємів даних, керованих комп'ютером. Найширше візуалізація застосовується в інтерфейсах для представлення деяких фізичних структур, наприклад молекулярної структури ліків, телекомунікаційних мереж і тому подібне Тривимірні уявлення, для створення яких використовується спеціальне устаткування віртуальної реальності, є особливо ефективним способом візуалізації.

### 15.3.1. Використання в інтерфейсах кольору

У всіх інтерактивних системах, незалежно від їх призначення, підтримуються кольорові екрани, тому в призначених для користувача інтерфейсах часто використовуються різні кольори. У деяких системах кольори застосовують в основному для виділення певних елементів (наприклад, в текстових редакторах для виділення фрагментів тексту); у інших системах (таких, як системи автоматичного проектування) квітами позначають різні рівні проектів.

Правильне використання квітів робить інтерфейс користувача зручнішим для розуміння і управління. Разом з тим використання квітів може бути неправильним, внаслідок чого створюються інтерфейси, які візуально непривабливі і навіть провокують помилки. Основним принципом розробників інтерфейсів повинне бути обережне використання квітів на екранах.

У роботі [316] дається 14 правил ефективного використання кольору в призначених для користувача інтерфейсах. Ось найбільш важливі з них.

1. *Використовуйте обмежену кількість квітів.* Для вікон не слід використовувати більше чотирьох або п'яти різних квітів, в інтерфейсі системи не повинно бути більше семи квітів.
2. *Використовуйте різні кольори для показу змін в стані системи.* Якщо на екрані змінилися кольори, значить, відбулася якась подія. Виділення кольором особливо важливе в складних екранах, в яких відображаються сотні різних об'єктів.
3. *Для допомоги користувачеві використовуйте колірне кодування.* Якщо користувачам необхідно виділяти аномальні елементи, виділите їх кольором; якщо потрібно знайти подібні елементи, виділите їх однаковим кольором.
4. *Використовуйте колірне кодування продумано і послідовно.* Якщо в якій-небудь частині системи повідомлення про помилку відображаються, наприклад, червоним кольором, то у всіх інших частинах подібні повідомлення повинні відображатися таким же кольором. Тоді червоний колір не слід використовувати де-небудь ще. Якщо ж червоний колір використовується ще десь в системі, користувач може інтерпретувати появу червоного кольору як повідомлення про помилку. Слід пам'ятати, що у певних типів користувачів є свої уявлення про значення окремих квітів.
5. *Обережно використовуйте доповнюючі кольори.* Фізіологічні особливості людського ока не дозволяють одночасно сфокусуватися на червоному і синьому кольорах. Тому послідовність червоних і синіх зображень викликає зорова напруга. Деякі комбінації квітів також можуть візуально порушувати або утрудняти читання.

Найчастіше розробники інтерфейсів допускають дві помилки: прив'язка значення до певного кольору і використання великої кількості квітів на екрані. Використовувати кольори для представлення значення не слід по двох причинах. Близько 10% людей мають нечітке уявлення про квіти і тому можуть неправильно інтерпретувати значення. У різних груп людей різне сприйняття квітів; крім того, в різних професіях існують свої угоди про значення окремих квітів. Користувачі на підставі отриманих знань можуть неадекватно інтерпретувати один і той же колір. Наприклад, водієм червоний колір сприймається як *небезпека.* А у хіміка червоний колір означає *гарячий.*

При використанні дуже яскравих квітів або дуже великої їх кількості відображення стають плутаними. Різноманіття квітів збиває з пантелику користувача (так, наприклад, на деякі абстрактні картини не можна дивитися тривалий час без напруги) і викликає у нього зорове стомлення. Непослідовне використання квітів також дезорієнтує користувача.

## 15.4. Засоби підтримки користувача

У першому розділі цього розділу був запропонований принцип проектування, згідно якому інтерфейс користувача повинен завжди забезпечувати деякий тип оперативної довідкової системи. Довідкові системи – один з основних аспектів проектування інтерфейсу користувача. Довідкову систему додатку складають:

- повідомлення, що генеруються системою у відповідь на дії користувача;
- діалогова довідкова система;
- документація, що поставляється з системою.

Оскільки проектування корисної і змістовної інформації для користувача – справа вельми серйозна, воно повинне оцінюватися на тому ж рівні, що і архітектура системи або програмний код. Проектування повідомлень вимагає значного часу і чималих зусиль. Доречно привертати до цього процесу професійних письменників і художників-графіків. При проектуванні повідомлень про помилки або текстової довідки необхідно враховувати чинники, перераховані в табл. 15.4.

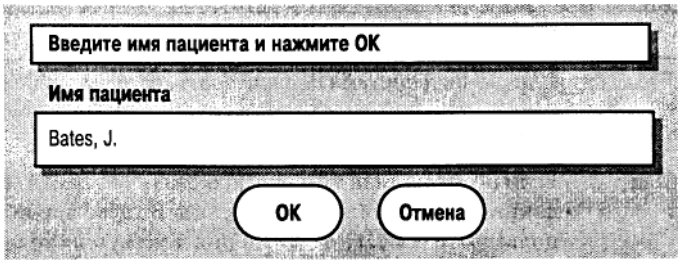
**Таблиця 15.4. Чинники проектування текстових повідомлень**

| Чинник | Опис |
|---|---|
| Зміст | Довідкова система повинна знати, що робить користувач, і реагувати на його дії повідомленнями відповідного змісту |
| Досвід користувача | Якщо користувачі добре знайомі з системою, їм не потрібні довгі і докладні повідомлення. В той же час початкуючим користувачам такі повідомлення здадуться складними, малозрозумілими і дуже короткими. У довідковій системі повинні підтримуватися обидва типи повідомлень, а також повинні бути засоби, що дозволяють користувачеві управляти складністю повідомлень |
| Професійний рівень користувача | Повідомлення повинні містити відомості, відповідні професійному рівню користувачів. У повідомленнях для користувачів різного рівня необхідно застосовувати різну термінологію |
| Стиль повідомлень | Повідомлення повинні мати позитивний, а не негативний відтінок. Завжди слід використовувати активний, а не пасивний тон звернення. У повідомленнях не повинно бути образ або спроб пожартувати |
| Культура | Розробник повідомлень повинен бути знайомий з культурою тієї країни, де продається система. Повідомлення, цілком доречне в культурі однієї країни, може виявитися неприйнятним в іншій |

### 15.4.1. Повідомлення про помилки

Перше враження, яке користувач отримує при роботі з програмною системою, грунтується на повідомленнях про помилки. Недосвідчені користувачі, зробивши помилку, повинні зрозуміти повідомлення, що з'явилося, про помилку.

Новачки і досвідчені користувачі повинні передбачати ситуації, при яких можуть виникнути повідомлення про помилки. Наприклад, хай користувачем системи є медсестра госпіталю, що працює у відділенні інтенсивної терапії. Обстеження пацієнтів виконується на відповідному устаткуванні, пов'язаному з обчислювальною системою. Щоб проглянути поточний стан пацієнта, користувач системи вибирає пункт меню Показати і набирає ім'я пацієнта в полі введення (мал. 15.9).

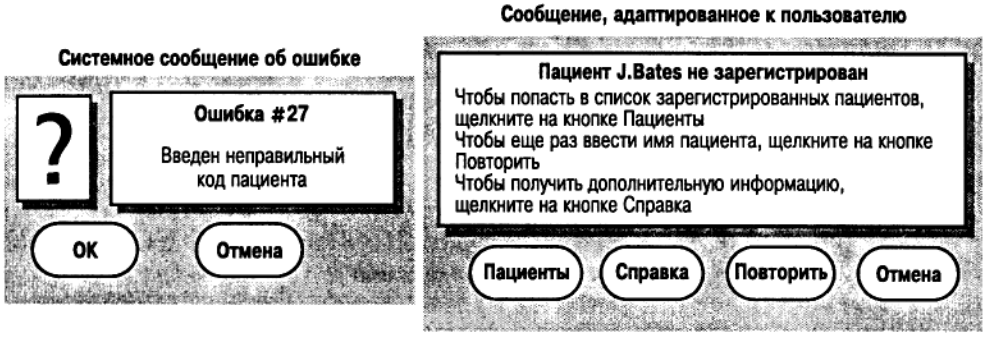



*Мал. 15.9. Введення імені пацієнта*

Хай медсестра ввела ім'я пацієнта Bates, замість Pates. Система не знаходить пацієнта з таким ім'ям і генерує повідомлення про помилку. Повідомлення про помилку повинні бути завжди ввічливими, короткими, послідовними і конструктивними, не містити образ. Не слід також використовувати звукові сигнали або інші звуки, які можуть збити з пантелику корис-

тувача. Непогано включити в повідомлення варіанти виправлення помилки. Повідомлення про помилку повинне бути пов'язане з контекстно-залежною довідкою.

На мал. 15.10 показані приклади двох повідомлень про помилку. Повідомлення, розташоване зліва, спроектоване погано. Воно негативне (звинувачує користувача в здійсненні помилки), не адаптоване до рівня знань і досвідченості користувача, не враховує змісту помилки. У цьому повідомленні не пропонуються способи виправлення ситуації, що склалася. Крім того, в повідомленні використані специфічні терміни (номер помилки), не зрозумілі користувачеві. Повідомлення справа набагато краще. Воно позитивне, в нім використовуються медичні терміни і пропонується простий спосіб виправлення помилки за допомогою клацання на одній з кнопок. У разі потреби користувач може викликати довідку.




*Мал. 15.10. Приклади повідомлень про помилки*

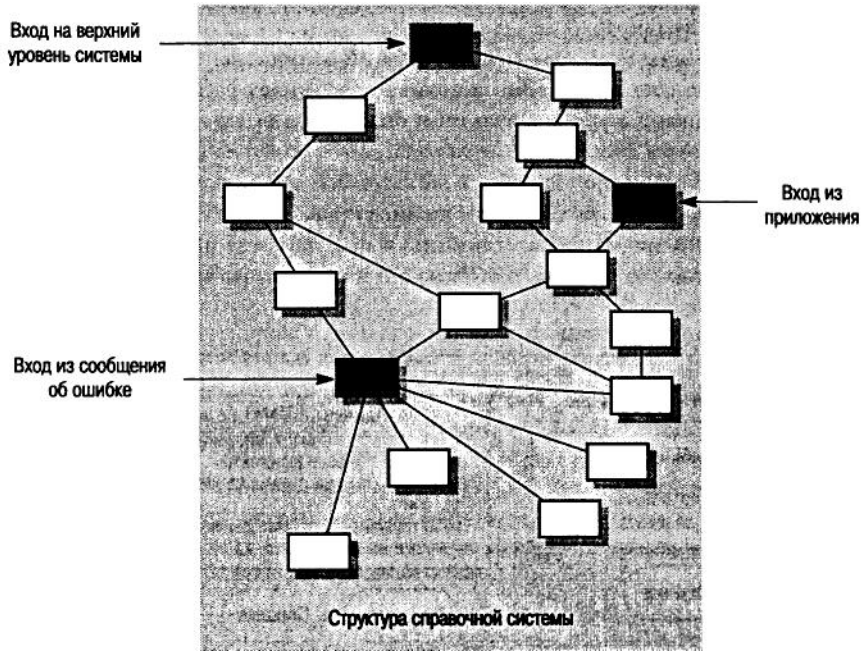
### 15.4.2. Проектування довідкової системи

При отриманні повідомлення про помилку користувач часто не знає, що робити, і звертається до довідкової системи за інформацією. Довідкова система повинна надавати різні типи інформації: як ту, що допомагає користувачеві в скрутних ситуаціях, так і конкретну інформацію, яку шукає користувач. Для цього довідкова система повинна мати різні засоби і різні структури повідомлень.

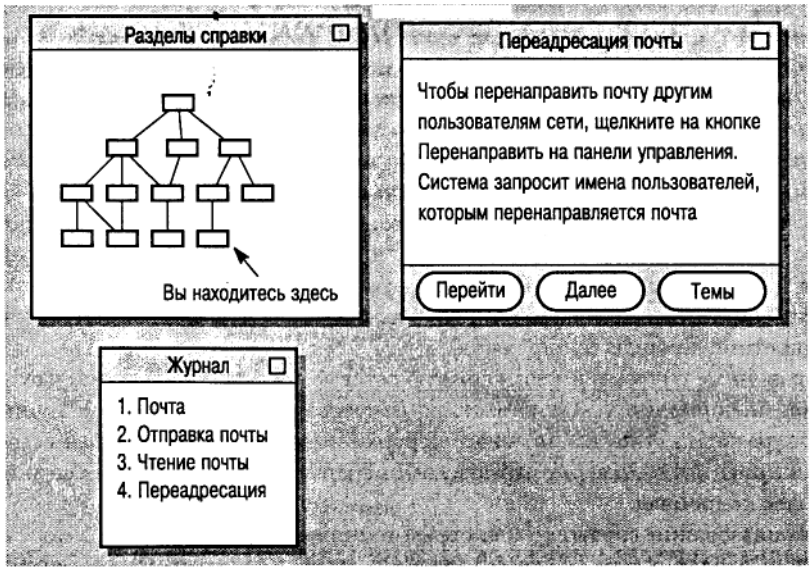
Довідкова система повинна забезпечувати користувачеві декілька різних точок входу (мал. 15.11). Користувач може увійти до неї на верхньому рівні її ієрархічної структури і тут оглянути всі розділи довідкової інформації. Інші точки входу в довідкову систему – за допомогою вікон повідомлень про помилки або шляхом отримання опису конкретної команди додатку.

Всі довідкові системи мають складну мережеву структуру, в якій кожен розділ довідкової інформації може посилатися на дещо інших інформаційних розділів. Структура такої мережі, як правило, ієрархічна, з перехресними посиланнями, як показано на мал. 15.11. Нагорі структурної ієрархії міститься загальна інформація, внизу – докладніша.

При використанні довідкової системи виникають проблеми, пов'язані з тим, що користувачі входять в мережу після здійснення помилки і потім переміщаються по мережі довідкової системи. Через деякий час вони заплутуються і не можуть визначити, в якому місці довідкової системи знаходяться. У такій ситуації користувачі повинні завершити сеанс роботи з довідковою системою і знов почати роботу з деякої відомої точки довідкової мережі.




*Мал. 15.11. Точки входу в довідкову систему*




*Мал. 15.12. Вікна довідкової системи*

Відображення довідкової інформації в декількох вікнах спрощує подібну ситуацію. На мал. 15.12 показаний екран, на якому розташовано три вікна довідки. Проте простір екрану завжди обмежений і розробникові слід пам'ятати, що додаткові вікна можуть приховати іншу потрібну інформацію.

Тексти довідкової системи необхідно готувати спільно з розробниками додатку. Довідкові розділи не повинні бути просто відтворенням керівництва користувача, оскільки інформація на папері і на екрані сприймається по-різному. Сам текст (а також його розташування і стиль) повинен бути ретельно продуманий, щоб його можна було прочитати у вікні щодо малого розміру. Розділ довідки **Переадресація** пошти на мал. 15.12 порівняно невеликий – в будь-якому довідковому розділі повинна бути тільки найнеобхідніша інформація. У вікні, що відображає довідковий розділ, розташовано три кнопки: для показу додатковій інформації, для переміщення по тексту розділу і для виклику списку довідкових тим.

У вікні **Журнал*** показаний список вже проглянутих розділів. Можна повернутися в один з них, вибравши відповідний пункт із списку. Вікно навігації по довідковій системі – це графічна "карта" мережі довідкової системи. Поточна позиція на карті повинна бути виділена кольором, тінями або, як в нашому випадку, підписом.

* *Таке вікно в англомовних програмних застосуваннях зазвичай називається History (Історія).* - Прим, ред.

У користувачів є декілька можливостей переміщення між розділами довідкової системи: можна перейти до розділу безпосередньо з розділу, що відображається, можна вибрати потрібний розділ з вікна **Журнал**, щоб проглянути його ще раз, і, нарешті, можна вибрати відповідний вузол на карті довідкової мережі і перейти до цього вузла.

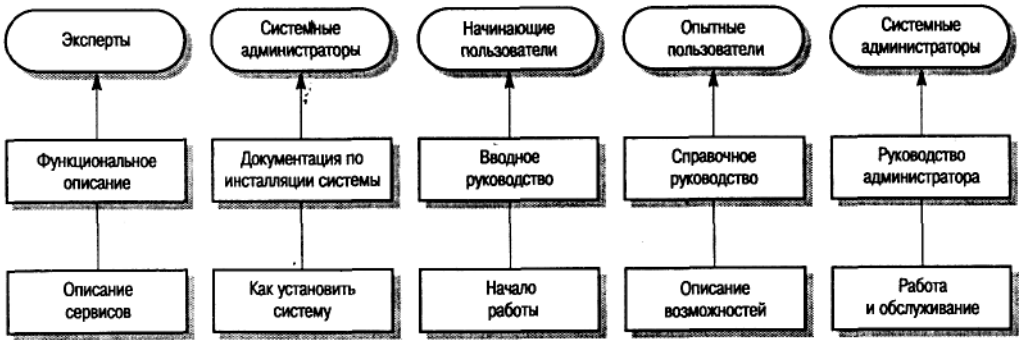
Довідкову систему можна реалізувати у вигляді групи зв'язаних Web-страниц або за допомогою узагальненої гіпертекстової системи, інтегрованої з додатком. Ієрархічна структура легко реалізується у вигляді гіпертекстових посилань. Web-системы мають переваги: вони прості в реалізації і не вимагають спеціального програмного забезпечення. Проте при створенні контекстно-залежної довідки можуть виникнути труднощі при пов'язанні її з додатком.

### 15.4.3. Документація користувача

Строго кажучи, документація не є частиною призначеного для користувача інтерфейсу, проте проектування оперативної довідкової підтримки разом з документацією є хорошим правилом. Системне керівництво надає докладнішу інформацію, ніж діалогові довідкові системи, і будуються так, щоб бути корисними користувачам різного рівня.

Для того, щоб документація, що поставляється спільно з програмною системою, була корисна всім системним користувачам, вона повинна містити п'ять описаних нижче документів (або, можливо, розділів в одному документі) (мал. 15.13).

1. *Функціональний опис,* в якому коротко представлені функціональні можливості системи. Прочитавши функціональний опис і ввідне керівництво, користувач повинен визначити, чи та це система, яка йому потрібна.
2. *Документ по інсталяції системи, в* якому міститься інформація по установці системи. Тут повинні бути відомості про диски, на яких поставляється система, опис файлів, що знаходяться на цих дисках, і мінімальні вимоги до конфігурації. У документі повинні бути інструкція по інсталяції і докладніша інформація по установці файлів, залежних від конфігурації системи.
3. *Ввідне керівництво,* що представляє неформальне введення в систему, що описує її "повсякденне" використання. У цьому документі повинна міститися інформація про те, як почати роботу з системою, як використовувати загальні можливості системи. Всі описи повинні бути забезпечені прикладами і містити відомості про те, як відновити систему після помилки і як почати наново роботу. У книзі [68] запропонований ефективний спосіб складання ввідного керівництва, при якому основна увага приділена відновленню системи після помилок, а решта всієї інформації, необхідної користувачам, зводиться до мінімуму.
4. *Довідкове керівництво,* в якому описані можливості системи і їх використання, представлений список повідомлень про помилки і можливі причини їх появи, розглянуті способи відновлення системи після виявлення помилок.
5. *Керівництво адміністратора,* необхідне для деяких типів програмних систем. У нім даний опис повідомлень, що генеруються системою при взаємодії з іншими системами, і описані способи реагування на ці повідомлення. Якщо в систему включена апаратна частина, то в керівництві адміністратора повинна бути інформація про те, як виявити і усунути несправності, пов'язані з апаратурою, як підключити нові периферійні пристрої і тому подібне




*Мал. 15.13. Типи призначеної для користувача документації*

Разом з перерахованим керівництвом необхідно надавати іншу зручну в роботі документацію. Для досвідчених користувачів системи зручні різного вигляду наочні покажчики, які допомагають швидко проглянути список можливостей системи і способи їх використання.

## 15.5. Оцінювання інтерфейсу

Це процес, в якому оцінюється зручність використання інтерфейсу і ступінь його відповідності вимогам користувача. Таким чином, оцінювання інтерфейсу є частиною загального процесу тестування і атестації систем ПІВ

У ідеалі оцінювання повинне проводитися відповідно до показників зручності використання інтерфейсу, перерахованих в табл. 15.5. Кожен з цих показників можна оцінити чисельно. Наприклад, изучаемость можна оцінити таким чином: досвідчений оператор після тригодинного навчання повинен уміти використовувати 80% функціональних можливостей системи. Проте частіше зручність використання інтерфейсу оцінюється якісно, а не через числові показники.

**Таблиця 15.5. Показники зручності використання інтерфейсу**

| Показник | Опис |
|---|---|
| Ізучаємость | Кількість часу навчання, необхідне для початку продуктивної роботи з системою |
| Швидкість роботи | Швидкість реакції системи на дії користувача |
| Стійкість | Стійкість системи до помилок користувача |
| Відновлюваність | Здатність системи відновлюватися після помилок користувача |
| Адаптується | Здатність системи "підстроюватися" до різних стилів роботи користувачів |

Повне оцінювання призначеного для користувача інтерфейсу може виявитися вельми дорогим, в цей процес будуть залучені фахівці з когнітивної психології і дизайнери. У процес оцінювання можуть входити розробка і виконання ряду статистичних експериментів з користувачами в спеціально створених лабораторіях і з необхідним для спостереження устаткуванням. Таке оцінювання інтерфейсу економічно нерентабельно для систем, що розробляються в невеликих організаціях з обмеженими ресурсами.

Існують простіші і менш дорогі методики оцінювання інтерфейсів користувача, що дозволяють виявити окремі дефекти в інтерфейсах.

1. Анкети, в яких користувач дає оцінку інтерфейсу.
2. Спостереження за роботою користувачів з подальшим обговоренням їх способів використання системи при вирішенні конкретних завдань.
3. Відеоспостереження типового використання системи.
4. Додавання в систему програмної коди, яка збирала б інформацію про найчастіше використовувані системні сервіси і найбільш поширені помилки.

Анкетування користувачів – відносно дешевий спосіб оцінки інтерфейсу. Питання повинні бути точними, а не загальними. Не слід використовувати питання типу "Будь ласка, прокоментуйте практичність системи", оскільки відповіді, ймовірно, істотно розрізнятимуться. Краще ставити конкретні питання, наприклад: "Оцініть зрозумілість повідомлень про помилки за шкалою від 1 до 5. Оцінка 1 означає повністю зрозуміле повідомлення, 5 – малозрозуміле". На такі питання легко відповісти і ймовірніше отримати в результаті корисну для поліпшення інтерфейсу інформацію.

Під час заповнення анкети користувачі повинні обов'язково оцінити власний досвід і знання. Такого роду відомості дозволять розробникам зафіксувати, користувачі з яким рівнем знань мають проблеми з інтерфейсом. Якщо проект інтерфейсу вже створений і пройшов оцінювання в паперовому вигляді, анкети можна використовувати навіть до повної реалізації системи.

При спостереженні користувачів за роботою оцінюється, як вони взаємодіють з системою, які використовують сервіси, які здійснюють помилки і тому подібне Разом із спостереженнями можуть проводитися семінари, на яких користувачі розповідають про свої спроби вирішити ті або інші проблеми і про те, як вони розуміють систему і як використовують її для досягнення цілей.

Відеоустаткування відносне недорого, тому до безпосереднього спостереження можна додати відеозапис призначених для користувача семінарів для подальшого аналізу. Повний аналіз відеоматеріалів дорогий і вимагає спеціально оснащеного комплекту з декількома камерами, направленими на користувача і на екран. Проте відеозапис окремих дій користувача може виявитися корисним для виявлення проблем. Щоб визначити, які саме дії викликають проблеми у користувача, слід удатися до інших методів оцінювання.

Аналіз відеозаписів дозволяє розробникові встановити, чи багато рухів руками вимушений здійснювати користувач (у деяких системах користувачеві постійно доводиться переходити з клавіатури на мишу), і виявити неприродні рухи очей. Якщо при роботі з інтерфейсом потрібно часто зміщувати зоровий фокус, користувач може зробити більше помилок і пропустити які-небудь частини зображення.

Вставка в програму коди, що збирає статистичні дані при використанні системи, покращує інтерфейс декількома способами. Виявляються найбільш часто використовувані операції. Інтерфейс змінюється так, щоб ці операції вибиралися швидше в порівнянні з іншими. Наприклад, у вертикальному або випадному меню найбільш часто використовувані команди повинні знаходитися вверху списку. Такий код також дозволить виявити і змінити команди, сприяючі появі помилок.

Нарешті, в кожній програмі повинні бути нескладні засоби, за допомогою яких користувач зможе передавати розробникам повідомлення з "скаргами". Такі засоби переконують користувачів в тому, що на їх думку зважають. А розробники інтерфейсу і інші фахівці можуть отримати швидкий зворотний зв'язок щодо окремих проблем інтерфейсу.

Жоден з цих далеко не складних методів оцінки призначеного для користувача інтерфейсу не є надійним і не гарантує вирішення всіх проблем інтерфейсу. Разом з тим перед випуском системи ці методи можна застосувати в групі добровольців, не витрачаючи значних засобів. При цьому виявляється і виправляється більшість проблем в інтерфейсі користувача.

## КЛЮЧОВІ ПОНЯТТЯ

- Процес проектування інтерфейсу повинен орієнтуватися на користувача. Інтерфейс повинен взаємодіяти з користувачем на його «мові», бути логічним і послідовним. У інтерфейсі повинні бути довідкові засоби, що допомагають користувачам при роботі з системою, і засоби відновлення після помилок.
- Існує декілька стилів взаємодії з програмними системами: безпосереднє маніпулювання, системне меню, заповнення форми, командні мови і природна мова.
- Для відображення тенденцій числових даних і їх приблизних значень слід використовувати графічні уявлення. Числове уявлення повинне застосовуватися тільки тоді, коли потрібно відобразити точні значення даних.

- Кольори в інтерфейсі користувача повинні використовуватися обережно і послідовно. Розробники повинні завжди пам'ятати, що багато хто не розрізняє квітів.
- Повідомлення про помилки не повинні містити звинувачень в адресу користувача. Вони повинні пропонувати варіанти виправлення помилки і забезпечувати зв'язок з довідковою системою.
- У документації користувача повинне бути керівництво для початкуючих і досвідчених користувачів. Для системного адміністратора повинні бути окремі документи.
- У системній специфікації бажано мати кількісні значення для показників зручності використання інтерфейсу, а процес його оцінювання повинен перевіряти систему на відповідність цим вимогам.

## Вправи

**15.1.** У розділі 15.1 наголошувалося, що об'єкти, якими маніпулює користувач, повинні відображати його поняття наочної області додатку ПО (а не комп'ютерній наочній області). Запропонуєте відповідні об'єкти маніпулювання для наступних типів користувачів і систем.
- Автоматизований каталог товарів для асистента на складі.
- Система спостереження за безпекою літака для льотчика цивільної авіації.
- Фінансова база даних для менеджера.
- Система управління патрульними машинами для поліцейського.

**15.2.** Опишіть ситуації, в яких безрозсудно або неможливо підтримувати інтерфейс користувача.

**15.3.** Які чинники слід враховувати при проектуванні інтерфейсів, що використовують меню, для таких систем, як банкомати? Опишіть основні риси інтерфейсу банкомату, яким ви користуєтеся.

**15.4.** Запропонуєте способи адаптації призначеного для користувача інтерфейсу в системах електронної комерції (наприклад, віртуального книжкового магазина або магазина музичних дисків) для користувачів, що мають фізичні недоліки, наприклад поганий зір або проблеми опорно-рухової системи.

**15.5.** Обговорите переваги графічного способу відображення інформації і приведіть чотири приклади додатків, в яких доречніше використовувати графічне представлення числових даних, а не табличне.

**15.6.** Якими основними принципами слід керуватися при використанні квітів в інтерфейсах користувача? Запропонуєте ефективніший спосіб використання квітів в інтерфейсі будь-якого відомого вам застосування.

**15.7.** Розглянете повідомлення про помилки, що генеруються операційними системами MS Windows, Unix, MACOS або який-небудь інший. Як їх можна поліпшити?

**15.8.** Складіть анкету по збору даних про інтерфейс якої-небудь відомої вам програми (наприклад, текстового редактора). Якщо є можливість, розповсюдите цю анкету серед інших користувачів і спробуйте оцінити результати анкетування. Що ви дізналися про інтерфейс програми з анкет?

**15.9.** Обговорите, чи етично розробляти програмні системи, не погодивши з кінцевими користувачами ті елементи системи, які вони контролюватимуть.

**15.10.** З якими етичними проблемами стикаються розробники інтерфейсів, коли намагаються погоджувати запити кінцевих користувачів системи з вимогами організації, яка оплачує розробку даної системи?