МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ
Факультет інформаційних технологій

С.В.БАРАН

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ВИКОРИСТАННЯМ ПАТЕРНІВ ПРОЕКТУВАННЯ

НАВЧАЛЬНИЙ ПОСІБНИК

КРИВИЙ РІГ 2023

Рецензенти:

А.І. Купін – доктор технічних наук, професор, завідувач кафедри комп'ютерних систем та мереж Криворізького національного університету;

І.О. Музика, кандидат технічних наук, доцент кафедри комп'ютерних систем та мереж, декан факультету інформаційних технологій Криворізького національного університету;

В.С. Лисенко, к.е.н., доцент кафедри інформатики та прикладного програмного забезпечення Державного університету економіки і технологій.

Рекомендовано до друку Вченою радою Державного університету економіки і технологій протокол № 9 від 23.02.2023р.

С.В.Баран. Розробка програмного забезпечення з використанням патернів проектування: Навчальний посібник. – Кривий Ріг: Державний університет економіки і технологій, 2023. –203 с.

Навчальний посібник призначений для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення» денної та заочної форм навчання.

Навчальний посібник містить практичні рекомендації для допомоги здобувачам вищої освіти у вивченні освітньої компоненти "Розробка програмного забезпечення з використанням патернів проектування", зокрема передбачають дослідження моделей архітектур, патернів та методологій розробки програмного забезпечення, використаня методів практико-орієнтованого навчання, зокрема: процес аналізу предметної області; розробки моделі варіантів використання із застосуванням шаблонів проектування; розробки класів; опису класів та їх атрибутів, методів; опису зв'язків та залежностей між класами; програмну реалізацію у вигляді С++ коду.

Розглянуто базові питання й підходи до розробки програмного забезпечення з використанням породжуючих та структурних патернів, а також патернів поведінки. Розглянуто сучасні моделі та підходи до конструювання програмного забезпечення.

ЗМІСТ

ВСТУП........ 5
РОЗДІЛ 1 МОДЕЛІ КОНСТРУЮВАННЯ........ 6
1.1. Визначення технології конструювання програмного забезпечення. 6
1.2. Класичний життєвий цикл. 6
1.3. Макетування. 8
1.4. Стратегії конструювання програмного забезпечення. 10
1.5. Інкрементна модель. 11
1.6. Швидка розробка застосувань. 12
1.7. Спіральна модель........ 13
1.8. Компонентно-орієнтована модель. 15
1.9. Ваговиті і полегшені процеси. 15
1.10. XP-процесс. 17
Контрольні питання........ 21
РОЗДІЛ 2 ПАТЕРНИ ПРОЕКТУВАННЯ. ПОРОДЖУЮЧІ ПАТЕРНИ........ 23
2.1. Поняття патернів проектування. 23
2.2. Характеристика породжуючих патернів. 23
2.3. Патерн Singleton........ 24
2.4. Патерн Abstract Factory. 32
2.5. Патерн Factory Method. 41
2.6. Патерн Builder........ 47
2.7. Патерн Prototype. 56
2.8. Патерн Object Pool. 65
2.9. Патерн Multiton. 69
Контрольні питання........ 71
РОЗДІЛ 3 СТРУКТУРНІ ПАТЕРНИ 72
3.1. Характеристика структурних патернів. 72
3.2. Патерн Adapter. 72
3.3. Патерн Bridge. 77
3.4. Патерн Composite........ 82
3.5. Патерн Decorator. 90
3.6. Патерн Facade. 96
3.7. Патерн Flyweight........ 102
3.8. Патерн Proxy. 108
Контрольні питання........ 113
РОЗДІЛ 4 ПАТЕРНИ ПОВЕДІНКИ........ 114
4.1. Характеристика патернів поведінки. 114
4.2. Патерн Chain of Responsibility. 115
4.3. Патерн Command. 120
4.4. Патерн Interpreter. 126
4.5. Патерн Iterator. 133

4.6. Патерн Mediator. 143
4.7. Патерн Memento.......... 149
4.8. Патерн Observer. 155
4.9. Патерн State.......... 162
4.10. Патерн Strategy.......... 169
4.11. Патерн Template Method.......... 175
4.12. Патерн Visitor.......... 179
Контрольні питання.......... 186
РОЗДІЛ 5 ЛАБОРАТОРНІ РОБОТИ 187
5.1. Лабораторна робота №1. Дослідження архітектурних моделей та методологій розробки ПЗ для обраної предметної області. 187
5.2. Лабораторна робота №2. Розробка програмного забезпечення з використанням патерну проектування “Abstract Factory” та “Factory Method”. 191
5.3. Лабораторна робота №3. Розробка програмного забезпечення з використанням патерну проектування “Decorator” та “Flyweight”. 192
5.4. Лабораторна робота №4. Розробка програмного забезпечення з використанням патерну проектування “Proxy” та “Facade”. 194
5.5. Лабораторна робота №5. Розробка програмного забезпечення з використанням патерну проектування “Observer” та “Visitor”. 196
5.6. Лабораторна робота №6. Розробка програмного забезпечення з використанням патерну проектування “Strategy” та “Chain of Responsibility”. 198
5.7. Лабораторна робота №7. Розробка програмного забезпечення з використанням патерну проектування “State” та “Composite”. 200
ВИСНОВКИ 202
СПИСОК ЛІТЕРАТУРИ 203

ВСТУП

Нині найвживанішим підходом до організації повторного використання коду є ідея патернів проектування як шаблонів, що визначають розв’язання окремих задач, які часто повторюються у різних проектах програмних систем. Введено класифікацію, яка розділяє патерни за їх призначенням (породжуючі, структурні, поведінки) та рівнем використання (клас, об’єкт). Породжуючі патерни пов’язані зі створенням екземплярів об’єктів; всі вони визначають засоби логічної ізоляції клієнта від створюваних об’єктів. Структурні патерни об’єднують класи чи об’єкти в більші структури. Патерни поведінки стосуються до взаємодії та розподілення обов’язків між класами та об’єктами.

Патерни проектування спрощують повторне використання вдалих проектних і архітектурних рішень. За допомогою патернів можна поліпшити якість документації і супроводу існуючих систем, дозволяючи явно описати взаємодії класів і об'єктів, а також причини, за якими система була побудована так, а не інакше. Простіше кажучи, патерни проектування дають розробнику можливість швидше знайти «правильний» шлях, що значно поліпшує якість розробки програмного забезпечення.

Тому, метою є ознайомлення студентів з основами конструювання програмного забезпечення з використанням сучасних підходів та технологій.

Завдання: вивчення основ конструювання, типів моделей та шаблонів проектування для якісної розробки програмного забезпечення.

Предмет: використання сучасних підходів конструювання для створення програмного забезпечення.

Студент повинен знати:

– моделі розробки програмного заьезпечення;

– патерни проектування для створення об’єктів;

– структурні патерни проектування;

– поведінкові патерни проектування.

Студент повинен уміти:

–моделювати різні аспекти системи, для якої створюється програмного забезпечення;

–володіти основами конструювання програмного забезпечення;

–застосовувати та створювати компоненти багаторазового використання;

–застосовівати шаблони проектування при розробці програмного забезпечення;

–застосовівати шаблони проектування при розробці WEB-сайтів.

РОЗДІЛ 1
МОДЕЛІ КОНСТРУЮВАННЯ

1.1. Визначення технології конструювання програмного забезпечення.

Технологія конструювання програмного забезпечення (ТКПЗ) — система інженерних принципів для створення економічного програмного забезпечення, яке надійне і ефективно працює в реальних комп'ютерах.

Розрізняють методи, засоби і процедури ТКПЗ.

Методи забезпечують вирішення наступних завдань:

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

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

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

- порядок застосування методів і утиліт;
- формування звітів, форм по відповідних вимогах;
- контроль, який допомагає забезпечувати якість і координувати зміни;
- формування «віх», по яких керівники оцінюють прогрес.

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

Застосування парадигм ТКПЗ гарантує систематичний, впорядкований підхід до промислової розробки, використання і супроводу програмного забезпечення. Фактично, парадигми вносять до процесу створення програмного забезпечення організуючий інженерний початок, необхідність якого важко переоцінити.

Розглянемо найбільш популярні парадигми ТКПЗ.

1.2. Класичний життєвий цикл.

Старою парадигмою процесу розробки програмного забезпечення є класичний життєвий цикл (автор Уїнстон Ройс, 1970).

Дуже часто класичний життєвий цикл називають каскадною або водопадною моделлю, підкреслюючи, що розробка розглядається як

послідовність етапів, причому перехід на наступний, ієрархічно нижній етап відбувається тільки після повного завершення робіт на поточному етапі (рис. 1.1).

Рис. 1.1. Класичний життєвий цикл розробки програмне забезпечення

Охарактеризуємо зміст основних етапів.

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

Системний аналіз задає роль кожного елементу в комп'ютерній системі, взаємодія елементів один з одним. Оскільки програмне забезпечення є лише частиною великої системи, то аналіз починається з визначення вимог до всіх системних елементів і призначення підмножини цих вимог програмному «елементу». Необхідність системного підходу явно виявляється, коли формується інтерфейс програмного забезпечення з іншими елементами (апаратурою, людьми, базами даних). На цьому ж етапі починається рішення задачі планування проекту програмне забезпечення. В ході планування проекту визначаються обсяг проектних робіт і їх ризик, необхідні трудовитрати, формуються робочі завдання і план-графік робіт.

Аналіз вимог відноситься до програмного елементу — програмного забезпечення. Уточнюються і деталізують його функції, характеристики і інтерфейс.

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

Проектування полягає в створенні уявлень:

- архітектури програмного забезпечення;
- модульної структури програмного забезпечення;
- алгоритмічної структури програмного забезпечення;
- структури даних;
- вхідного і вихідного інтерфейсу (вхідних і вихідних форм даних).

Початкові дані для проектування містяться в *специфікації аналізу,* тобто в ході проектування виконується трансляція вимог до програмного забезпечення в безліч проектних уявлень. При вирішенні завдань проектування основна увага приділяється якості майбутнього програмного продукту.

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

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

Супровід — це внесення змін до експлуатованого програмного забезпечення.

Цілі змін:

- виправлення помилок;
- адаптація до змін зовнішньою для програмного забезпечення середовища;
- удосконалення програмного забезпечення згідно вимог замовника.

Супровід програмного забезпечення полягає в повторному застосуванні кожного з попередніх кроків (етапів) життєвого циклу до існуючої програми але не в розробці нової програми.

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

Переваги класичного життєвого циклу: дає план і часовий графік по всіх етапах проекту, упорядковує хід конструювання.

Недоліки класичного життєвого циклу:

1. Реальні проекти часто вимагають відхилення від стандартної послідовності кроків.
2. Цикл заснований на точному формулюванні початкових вимог до програмного забезпечення (реально на початку проекту вимоги замовника визначені лише частково).
3. Результати проекту доступні замовникові тільки в кінці роботи.

1.3. Макетування.

Достатньо часто замовник не може сформулювати докладні вимоги по введенню, обробці або виведенню даних для майбутнього програмного продукту. З іншого боку, розробник може сумніватися в адаптації продукту під операційну систему, формі діалогу з користувачем або в ефективності алгоритму, що реалізовується. У цих випадках доцільно використовувати

макетування.

Основна мета макетування — зняти невизначеності у вимогах замовника.

Макетування (прототипування) — це процес створення моделі необхідного програмного продукту.

Модель може приймати одну з трьох форм:

1. Паперовий макет або макет на основі ПК (зображає або малює людино-машинний діалог).
2. Працюючий макет (виконує деяку частину необхідних функцій).
3. Існуюча програма (характеристики якої потім мають бути покращенні).

Як показано на рис. 1.2, макетування ґрунтується на багатократному повторенні ітерацій, в яких беруть участь замовник і розробник.

Рис. 1.2. Макетування

Послідовність дій при макетуванні представлена на рис. 1.3. Макетування починається із збору і уточнення вимог до програмного забезпечення, що створюється. Розробник і замовник зустрічаються і визначають всі цілі програмне забезпечення, встановлюють, які вимоги відомі, а які належить визначити додатково.

Потім виконується швидке проектування. У нім увага зосереджується на тих характеристиках програмного забезпечення, які мають бути видимі користувачеві.

Швидке проектування приводить до побудови макету.

Макет оцінюється замовником і використовується для уточнення вимог до програмного забезпечення.

Ітерації повторюються до тих пір, поки макет не виявить всі вимоги замовника і, тим самим, не дасть можливість розробникові зрозуміти, що має бути зроблене.

Переваги макетування: забезпечує визначення повних вимог до програмного забезпечення.

Недоліки макетування:

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

Пояснимо суть недоліків. Коли замовник бачить працюючу версію програмного забезпечення, він перестає усвідомлювати, що деталі макету скріпляють «жувальною гумкою і дротом»; він забуває, що в гонитві за

працюючим варіантом залишені невирішеними питання якості і зручності супроводу програмного забезпечення. Коли замовникові говорять, що продукт має бути перебудований, він починає обурюватися і вимагати, щоб макет «в три прийоми» був перетворений на робочий продукт. Дуже часто це негативно позначається на управлінні розробкою програмного забезпечення.

З іншого боку, для швидкого отримання працюючого макету розробник часто йде на певні компроміси. Можуть використовуватися не самі відповідні мова програмування або операційна система. Для простої демонстрації можливостей може застосовуватися неефективний алгоритм. Через деякий час розробник забуває про причини, по яких ці засоби не підходять. В результаті далеко не ідеальний вибраний варіант інтегрується в систему.

Очевидно, що подолання цих недоліків вимагає боротьби з життєвою спокусою — прийняти бажане за дійсне.

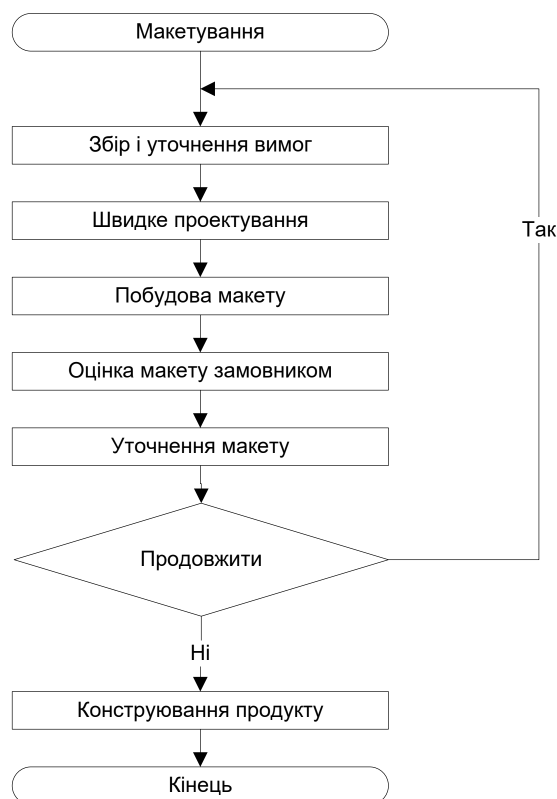

Рис. 1.3. Послідовність дій при макетуванні.

1.4. Стратегії конструювання програмного забезпечення.

Існують 3 стратегії конструювання програмне забезпечення:

- *одноразовий прохід* (стратегія водопаду) — лінійна послідовність етапів конструювання;
- *інкрементна стратегія.* На початку процесу визначаються всі користувацькі і системні вимоги, частина конструювання, що залишилася, виконується у вигляді послідовності версій. Перша версія реалізує частину запланованих можливостей, наступна версія реалізує додаткові можливості і т. д., поки не буде отримана повна система;

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

Характеристики стратегій конструювання програмного забезпечення відповідно до вимог стандарту IEEE/EIA 12207.2 приведені в таблиці. 1.1.

Таблиця 1.1

Характеристики стратегій конструювання

Стратегія конструювання	**На початку процесу визначені всі вимоги?**	**Безліч циклів конструювання?**	**Проміжне програмне забезпечення розповсюджується?**
Одноразовий прохід	Так	Немає	Немає
Інкрементна (заплановане поліпшення продукту)	Так	Так	Можливо
Еволюційна	Немає	Так	Так

1.5. Інкрементна модель.

Інкрементна модель є класичним прикладом інкрементної стратегії конструювання (рис. 1.4). Вона об'єднує елементи послідовної моделі водопаду з ітераційною філософією макетування.

Рис. 1.4. Інкрементна модель.

Кожна лінійна послідовність тут виробляє інкремент програмного забезпечення, що поставляється. Наприклад, програмне забезпечення для обробки слів в 1-му інкременті реалізує функції базової обробки файлів, функції редагування і документування; у 2-му інкременті — складніші можливості редагування і документування; у 3-му інкременті — перевірку орфографії і граматики; у 4-му інкременті — можливості компоновки сторінки.

Перший інкремент приводить до отримання базового продукту, що реалізовує базові вимоги (правда, багато допоміжних вимог залишаються

нереалізованими).

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

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

Забігаючи вперед, відзначимо, що сучасна реалізація інкрементного підходу — екстремальне програмування XP (Кент Бек, 1999). Воно орієнтоване на дуже малі прирости функціональності.

1.6. Швидка розробка застосувань.

Модель швидкої розробки додатків (Rapid Application Development) — другий приклад застосування інкрементної стратегії конструювання (рис. 1.5).

RAD-модель забезпечує екстремально короткий цикл розробки. RAD — високошвидкісна адаптація лінійної послідовної моделі, в якій швидка розробка досягається за рахунок використання компонентно-орієнтованого конструювання. Якщо вимоги повністю визначені, а проектна область обмежена, RAD-процесс дозволяє групі створити повністю функціональну систему за дуже короткий час (60-90 днів). RAD-підход орієнтований на розробку інформаційних систем і виділяє наступні етапи:

- **бізнес-моделювання.** Моделюється інформаційний потік між бізнес-функціями. Шукається відповідь на наступні питання: Яка інформація керує бізнес-процесом? Яка генерується інформація? Хто генерує її? Де інформація застосовується? Хто обробляє її?
- **моделювання даних**. Інформаційний потік, визначений на етапі бізнес-моделювання, відображається в набір об’єктів даних, які потрібні для підтримки бізнесу. Ідентифікуються характеристики (властивості, атрибути) кожного об’єкту, визначаються стосунки між об’єктами;
- **моделювання обробки.** Визначаються перетворення об’єктів даних, що забезпечують реалізацію бізнес-функцій. Створюються описи обробки для додавання, модифікації, видалення або знаходження (виправлення) об’єктів даних;
- **генерація застосування.** Передбачається використання методів, орієнтованих на мови програмування 4-го покоління. Замість створення програмного забезпечення за допомогою мов програмування 3-го покоління, RAD-процес працює з повторно використовуваними програмними компонентами або створює повторно використовувані компоненти. Для забезпечення конструювання використовуються утиліти автоматизації;
- **тестування і об'єднання.** Оскільки застосовуються повторно компоненти, що повторно використовуються, багато програмних елементів вже протестовано. Це зменшує час тестування (хоча все нові елементи мають бути протестовані).

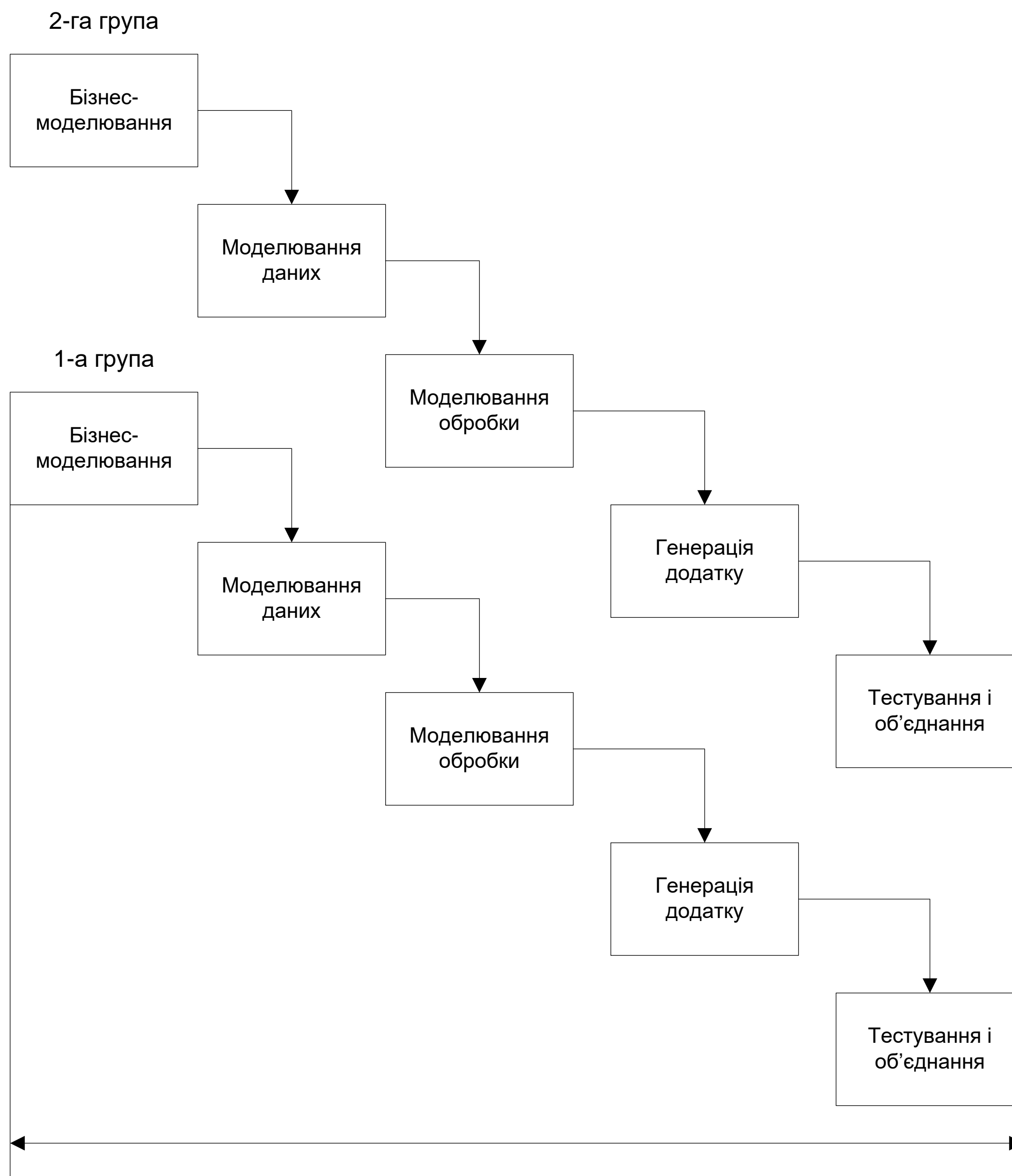

Рис. 1.5. Модель швидкої розробки застосувань.

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

Застосування RAD має і свої недоліки, і обмеження.

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

2. RAD можна застосовувати тільки для таких додатків, які можуть декомпозуватися на окремі модулі і в яких продуктивність не є критичною величиною.

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

1.7. Спіральна модель.

Спіральна модель — класичний приклад застосування еволюційної стратегії

конструювання.

Спіральна модель (автор Баррі Боем, 1988) базується на кращих властивостях класичного життєвого циклу і макетування, до яких додається новий елемент, — аналіз ризику, відсутній в цих парадигмах.

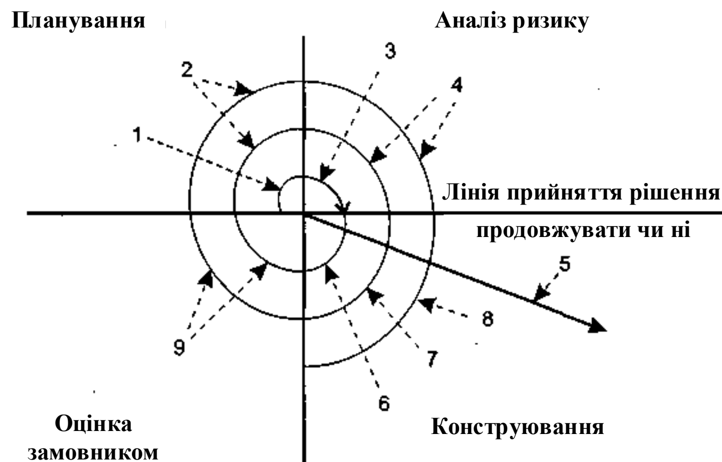

Рис. 1.6. Спіральна модель:

1 — початковий збір вимог і планування проекту;

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

3 — аналіз ризику на основі початкових вимог;

4 — аналіз ризику на основі реакції замовника;

5 — перехід до комплексної системи;

6 — початковий макет системи;

7 — наступний рівень макету;

8 — сконструйована система;

9 — оцінювання замовником.

Як показано на рис. 1.6, модель визначає чотири дії, що представляються чотирма квадрантами спіралі.

1. Планування — визначення цілей, варіантів і обмежень.
2. Аналіз ризику — аналіз варіантів і розпізнавання/вибір ризику.
3. Конструювання — розробка продукту наступного рівня.
4. Оцінювання — оцінка замовником поточних результатів конструювання.

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

У першому витку спіралі визначаються початкові цілі, варіанти і обмеження, розпізнається і аналізується ризик. Якщо аналіз ризику показує невизначеність вимог, на допомогу розробникові і замовникові приходить макетування (використовуване в квадранті конструювання). Для подальшого визначення проблемних і уточнених вимог може бути використане моделювання. Замовник оцінює інженерну (конструкторську) роботу і вносить пропозиції по модифікації (квадрант оцінки замовником). Наступна фаза планування і аналізу ризику базується на пропозиціях замовника. У кожному циклі по спіралі результати аналізу ризику формуються у вигляді

«продовжувати, не продовжувати». Якщо ризик дуже великий, проект може бути зупинений.

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

Переваги спіральної моделі:

1. Найреальніше (у вигляді еволюції) відображає розробку програмного забезпечення.
2. Дозволяє явно враховувати ризик на кожному витку еволюції розробки.
3. Включає крок системного підходу в ітераційну структуру розробки.
4. Використовує моделювання для зменшення ризику і вдосконалення програмного виробу.

Недоліки спіральної моделі:

1. Новизна (відсутня достатня статистика ефективності моделі).
2. Підвищені вимоги до замовника.
3. Труднощі контролю і управління часом розробки.

1.8. Компонентно-орієнтована модель.

Компонентно-орієнтована модель є розвитком спіральної моделі і теж ґрунтується на еволюційній стратегії конструювання. У цій моделі конкретизується зміст квадранта конструювання — воно відображає той факт, що в сучасних умовах нова розробка повинна ґрунтуватися на повторному використанні існуючих програмних компонентів (рис. 1.7).

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

Переваги компонентно-орієнтованої моделі:

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

1.9. Ваговиті і полегшені процеси.

Традиційно для впорядкування і прискорення програмних розробок пропонувалися ваговиті (heavyweight) процеси, що строго впорядковують. У цих процесах прогнозується весь обсяг майбутніх робіт, тому вони називаються прогнозуючими (predictive) процесами. Порядок, який повинен виконувати при цьому людина-розробник, надзвичайно строгий — «крок управо, крок вліво —

віртуальний розстріл!». Іншими словами, людські слабкості в розрахунок не приймаються, а обсяг необхідної документації здатний відняти спокій і сон у «сумлінного» розробника.

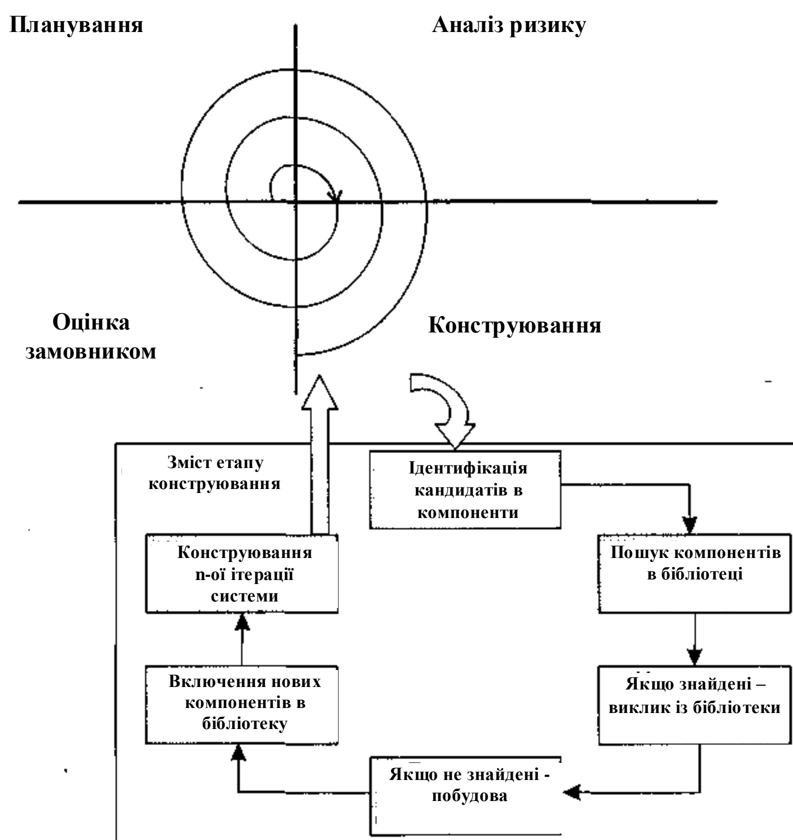

Рис. 1.7. Компонентно-орієнтована модель.

Останніми роками з’явилася група нових, полегшених (lightweight) процесів. Тепер їх називають рухомими (agile) процесами. Вони привабливі відсутністю бюрократизму, характерного для ваговитих (що прогнозують) процесів. Нові процеси повинні втілити в життя розумний компроміс між дуже строгою дисципліною і повною її відсутністю. Інакше кажучи, порядку в них достатньо для того, щоб отримати розумну віддачу від розробників.

Рухомі процеси вимагають меншого обсягу документації і орієнтовані на людину. У них явно вказано на необхідність використання природних якостей людської натури (а не на застосування дій, направлених наперекір цим якостям).

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

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

- сімейство прогнозуючих (ваговитих) процесів;
- сімейство адаптивних (рухомих, полегшених) процесів.

У кожного сімейства є свої переваги, недоліки і сфера застосування:

- адаптивний процес використовують при частих змінах вимог, нечисленній групі висококваліфікованих розробників і грамотному замовнику, який згоден брати участь в розробці;
- прогнозуючий процес застосовують при фіксованих вимогах і численній групі розробників різної кваліфікації.

1.10. XP-процесс.

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

Таблиця 1.2

Екстремуми в екстремальному програмуванні

Практика здорового глузду	**XP-екстремум**	**XP-реалізація**
Перевірки коду	Код перевіряється весь час	Парне програмування
Тестування	Тестування виконується весь час, навіть за допомогою замовників	Тестування модуля, функціональне тестування
Проектування	Проектування є частиною щоденної діяльності кожного розробника	Реорганізація (refactoring)
Простота	Для системи вибирається просте проектне рішення, що підтримує її поточну функціональність	Найпростіша річ, яка могла б працювати
Архітектура	Кожен постійно працює над уточненням архітектури	Метафора
Тестування інтеграції	Інтегрується і тестується кілька разів в день	Безперервна інтеграція
Короткі ітерації	Ітерації є гранично короткими, продовжуються секунди, хвилини, години, а не тижні, місяці або роки	Гра планування

Основна ідея XP — усунути високу вартість зміни, характерну для застосувань з використанням об'єктів, патернів і реляційних баз даних. Патерн є

рішенням типової проблеми у визначеному контексті. Тому XP-процесс має бути високодинамічним процесом. XP-група має справу із змінами вимог на всьому протязі ітераційного циклу розробки, причому цикл складається з дуже коротких ітерацій. Чотирма базовими діями в XP-циклі є: кодування, тестування, вислуховування замовника і проектування. Динамізм забезпечується за допомогою чотирьох характеристик: безперервного зв'язку із замовником (і в межах групи), простоти (завжди вибирається мінімальне рішення), швидкого зворотного зв'язку (за допомогою модульного і функціонального тестування), сміливості в проведенні профілактики можливих проблем.

Більшість принципів, що підтримується в XP (мінімальність, простота, еволюційний цикл розробки, мала тривалість ітерації, участь користувача, оптимальні стандарти кодування і т. д.), продиктовані здоровим глуздом і застосовуються в будь-якому впорядкованому процесі. Просто в XP ці принципи, як показано в таблиці. 1.2, досягають «екстремальних значень».

Той, хто приймає принцип «мінімального рішення» за хакерство, помиляється, насправді XP — строго впорядкований процес. Прості рішення, що мають вищий пріоритет, в даний час розглядаються як найбільш цінні частини системи, на відміну від проектних рішень, які поки не потрібні, а можуть (в умовах зміни вимог і операційного середовища) і взагалі не знадобитися.

Базис XP утворюють перераховані нижче дванадцять методів.

1. Гра планування (Planning game) — швидке визначення зони дії наступної реалізації шляхом об'єднання ділових пріоритетів і технічних оцінок. Замовник формує зону дії, пріоритетність і терміни з погляду бізнесу, а розробники оцінюють і простежують просування (прогрес).

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

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

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

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

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

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

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

9. Безперервна інтеграція (Continuous integration) — система інтегрується і будується багато раз в день, у міру завершення кожного завдання. Безперервне регресійне тестування, тобто повторення попередніх тестів,

гарантує, що зміни вимог не приведуть до регресу функціональності.

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

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

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

Гра планування і часта зміна версій залежать від замовника, що забезпечує набір «історій» (коротких описів), що характеризують роботу, яка виконуватиметься для кожної версії системи. Версії генеруються кожні два тижні, тому розробники і замовник повинні прийти до угоди про те, які історії будуть здійснені в межах двох тижнів. Повну функціональність, потрібну замовникові, характеризує пул історій; але для наступної двотижневої ітерації з пулу вибирається підмножина історій, найбільш важлива для замовника. У будь-який час в пул можуть бути додані нові історії, таким чином, вимоги можуть швидко змінюватися. Проте процеси двотижневої генерації засновані на найбільш важливих функціях, що входять в поточний пул, отже, мінливість управляється. Локальний замовник забезпечує підтримку цього стилю ітераційної розробки.

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

Парне програмування — один з найбільш спірних методів в XP, воно впливає на ресурси, що важливе для менеджерів, які вирішують, чи буде проект використовувати XP. Може здатися, що парне програмування подвоює ресурси, але дослідження довели: парне програмування приводить до підвищення якості і зменшення часу циклу. Для узгодженої групи витрати збільшуються на 15%, а час циклу скорочується на 40-50%. Для Інтернет-середовища збільшення швидкості продажів покриває підвищення витрат. Співпрацю покращує процес вирішення проблем, поліпшення якості істотно знижує витрати супроводу, які перевищують вартість додаткових ресурсів по всьому циклу розробки.

Колективне володіння означає, що будь-який розробник може змінювати

будь-який фрагмент коду системи у будь-який час. Безперервна інтеграція, безперервне регресійне тестування і парне програмування XP забезпечують захист від проблем, що виникають при цьому.

«Тестуй, а потім кодуй» — ця фраза виражає акцент XP на тестуванні. Вона відображає принцип, по якому спочатку планується тестування, а тестові варіанти розробляються паралельно аналізу вимог, хоча традиційний підхід полягає в тестуванні «чорного ящика». Роздум про тестування на початку циклу життя — добре відома практика конструювання програмне забезпечення (правда, рідко здійснювана практично).

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

При ухваленні XP рекомендується освоювати його методи поодинці, кожного разу вибираючи метод, орієнтований на найважчу проблему групи. Звичайно, всі ці методи є «не більше ніж правилами» — група може у будь-який момент поміняти їх (якщо її співробітники досягли принципової угоди з приводу внесених змін). Захисники XP визнають, що XP надає сильну соціальну дію, і не кожен може прийняти її. Разом з тим, XP — це методологія, що забезпечує переваги тільки при використанні закінченого набору базових методів.

Розглянемо структуру «ідеального» XP-процесу. Основним структурним елементом процесу є XP-реалізація, в яку багато разів вкладається базовий елемент — XP-ітерація. До складу XP-реалізації і XP-ітерації входять три фази — дослідження, блокування, регулювання. Дослідження (exploration) — це пошук нових вимог (історій, завдань), які повинна виконувати система. Блокування (commitment) — вибір для реалізації конкретної підмножини зі всіх можливих вимог (іншими словами, планування). Регулювання (steering) — проведення розробки, втілення плану в життя.

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

XP-процес для проекту з сім'ю реалізаціями, що здійснюється за 15 місяців, показаний на рис. 1.8.

Процес ініціюється початковою дослідницькою фазою.

Фаза дослідження, з якою починається будь-яка реалізація і ітерація, має клапан «пропуску», на цій фазі ухвалюється рішення про доцільність подальшого продовження роботи.

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

Найбільш важка перша реалізація — пройти за три місяці від звичайного

старту (скажімо, окремий співробітник не зафіксував ніяких вимог, не визначені обмеження) до постачання замовникові системи промислової якості дуже складно.

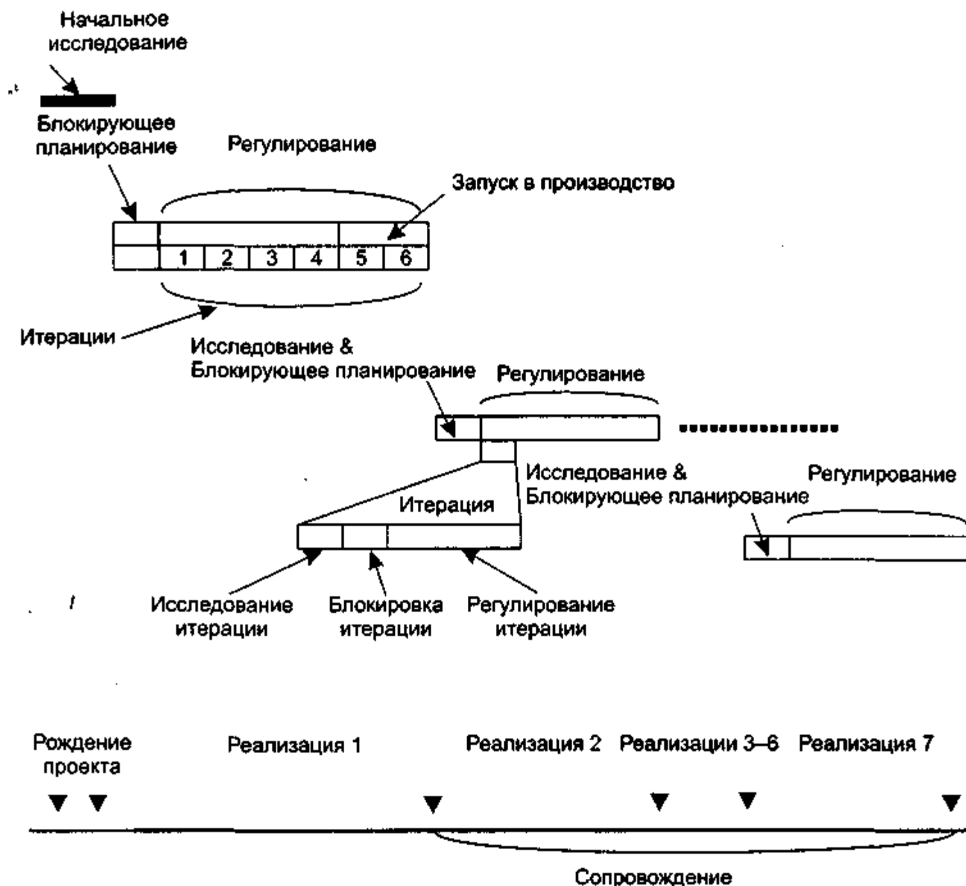

Рис. 1.8. Ідеальний ХР-процес.

Контрольні питання

1. Дайте визначення технології конструювання програмного забезпечення.
2. Які етапи класичного життєвого циклу ви знаєте?
3. Охарактеризуйте зміст етапів класичного життєвого циклу.
4. Поясните переваги і недоліки класичного життєвого циклу.
5. Чим відрізняється класичний життєвий цикл від макетування?
6. Які існують форми макетування?
7. Чим відрізняються один від одного стратегії конструювання програмного забезпечення?
8. Вкажіть схожість і відмінності класичного життєвого циклу і інкрементної моделі.
9. Поясните переваги і недоліки інкрементної моделі.
10. Чим відрізняється модель швидкої розробки додатків від інкрементної моделі?
11. Поясните переваги і недоліки моделі швидкої розробки застосувань.
12. Вкажіть схожість і відмінності спіральної моделі і класичного життєвого циклу.

13. У чому полягає головна особливість спіральної моделі?
14. Чим відрізняється компонентно-орієнтована модель від спіральної моделі і класичного життєвого циклу?
15. Перерахуєте переваги і недоліки компонентно-орієнтованої моделі.
16. Перерахуєте характеристики ХР-процесу.
17. Перерахуєте методи ХР-процесу.
18. У чому полягає головна особливість ХР-процесу?
19. Яка особливість проектування в ХР-процесі?
20. Яка особливість програмування в ХР-процесі?
21. Яка максимальна чисельність групи ХР-розробників?

РОЗДІЛ 2
ПАТЕРНИ ПРОЕКТУВАННЯ. ПОРОДЖУЮЧІ ПАТЕРНИ.

2.1. Поняття патернів проектування.

Проектування об'єктно-орієнтованих програм – складна справа, а якщо їх потрібно використовувати повторно, то все стає ще складніше. Необхідно підібрати відповідні об'єкти, віднести їх до різних класів, дотримуючись розумну ступінь деталізації, визначити інтерфейси класів і ієрархію наслідування і встановити суттєві відносини між класами. Дизайн повинен, з одного боку, відповідати розв'язуваної задачі, з іншого - бути загальним, щоб вдалося врахувати всі вимоги, які можуть виникнути в майбутньому.

Перш за все, досвідченому розробнику зрозуміло, що не потрібно вирішувати кожну нову задачу з нуля. Замість цього він намагається повторно скористатися тими рішеннями, які виявилися вдалими в минулому. Відшукавши гарне рішення один раз, він буде вдаватися до нього знову і знову. Саме завдяки накопиченому досвіду проектувальник і стає експертом в своїй області. У багатьох об'єктно-орієнтованих системах ви зустрінете повторювані патерни, що складаються з класів і взаємодіючих об'єктів. З їх допомогою вирішуються конкретні завдання проектування, в результаті чого об'єктно-орієнтований дизайн стає більш гнучким, елегантним, і їм можна скористатися повторно.

Патерни проектування спрощують повторне використання вдалих проектних і архітектурних рішень. За допомогою патернів можна поліпшити якість документації і супроводу існуючих систем, дозволяючи явно описати взаємодії класів і об'єктів, а також причини, за якими система була побудована так, а не інакше. Простіше кажучи, патерни проектування дають розробнику можливість швидше знайти «правильний» шлях.

2.2. Характеристика породжуючих патернів.

Мабуть, створення нових об'єктів є найбільш поширеним завданням, що встає перед розробниками програмних систем. Породжуючі патерни проектування призначені для створення об'єктів, дозволяючи системі залишатися незалежною як від самого процесу породження, так і від типів породжуваних об'єктів.

Породжуючи патерни проектування абстрагують процес інстанціонування. Вони допоможуть зробити систему незалежною від способу створення, композиції та представлення об'єктів. Патерн, який породжує класи, використовує спадкування, щоб варіювати інстанційований клас, а патерн, який породжує об'єкти, делегує інстанціюваня іншому об'єкту.

Ці патерни виявляються важливі, коли система більше залежить від композиції об'єктів, ніж від успадкування класів. Виходить так, що основний акцент робиться не на жорсткому кодуванні фіксованого набору поведінок, а на визначенні невеликого набору фундаментальних поведінок, за допомогою композиції яких можна отримувати будь-яке число складніших. Таким чином,

для створення об'єктів з конкретною поведінкою потрібно щось більше, ніж просте інстанціонування класу.

Для породжуючих патернів актуальні дві теми. По-перше, ці патерни інкапсулюють знання про конкретні класи, які застосовуються в системі. По-друге, приховують деталі того, як ці класи створюються і стикуються. Єдина інформація про об'єкти, відома системі, - це їх інтерфейси, визначені за допомогою абстрактних класів. Отже, породжуючі патерни забезпечують більшу гнучкість при вирішенні питання про те, що створюється, хто це створює, як і коли. Можна зібрати систему з «готових» об'єктів з самої різною структурою і функціональністю статично (на етапі компіляції) або динамічно (під час виконання).

2.3. Патерн Singleton.

Призначення. Патерн гарантує, що у класу є тільки один екземпляр, і надає до нього глобальну точку доступу.

Мотивація. Часто в системі можуть існувати об’єкти тільки в єдиному екземплярі, наприклад, система ведення системного журналу повідомлень або драйвер дисплея. У таких випадках необхідно вміти створювати єдиний екземпляр деякого типу, надавати до нього доступ ззовні і забороняти створення декількох екземплярів того ж типу.

Глобальна змінна дає доступ до об'єкта, але не забороняє інстанціювати клас в декількох примірниках.

Більш вдале рішення - сам клас контролює те, що у нього є тільки один екземпляр, може заборонити створення додаткових примірників, перехоплюючи запити на створення нових об'єктів, і він же здатний надати доступ до свого екземпляру. Це і є призначення патерну одинак.

Структура.

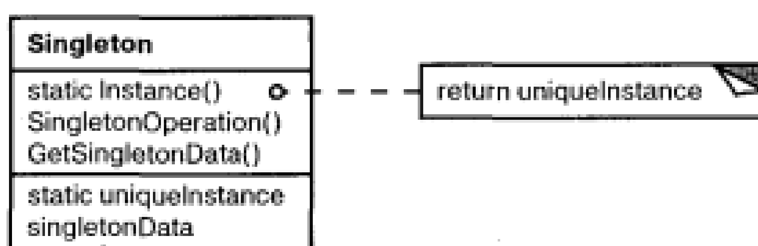

Рис. 2.1. Структура патерну Singleton.

Патерн визначає операцію Intance, яка дозволяє клієнтам отримувати доступ до єдиного екземпляру. Може нести відповідальність за створення власного унікального екземпляра. Клієнти отримують доступ до примірника класу Singleton тільки через його операцію Intance.

Застосування. Використовуйте патерн одинак, коли:

- повинен бути рівно один екземпляр деякого класу, легко доступний всім клієнтам;
- єдиний екземпляр повинен розширюватися шляхом породження

підкласів, і клієнтам потрібно мати можливість працювати з розширеним екземпляром без модифікації свого коду.

Учасники:

Singleton - одинак: визначає операцію Instance, яка дозволяє клієньам отримувати доступ до єдиного екземпляру. Може нести відповідальність за створення власного унікального екземпляру. Клієнти отримують доступ до екземпляру класу Singleton тільки через його операцію Instance.

Результати. У патерна одинак є певні переваги:

- контрольований доступ до єдиного екземпляру. Оскільки клас інкапсулює свій єдиний екземпляр, він повністю контролює те, як і коли клієнти отримують доступ до нього;
- зменшення числа імен. Патерн одинак – це крок вперед у порівнянні з глобальними змінними. Він дозволяє уникнути засмічення простору імен глобальними змінними, в яких зберігаються унікальні екземпляри;
- допускає уточнення операцій та подання. Від класу Singleton можна породжувати підкласи, а додаток легко конфігурувати екземпляром розширеного класу. Можна конкретизувати додаток екземпляром того класу, який необхідний під час виконання;
- допускає змінне число примірників. Патерн дозволяє вам легко змінити своє рішення і дозволити появу більш одного екземпляра класу Singleton. Ви можете застосовувати один і той же підхід для управління числом примірників, які використовуються в додатку. Змінити потрібно буде лише операцію, що дає доступ до примірника класу Singleton;
- велика гнучкість, ніж у операцій класу. Ще один спосіб реалізувати функціональність одинака - використовувати операції класу, тобто статичні функції-члени. Але обидва цих прийоми перешкоджають зміні дизайну, якщо буде потрібно дозволити наявність кількох екземплярів класу. Крім того, статичні функції-члени в C ++ не можуть бути віртуальними, так що їх не можна поліморфно замістити в підкласах.

Реалізація.

Розглянемо реалізацію патерна Singleton, яка найчастіше зустрічається.

```
// Singleton.h
class Singleton
{
  private:
    static Singleton * _instance;
    // Конструктори і оператор присвоювання недоступні клієнтам
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * Instance();

// Singleton.cpp
#include "Singleton.h"
```

```
Singleton* Singleton::_instance = 0;
Singleton * Singleton::Instance() {
      if(!_instance)
         _instance = new Singleton();
      return _instance;
}
```

Змінна _instance ініціалізується нулем, а статична функція-член Instance повертає її значення, ініціалізувавши її унікальним екземпляром, якщо в поточний момент воно дорівнює 0. Функція Instance використовує відкладену ініціалізацію: значення не створюється і не зберігається аж до моменту першого звернення. Зверніть увагу, що конструктор захищений. Клієнт, який спробує інстанціювати клас безпосередньо, отримає помилку на етапі компіляції. Це дає гарантію, що буде створений тільки один екземпляр.

Далі, оскільки _instance – вказує на об'єкт класу, то функція-член може привласнити цій змінній вказівник на будь-який підклас даного класу.

Між одинаками не може існувати ніяких залежностей. Якщо вони є, то помилок не уникнути. Ще один (хоча і не дуже серйозний) недолік глобальних / статичних об'єктів в тому, що доводиться створювати всіх одинаків, навіть якщо вони не використовуються. Застосування статичної функції-члена вирішує цю проблему.

Те, що клієнти самостійно повинні виконати delete одинаку є недоліком. Так як клас сам контролює створення єдиного об'єкта, було б логічним покласти на нього відповідальність і за руйнування об'єкта. Цей недолік відсутній в реалізації Singleton, вперше запропонованої Скоттом Мейерсом.

```
// Singleton.h
class Singleton
{
private:
   Singleton() {}
   Singleton( const Singleton&);
   Singleton& operator=( Singleton& );
public:
   static Singleton& Instance() {
      static Singleton  instance;
      return instance;
   }
};
```

Статична функція-член Instance () повертає не вказівку, а посилання на цей об'єкт, тим самим, ускладнюючи можливість помилкового звільнення пам'яті клієнтами.

На жаль, у реалізації Мейерса є недоліки: складність створення об'єктів похідних класів і неможливість безпечного доступу декількох клієнтів до єдиного об'єкту в багатопотоковому середовищі.

З урахуванням усього вищесказаного класична реалізація патерну Singleton

може бути поліпшена.

```
// Singleton.h
class Singleton;

class SingletonDestroyer
{
  private:
    Singleton* _instance;
  public:
    SingletonDestroyer() { _instance = 0; }
    ~SingletonDestroyer();
    void initialize( Singleton* p );
};

class Singleton
{
  private:
    static Singleton* _instance;
    static SingletonDestroyer destroyer;
  protected:
    Singleton() { }
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
   ~Singleton() { }
    friend class SingletonDestroyer;
  public:
    static Singleton& Instance();
};

// Singleton.cpp
#include "Singleton.h"

Singleton * Singleton::_instance = 0;
SingletonDestroyer Singleton::destroyer;

SingletonDestroyer::~SingletonDestroyer() {
    delete _instance;
}
void SingletonDestroyer::initialize( Singleton* p ) {
    _instance = p;
}
Singleton& Singleton::Instance() {
    if(!_instance)     {
        _instance = new Singleton();
        destroyer.initialize(_instance);
```

```
    }
    return *_instance;
}
```

Ключовою особливістю цієї реалізації є наявність класу SingletonDestroyer, призначеного для автоматичного руйнування об'єкта Singleton. Клас Singleton має статичний член SingletonDestroyer, що інстанціюється при першому виклику Singleton :: Instance () створюваним об'єктом Singleton. При завершенні програми цей об'єкт буде автоматично зруйнований деструктором SingletonDestroyer (для цього SingletonDestroyer оголошений дружнім класу Singleton).

Для запобігання випадкового видалення користувачами об'єкта класу Singleton, деструктор тепер уже не є загальнодоступним як раніше. Він оголошений захищеним.

До сих пір передбачалося, що в програмі використовується один одинак або кілька незв'язаних між собою. При використанні взаємопов'язаних одинаків з'являються нові питання:

Як гарантувати, що до моменту використання одного одинака, екземпляр іншого залежного вже створено?

Як забезпечити можливість безпечного використання одного одинака іншим при завершенні програми? Іншими словами, як гарантувати, що в момент руйнування першого одинака в його деструкції ще можливе використання другого залежного одинака (тобто другий одиночка до цього моменту ще не зруйнований)?

Управляти порядком створення одинаків просто. Наступний код демонструє один з можливих методів.

```
class Singleton1
{
private:
  Singleton1() { }
  Singleton1(const Singleton1&);
  Singleton1& operator=(Singleton1&);
public:
  static Singleton1& Instance() {
          static Singleton1 instance;
          return instance;
  }
};

class Singleton2
{
private:
  Singleton2(Singleton1& instance) : s1(instance) {}
  Singleton2(const Singleton2&);
  Singleton2& operator=(Singleton2&);
  Singleton1& s1;
```

```
public:
  static Singleton2& Instance() {
          static Singleton2 instance(Singleton1::Instance());
          return instance;
  }
};


int main()
{
  Singleton2& s = Singleton2::Instance();
  return 0;
}
```

Проблема успадкування

Якщо існує необхідність наслідувати від класу Singleton, то слід дотримуватися певних правил.

По-перше, клас-спадкоємець повинен перевизначити метод Instance (), так, щоб створювати екземпляр похідного класу. Якщо не передбачається, що вказівка буде використовуватися поліморфно, то можна оголосити тип повернення методу Instance () як вказівку на клас-спадкоємець, в іншому випадку, метод Instance () повинен повертати вказівку на базовий клас (Singleton).

По-друге, в базовому класі деструктор повинен бути оголошений як віртуальний: в певний момент клієнт викликає метод FreeInst для вказівки на базовий клас. Оскільки метод FreeInst зводиться до оператора delete this, то в разі, якщо деструктор не віртуальний, буде викликаний деструктор базового класу, але не буде викликаний деструктор класу-нащадка. Щоб уникнути такої ситуації, слід явно оголосити деструктор базового класу віртуальним.

По-третє, конструктор класу-нащадка також повинен бути оголошений в захищеній секції, щоб уникнути можливості створення об'єкта класу безпосередньо, минаючи метод Instance ().

```
class Singleton
{
protected:
  static Singleton* _self;
  static int _refcount;
  Singleton(){}
  virtual ~Singleton()  {printf ("~Singleton\n");}
public:
  static Singleton* Instance();
  void FreeInst() ;
};

class SinglImpl: public Singleton
```

```
{
protected:
  SinglImpl(){}
  ~SinglImpl() { printf ("~SinglImpl\n"); }
public:
  static Singleton* Instance()
  {
    if(!_self) _self = new SinglImpl();
    _refcount++;
    return _self;
  }
};

void  main()
{
  Singleton *p = SinglImpl::Instance();
  ...
  ...
  ...
  p->FreeInst();
}
```

Іноді може виникнути ситуація, при якій клієнт повинен поліморфно працювати з об'єктами, що мають загальний базовий клас, але деякі з них реалізують патерн Singleton, а деякі ні. Проблема виникає в момент звільнення об'єктів, так як у простих класів немає механізму відстеження посилань, а у класів, що реалізують Singleton, він є. При виклику методу FreeInst () через вказівку на базовий клас буде викликатися FreeInst () базового класу, що не має поняття про підрахунок посилань. Це призведе і до безумовного видалення об'єктів "Singleton" з пам'яті. Для запобігання такої поведінки слід оголосити віртуальним метод FreeInst () в базовому класі і реалізувати специфічну поведінку методу для класів Singleton. Реалізація FreeInst () в базовому класі надає механізм видалення об'єктів, які не є Singleton'ами.

Реалізація класу Singleton, в такому випадку, можлива за допомогою шаблонів мови C ++. Перевага такого підходу полягає в автоматичній параметризації методу instance (), що призводить до відсутності необхідності перевизначати його в класах нащадках. Конструктор класу-нащадка також повинен бути оголошений захищеним, а деструктор віртуальним. Крім того, базовий клас Singleton повинен бути оголошений одним класу спадкоємця, оскільки метод instance () базового класу в цій моделі створює об'єкт похідного класу.

```
template <class T>
class Singleton
{
protected:
  static T* _instance;
```

```
  static int _refcount;
  Singleton() {}
  virtual ~Singleton() { }
public:
  static T* instance();
  void free();
};

template <class T> T*  Singleton<T>::_instance = NULL;
template <class T> int Singleton<T>::_refcount = 0;


template <class T>
T*  Singleton<T>::instance()
{
  if (!_instance) {
          _instance = new T;
  }
  _refcount++;
  return _instance;
}

template <class T>
void  Singleton<T>::free()
{
  if (--_refcount == 0)
          delete this;
}

class Derived : public Singleton<Derived>
{
protected:
  Derived() {}
  friend class Singleton<Derived>;
};

int main()
{
  Derived *p = Derived::instance();

  p->free();
  return 0;
}
```

Класи, об'єкти яких повинні існувати в єдиному екземплярі, просто успадковуються від шаблонного класу Singleton.

Таким чином, У класі Singleton конструктор повинен бути оголошений в захищеній секції для запобігання створенню об'єкта способом, відмінним від виклику методу Instance (). Деструктор також слід помістити в захищену секцію класу, щоб виключити можливість видалення об'єкта оператором delete.

Для автоматичного підрахунку посилань при звільненні об'єкта слід використовувати особливий метод, такий як FreeInst ().

При спадкуванні від класу, що реалізує патерн Singleton, конструктор класу-нащадка також повинен бути оголошений в захищеній секції. Деструктор має бути оголошений як віртуальний. Клас-нащадок повинен перевизначити метод Instance (), так, щоб він створював об'єкт потрібного типу.

Якщо передбачається поліморфна робота з класами, успадкованими від одного базового класу, причому деякі класи нащадки реалізують патерн Singleton, а деякі ні, слід в базовому класі визначити метод FreeInst () як віртуальний. Базовий клас надає реалізацію за замовчуванням цього методу, просто викликаючи оператор delete this.

У класах-нащадках, що реалізують патерн Singleton, при реалізації методу FreeInst (), використовуйте механізм підрахунку посилань.

Якщо використовується парамеьризована версія Singleton, то в похідних класах слід оголосити базовий клас (Singleton) дружнім.

2.4. Патерн Abstract Factory.

Призначення. Надає інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів не специфікуючи їх конкретних класів. Відомий також під іменем Kit.

Мотивація. Розглянемо інструментальну програму для створення призначеного для користувача інтерфейсу, що підтримує різні стандарти зовнішнього вигляду. Зовнішній вигляд визначає візуальне уявлення і поведінку елементів призначеного для користувача інтерфейсу («віджетів») - смуг прокрутки, вікон і кнопок. Щоб додаток можна було перенести на інший стандарт, в ньому не повинен бути жорстко закодований зовнішній вигляд віджетів. Якщо інстанціювання класів для конкретного зовнішнього вигляду розкидано по всьому додатком, то змінити вигляд згодом буде нелегко.

Ми можемо вирішити цю проблему, визначивши абстрактний клас WidgetFactory, в якому оголошено інтерфейс для створення всіх основних видів віджетів. Є також абстрактні класи для кожного окремого виду і конкретні підкласи, що реалізують віджети з певним зовнішнім виглядом. В інтерфейсі WidgetFactory є операція, яка повертає новий об'єкт-віджет для кожного абстрактного класу віджетів. Клієнти викликають ці операції для отримання примірників віджетів, але при цьому нічого не знають про те, які саме класи використовують. Тому, клієнти залишаються незалежними від обраного стандарту зовнішнього вигляду.

Для кожного стандарту зовнішнього вигляду існує певний підклас WidgetFactory. Кожен такий підклас реалізує операції, необхідні для створення відповідного стандарту віджета Клієнти створюють віджети, користуючись

виключно інтерфейсом WidgetFactory, і їм нічого не відомо про класи, що реалізують віджети для конкретного стандарту. Іншими словами, клієнти повинні лише дотримуватися інтерфейсу, визначеного абстрактним, а не конкретним класом.

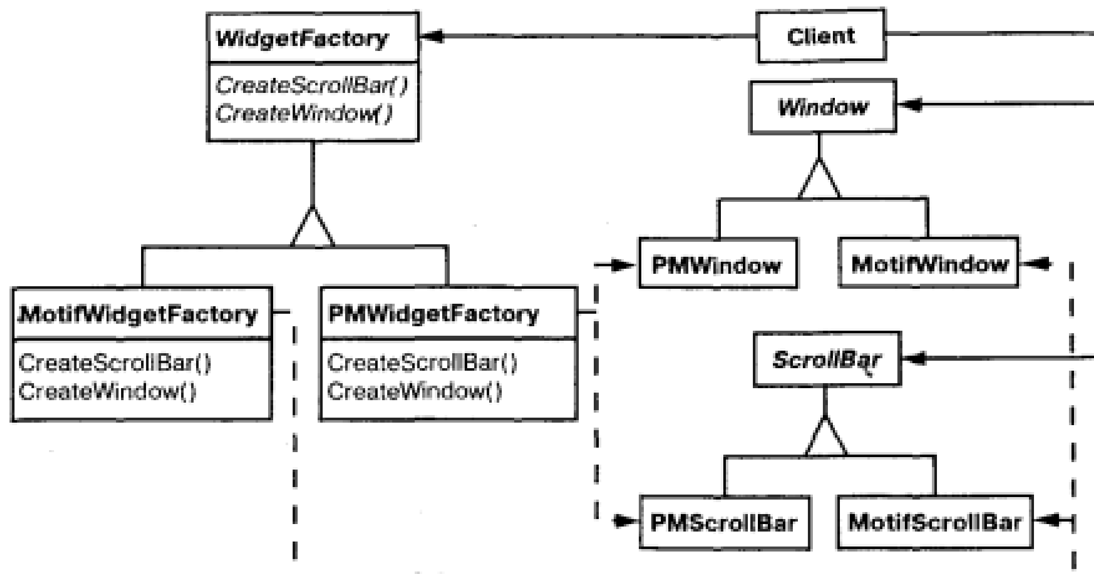

Рис. 2.2. Приклад взаємодії класів у патепні Abstract Factory

Застосування. Використовуйте патерн, коли:

- система не повинна залежати від того, як створюються, компонуються і представляються об'єкти, що входять до неї;
- взаємопов'язані об'єкти, що входять в сімейство, повинні використовуватися разом і вам необхідно забезпечити виконання цього обмеження;
- система повинна конфігуруватися одним з сімейств складових її об'єктів; необхідно створювати групи або сімейства взаємопов'язаних об'єктів, виключаючи можливість одночасного використання об'єктів з різних родин в одному контексті;
- необхідно надати бібліотеку об'єктів, розкриваючи тільки їх інтерфейси, але не реалізацію.

Учасники:

- **AbstractFactory** (WidgetFactory) – абстрактна фабрика: робить об’яву інтерфейсудля операцій, що створюють абстрактні об’єкти-продукти;
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory) – конкретна фабрика: реалізує операції, що створюють конкретні об’єкти-продукти;
- **AbstractProduct** (Window, Scrollbar) – абстрактний продукт: робить об’яву інтерфейсу для конкретного об’єкту-продукту;
- **ConcreteProduct** (Motif Window, MotifScrollbar) – конкретний продукт: визначає об’єкт-продукт, що створюється конкретною фабрикою. Реалізує інтерфейс AbstractProduct;
- **Client** – клієнт: використовує лише інтерфейси, які об’явлені в класах AbstractFactory та AbstractProduct.

Структура.

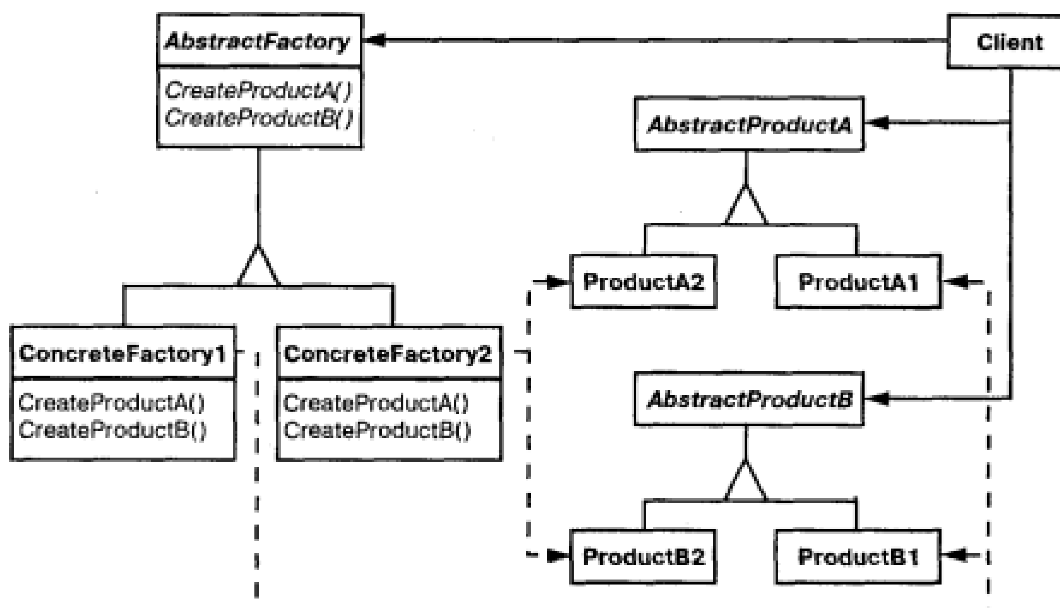

Рис. 2.3. Структура патерну Abstract Factory

Результати.

Патерн володіє наступними плюсами і мінусами:

- ізолює конкретні класи. Допомагає контролювати класи об'єктів, що створюються додатком. Оскільки фабрика інкапсулює відповідальність за створення класів і сам процес їх створення, то вона ізолює клієнта від деталей реалізації класів. Клієнти маніпулюють екземплярами через їх абстрактні інтерфейси. Імена класів відомі тільки конкретній фабриці, в коді клієнта вони не згадуються;
- спрощує заміну сімейств продуктів. Клас конкретної фабрики з'являється в додатку тільки один раз: при інстанціюванні. Це полегшує заміну використовуваної додатком конкретної фабрики. Додаток може змінити конфігурацію продуктів, просто підставивши нову конкретну фабрику. Оскільки абстрактна фабрика створює все сімейство продуктів, то і замінюється відразу все сімейство.;
- гарантує сполучуваність продуктів. Якщо продукти деякого сімейства спроектовані для спільного використання, то важливо, щоб додаток в кожен момент часу працював тільки з продуктами єдиного сімейства;
- підтримати новий вид продуктів важко. Розширення абстрактної фабрики для виготовлення нових видів продуктів - непросте завдання.

Реалізація.

Будь-яке сімейство або група взаємопов'язаних об'єктів характеризується кількома загальними типами створюваних продуктів, при цьому самі продукти таких типів будуть різними для різних сімейств. Наприклад, для випадку стратегічної гри загальними типами створюваних продуктів будуть піхота, лучники і кіннота.

Для того щоб система залишалася незалежною від специфіки того чи іншого сімейства продуктів необхідно використовувати загальні інтерфейси для всіх

основних типів продуктів. У разі стратегічної гри це означає, що необхідно використовувати три абстрактних базових класу для кожного типу воїнів: піхоти, лучників і кінноти. Похідні від них класи будуть реалізовувати специфіку відповідного типу воїнів тієї чи іншої армії.

Для вирішення завдання зі створення сімейств взаємопов'язаних об'єктів патерн Abstract Factory вводить поняття абстрактної фабрики. Абстрактна фабрика представляє собою деякий поліморфний базовий клас, призначенням якого є оголошення інтерфейсів фабричних методів, що служать для створення продуктів всіх основних типів (один фабричний метод на кожен тип продукту). Похідні від нього класи, що реалізують ці інтерфейси, призначені для створення продуктів всіх типів всередині сімейства або групи. У разі нашої гри базовий клас абстрактної фабрики повинен визначати інтерфейс фабричних методів для створення піхотинців, лучників і кінноти, а два похідних від нього класу будуть реалізовувати цей інтерфейс, створюючи воїнів усіх родів військ для тієї чи іншої армії.

```
#include <iostream>
#include <vector>
using namespace std;


// абстрактні базові класи усих можливих видів воїнів
class Infantryman
{
public:
  virtual void info() = 0;
  virtual ~Infantryman() {}
};

class Archer
{
public:
  virtual void info() = 0;
  virtual ~Archer() {}
};

class Horseman
{
public:
  virtual void info() = 0;
  virtual ~Horseman() {}
};


// класи всіх видів воїнів Римської імперії
class RomanInfantryman : public Infantryman
```

```
{
public:
  void info() {
           cout << "RomanInfantryman" << endl;
  }
};

class RomanArcher : public Archer
{
public:
  void info() {
           cout << "RomanArcher" << endl;
  }
};

class RomanHorseman : public Horseman
{
public:
  void info() {
           cout << "RomanHorseman" << endl;
  }
};


// класи всіх видів воїнів армії Карфагена
class CarthaginianInfantryman : public Infantryman
{
public:
  void info() {
           cout << "CarthaginianInfantryman" << endl;
  }
};

class CarthaginianArcher : public Archer
{
public:
  void info() {
           cout << "CarthaginianArcher" << endl;
  }
};

class CarthaginianHorseman : public Horseman
{
public:
  void info() {
```

```
            cout << "CarthaginianHorseman" << endl;
  }
};


// абстрактна фабрика для виробництва  воїнів
class ArmyFactory
{
public:
  virtual Infantryman* createInfantryman() = 0;
  virtual Archer* createArcher() = 0;
  virtual Horseman* createHorseman() = 0;
  virtual ~ArmyFactory() {}
};


// Фабрика для создания воинов Римской армии
class RomanArmyFactory : public ArmyFactory
{
public:
  Infantryman* createInfantryman() {
            return new RomanInfantryman;
  }
  Archer* createArcher() {
            return new RomanArcher;
  }
  Horseman* createHorseman() {
            return new RomanHorseman;
  }
};


// Фабрика для створення воїнів армії Карфагена
class CarthaginianArmyFactory : public ArmyFactory
{
public:
  Infantryman* createInfantryman() {
            return new CarthaginianInfantryman;
  }
  Archer* createArcher() {
            return new CarthaginianArcher;
  }
  Horseman* createHorseman() {
            return new CarthaginianHorseman;
  }
```

```
};


// клас воїнів однієї та іншої імперії
class Army
{
public:
  ~Army() {
          unsigned int i;
          for (i = 0; i < vi.size(); ++i)  delete vi[i];
          for (i = 0; i < va.size(); ++i)  delete va[i];
          for (i = 0; i < vh.size(); ++i)  delete vh[i];
  }
  void info() {
          unsigned int i;
          for (i = 0; i < vi.size(); ++i)  vi[i]->info();
          for (i = 0; i < va.size(); ++i)  va[i]->info();
          for (i = 0; i < vh.size(); ++i)  vh[i]->info();
  }
  vector<Infantryman*> vi;
  vector<Archer*> va;
  vector<Horseman*> vh;
};

class Game
{
public:
  Army* createArmy(ArmyFactory& factory) {
          Army* p = new Army;
          p->vi.push_back(factory.createInfantryman());
          p->va.push_back(factory.createArcher());
          p->vh.push_back(factory.createHorseman());
          return p;
  }
};


int main()
{
  Game game;
  RomanArmyFactory ra_factory;
  CarthaginianArmyFactory ca_factory;

  Army * ra = game.createArmy(ra_factory);
  Army * ca = game.createArmy(ca_factory);
```

```
  cout << "Roman army:" << endl;
  ra->info();
  cout << "\nCarthaginian army:" << endl;
  ca->info();
  // ...
}
```

Наступний приклад C++ ілюструє, як отримати різні типи об’єктів одного (гіпотетичного) сімейства GUI:

```
#include <iostream>
#include <string>

/* Abstract definitions */
class GUIComponent {
public:
  virtual ~GUIComponent() = default;
  virtual void draw() const = 0;
};
class Frame : public GUIComponent {};
class Button : public GUIComponent {};
class Label : public GUIComponent {};

class GUIFactory {
public:
  virtual ~GUIFactory() = default;
  virtual Frame * createFrame() = 0;
  virtual Button * createButton() = 0;
  virtual Label * createLabel() = 0;
  static GUIFactory* create(const std::string& type);
};

/* Windows support */
class WindowsFactory : public GUIFactory {
private:
  class WindowsFrame : public Frame {
  public:
        void draw() const override { std::cout << "I'm a Windows-like frame" <<
std::endl; }
  };
  class WindowsButton : public Button {
  public:
        void draw() const override { std::cout << "I'm a Windows-like button" <<
std::endl; }
  };
  class WindowsLabel : public Label {
  public:
```

```
            void draw() const override { std::cout << "I'm a Windows-like label" <<
std::endl; }
        };
    public:
        Frame* createFrame() { return new WindowsFrame(); }
        Button * createButton() { return new WindowsButton(); }
        Label * createLabel() { return new WindowsLabel(); }
    };

    /* Linux support */
    class LinuxFactory : public GUIFactory {
    private:
        class LinuxFrame : public Frame {
        public:
            void draw() const override { std::cout << "I'm a Linux-like frame" <<
std::endl; }
        };
        class LinuxButton : public Button {
        public:
            void draw() const override { std::cout << "I'm a Linux-like button" <<
std::endl; }
        };
        class LinuxLabel : public Label {
        public:
            void draw() const override { std::cout << "I'm a Linux-like label" <<
std::endl; }
        };
    public:
        Frame* createFrame() { return new LinuxFrame(); }
        Button * createButton() { return new LinuxButton(); }
        Label * createLabel() { return new LinuxLabel(); }
    };

    GUIFactory* GUIFactory::create(const std::string& type) {
        if (type == "windows") return new WindowsFactory();
        return new LinuxFactory();
    }

    /* User code */
    void buildInterface(GUIFactory& factory) {
        auto frame = factory.createFrame();
        auto button = factory.createButton();
        auto label = factory.createLabel();

        frame->draw();
```

```
  button->draw();
  label->draw();
}


int main()
{
  auto guiFactory = GUIFactory::create("windows");
  buildInterface(*guiFactory);
  return 0;
}
```

2.5. Патерн Factory Method.

Призначення. Визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласів. Відомий також під ім'ям Virtual Constructor (віртуальний конструктор).

Мотивація. Каркаси користуються абстрактними класами для визначення та підтримки відносин між об'єктами. Крім того, каркас часто відповідає за створення самих об'єктів.

Розглянемо каркас для додатків, здатних представляти користувачеві кілька документів. Дві основні абстракції в такому каркасі - це класи Application і Document. Обидва класи абстрактні, тому клієнти повинні породжувати від них підкласи для створення специфічних додатків для реалізації. Наприклад, щоб створити програму для малювання, ми визначимо класи DrawApplication та DrawDocument. Клас Application відповідає за управління документами і створює їх у міру необхідності, припустимо, коли користувач вибирає з меню пункт Open (відкрити) або New (створити).

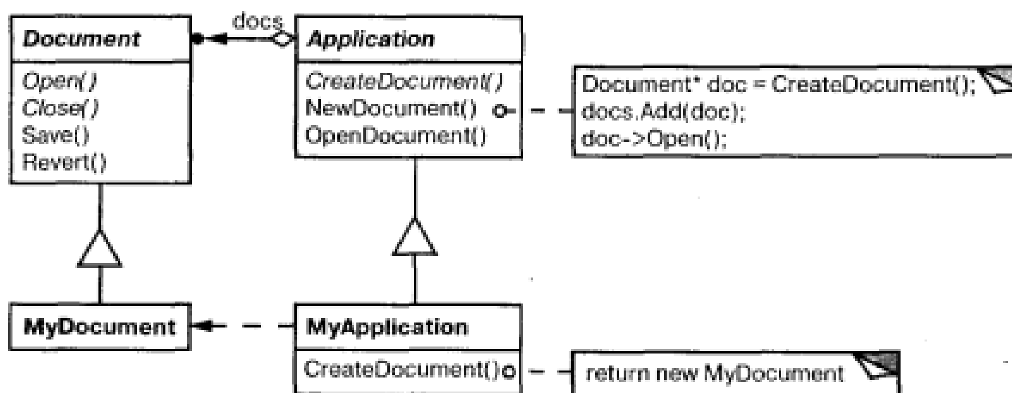

Рис. 2.4. Приклад взаємодії класів в патерні Factory Method

Оскільки рішення про те, який підклас класу Document інстанціювати, залежить від додатку, то Application не може «передбачити», що саме знадобиться. Цьому класу відомо лише, коли потрібно інстанціювати новий

документ, а не який документ створити. Виникає дилема: каркас має інстанціювати класи, але «знає» він лише про абстрактні класи, які інстанціювати не можна.

Рішення пропонує патерн фабричний метод. У ньому інкапсулюється інформація про те, який підклас класу Document створити, і це знання виводиться за межі каркасу.

Застосування. Фабричний метод використовують у таких випадках:

- класу заздалегідь невідомо, об'єкти яких підкласів йому треба створювати;
- клас спроектований так, щоб об'єкти, які він створює, специфіковані підкласами;
- клас делегує свої обов'язки одному з декількох допоміжних підкласів, і планується локалізувати дані про те, який клас приймає ці обов'язки на себе.

Структура.

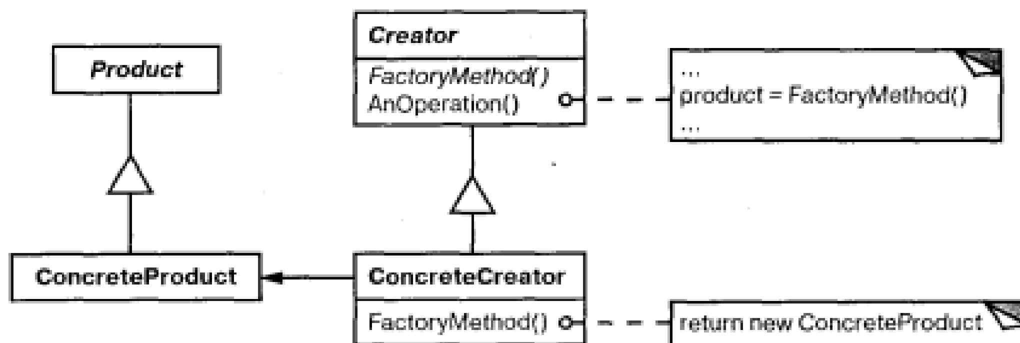

Рис. 2.5. Структура патерну Factory Method

Учасники:

- **Product** (Document) – продукт: робить об'яву інтерфейсу, що створюються фабричним методом;
- **ConcreteProduct** (MyDocument) – конкретний продукт: реалізує інтерфейс Product;
- **Creator** (Application) – творець: оголошує фабричний метод, що повертає об'єкт типу Product; може також визначати реалізацію за умовчанням фабричного методу, який повертає об'єкт ConcreteProduct; може викликати фабричний метод створення об'єкта Product;
- **ConcreteCreator** (MyApplication) – конкретний творець: Заміщає фабричний метод, що повертає об'єкт ConcreteProduct.

Результати. Фабричні методи позбавляють проектувальника необхідності вбудовувати в код класи, що залежать від програми. Код має справу лише з інтерфейсом класу, тому він може працювати з будь-якими визначеними користувачем класами з конкретною реалізацією.

Потенційний недолік фабричного методу полягає в тому, що клієнтам, можливо, доведеться створювати підклас класу Creator для створення лише одного об'єкта ConcreteProduct. Породження підкласів виправдане, якщо клієнту так чи інакше доводиться створювати підкласи Creator, інакше клієнту

доведеться мати справу з додатковим рівнем підкласів.

Реалізація. Однією з реалізацій даного патерну є **параметризований фабричний метод**. Цей варіант дозволяє створювати різні класи в залежності від параметру. Фабричному методу передається параметр, який ідентифікує об’єкт, який необхідно створити.

```
#include <iostream>

class DrawableObject {
public:
  virtual void draw() = 0;
};

class CircleObject : public DrawableObject {
public:
  void draw() {
	std::cout << "Draw Circle" << std::endl;
  }

};

class SquareObject : public DrawableObject {
public:
  void draw() {
	std::cout << "Draw Square" << std::endl;
  }

};

class DrawableFactory {

  public :
	enum Type{
		Circle = 0,
		Square = 1
	};

virtual DrawableObject * create(Type type) {
	if (type == Circle) return new CircleObject();
	if (type == Square)  return new SquareObject();
	//other drawable object
	return 0;
  }
};

int main()
```

```
{
  DrawableFactory * factory = new DrawableFactory();
  DrawableObject * f1 = factory->create(DrawableFactory::Circle);
  f1->draw();
  DrawableObject * f2 = factory->create(DrawableFactory::Square);
  f2->draw();

  delete f1;
  delete f2;
}
```

Фабричний метод — це просто звичайний виклик методу, який може повернути екземпляр класу. Але вони часто використовуються в поєднанні з успадкуванням, так що похідний клас перевизначає фабричний метод і повертає екземпляр похідного класу. Частіше ми реалізуємо фабрики за допомогою абстрактних базових класів.

У наступному прикладі ми хочемо відобразити 3D-сцену за допомогою OpenGL, DirectX тощо. Спочатку ми пишемо базовий клас, де похідні класи забезпечують реалізацію чистих віртуальних методів:

```
#include <iostream>
#include <string>

class Renderer
{
public:
  virtual ~Renderer() {};
  virtual void render() = 0;
};

class OpenGLRenderer : public Renderer
{
  void render() {
        std::cout << "OpenGL render \n";
  }
};

class DirectXRenderer : public Renderer
{
  void render() {
        std::cout << "DirectX render \n";
  }
};


class RendererFactory
{
```

```
public:
  Renderer *createRenderer(const std::string& type)
  {
  if(type == "opengl")
          return new OpenGLRenderer();
  else if(type == "directx")
          return new DirectXRenderer();
  return NULL;
  }
};


int main()
{
  RendererFactory *factory = new RendererFactory();
  factory->createRenderer("opengl")->render();
  return 0;
}
```

Друга можливість – це коли клас має реалізації фабричного методу.

Розглянемо приклад, коли реалізації класу декартових або полярних координат точки. В класичному вигляді:

```
class Point {
  double x, y;
public:
  enum PointType { cartesian, polar };
  Point(double a, double b, PointType type = cartesian) {
          if (type == cartesian) {
                  x = a; b = y;
          }
          else {
                  x = a * cos(b);
                  y = a * sin(b);
          }
  }
};
```

У парадигмі фабричного методу:

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>


class Point {
public:
```

```
  enum PointType { cartesian, polar };
private:
  double       m_x;
  double       m_y;
  PointType    m_type;

  // Private constructor, so that object can't be created directly
  Point(const double x, const double y, PointType t) : m_x{ x }, m_y{ y },
m_type{ t } {}

public:

  friend std::ostream &operator<<(std::ostream &os, const Point &obj) {
          return os << "x: " << obj.m_x << " y: " << obj.m_y;
  }
  static Point NewCartesian(float x, float y) {
          return { x, y, PointType::cartesian };
  }
  static Point NewPolar(float a, float b) {
          return { a * cos(b), a * sin(b), PointType::polar };
  }
};

int main()
{
  auto p = Point::NewPolar(5, M_PI_4);
  std::cout << p << std::endl;  // x: 3.53553 y: 3.53553
}
```

Як ви можете помітити з реалізації. Він фактично забороняє використання конструктора та змушує користувачів замість цього використовувати статичні методи. І це суть Фабричного методу, тобто приватного конструктора та статичного методу.

Якщо у вас є спеціальний код для побудови, тоді ми перемістимо його до спеціального класу. І просто відокремити проблеми, тобто принцип єдиної відповідальності від принципів проектування SOLID.

```
class Point {
   // ... as it is from above
   friend class PointFactory;
};

class PointFactory {
public:
   static Point NewCartesian(float x, float y) {
      return { x, y };
```

```
    }
    static Point NewPolar(float r, float theta) {
        return { r*cos(theta), r*sin(theta) };
    }
};
```

Майте на увазі, що це не абстрактна фабрика, це конкретна фабрика. Роблячи PointFactory дружнім класом Point, ми порушили принцип «відкрито-закрито» (OCP).

Існує критична річ, яку ми пропустили в нашій Фабриці: немає міцного зв’язку між PointFactory і Point, що збиває користувача з пантелику використовувати Point, просто бачачи, що все приватно.

Тож замість того, щоб проектувати фабрику за межами класу. Ми можемо просто помістити його в клас, який заохочує користувачів використовувати Factory.

Таким чином, ми також обслуговуємо другу проблему, яка полягає в порушенні принципу «відкрито-закрито». І це буде дещо більш інтуїтивно зрозумілим для користувача у використанні Factory.

```
class Point {
    float   m_x;
    float   m_y;

    Point(float x, float y) : m_x(x), m_y(y) {}
public:
    struct Factory {
        static Point NewCartesian(float x, float y) { return { x,y }; }
        static Point NewPolar(float r, float theta) { return{ r*cos(theta), r*sin(theta)
}; }
    };
};

int main() {
    auto p = Point::Factory::NewCartesian(2, 3);
    return 0;
}
```

2.6. Патерн Builder.

Призначення. Відокремлює конструювання складного об'єкта від його уявлення, так що в результаті того самого процесу конструювання можуть виходити різні уявлення. Будівельник — це патерн проектування, що дозволяє створювати складні об'єкти покроково. Будівельник дає можливість використовувати той самий код будівництва для отримання різних уявлень об'єктів.

Мотивація. Програма, в яку закладена можливість розпізнавання та читання документа у форматі RTF (Rich Text Format), повинна також «вміти»

перетворювати його на багато інших форматів, наприклад у простий ASCII-текст або у подання, яке можна відобразити у віджеті для введення тексту. Однак кількість ймовірних перетворень наперед невідома. Тому має бути забезпечена можливість легко додавати новий конвертор.

Таким чином, потрібно налаштувати клас RTFReader за допомогою об'єкта TextConverter, який міг би перетворювати RTF на інший текстовий формат. При розборі документа у форматі RTF клас RTFReader викликає TextConverter для виконання перетворення. Щоразу, як RTFReader розпізнає лексему RTF (простий текст чи керуюче слово), для її перетворення об'єкту TextConverter надсилається запит. Об'єкти TextConverter відповідають як за перетворення даних, так і за подання лексеми у конкретному форматі.

Підкласи TextConverter спеціалізуються на різних перетвореннях та форматах. Наприклад, ASCIIConverter ігнорує запити на перетворення будь-чого, крім простого тексту. З іншого боку, TeXConverter реалізовуватиме всі запити для отримання подання у форматі редактору TeX, збираючи необхідну інформацію про стилі. А TextWidgetConverter буде будувати складний об'єкт інтерфейсу користувача, який дозволить користувачеві переглядати і редагувати текст.

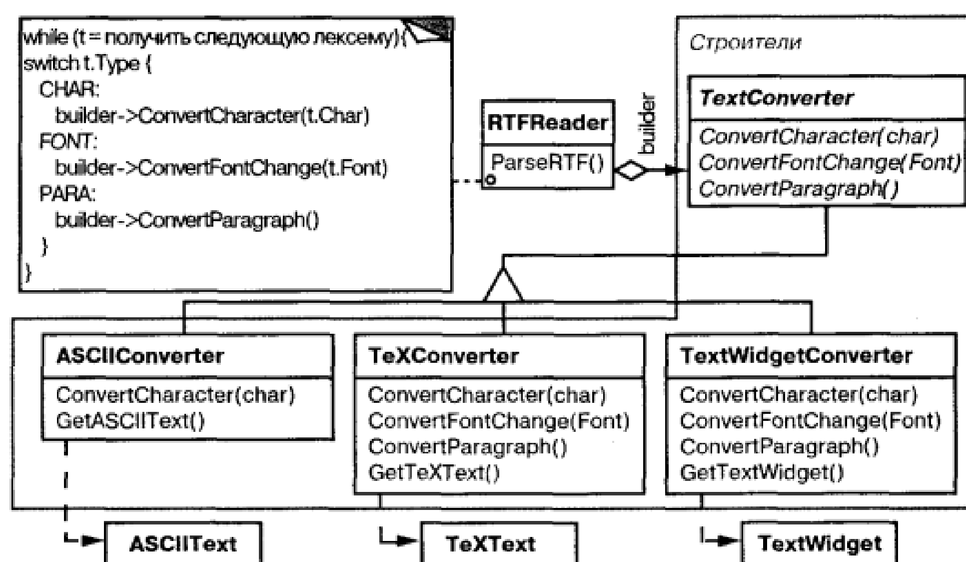

Рис. 2.6. Приклад взаємовідношень між класами в патерні Builder

Клас кожного конвертора приймає механізм створення та складання складного об'єкта та приховує його за абстрактним інтерфейсом. Конвертор відокремлений від завантажувача, який відповідає за синтаксичний аналіз RTF-документа.

У патерні будівельник абстраговані усі ці відносини. У ньому будь-який клас конвертора називається будівельником, а завантажувач – розпорядником. У застосуванні до розглянутого прикладу будівельник відокремлює алгоритм інтерпретації формату тексту (тобто аналізатор RTF-документів) від того, як створюється і представляється документ у перетвореному форматі. Це дозволяє

повторно використовувати алгоритм аналізу, реалізований в RTFReader, для створення різних текстових уявлень RTF-документів; достатньо передати до RTFReader різні підкласи класу TextConverter.

Застосування. Builder доцільно використовувати у випадках коли:

- алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт і як вони стикуються між собою;
- процес конструювання повинен забезпечувати різні уявлення об'єкта, що реконструюється.

Структура.

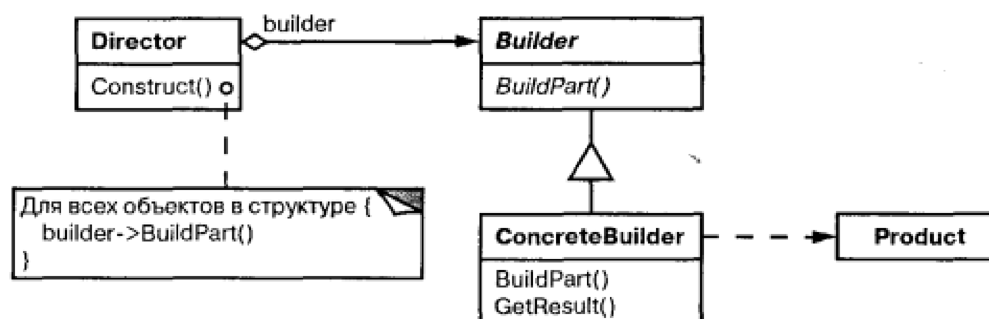

Рис. 2.7. Структура патерну Builder.

Учасники:

- **Builder** (TextConverter) – будівник: задає абстрактний інтерфейс для створення частин об’єкту Product;
- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter) – конкретний будівник: конструює та збирає разом частини продукту за допомогою реалізації інтерфейсу Builder, визначає створюване уявлення та стежить за ним, надає інтерфейс доступу до продукту (до прикладу, GetASCIIText, GetTextWidget);
- **Director** (RTFReader) – розпорядник: конструює об'єкт, користуючись інтерфейсом Builder;
- **Product** (ASCIIText, TeXText, TextWidget) – продукт: представляє складний об'єкт, що конструюється. ConcreteBuilder будує внутрішнє уявлення продукту і визначає процес його складання. Включає класи, які визначають складові, у тому числі інтерфейси для складання кінцевого результату з частин.

Відносини:

- **клієнт** створює об'єкт-розпорядник Director і конфігурує його потрібним об'єктом-будівельником Builder;
- **розпорядник** повідомляє **будівельника** про те, що потрібно збудувати чергову частину **продукту;**
- **будівельник** обробляє запити розпорядника та додає нові частини до **продукту;**
- **клієнт** забирає продукт у **будівельника.**

Результати. Плюси та мінуси будівельника та його застосування:

- дозволяє змінювати внутрішнє уявлення продукту. Об'єкт Builder надає

розпоряднику абстрактний інтерфейс для конструювання продукту, яким він може приховати уявлення і внутрішню структуру продукту, і навіть процес його складання. Оскільки продукт конструюється через абстрактний інтерфейс, то зміни внутрішнього уявлення досить лише визначити новий вид будівельника;

- ізолює код, що реалізує конструювання та подання. Патерн будівельник покращує модульність, інкапсулюючи спосіб конструювання та уявлення складного об'єкта. Клієнтам нічого не треба знати про класи, що визначають внутрішню структуру продукту, вони відсутні в інтерфейсі будівельника. Кожен конкретний будівельник ConcreteBuilder містить весь код, необхідний для створення і складання конкретного виду продукту. Код пишеться лише один раз, після чого різні розпорядники можуть використовувати його повторно для побудови варіантів продукту з тих самих частин. У прикладі з RTF-документом ми могли б визначити завантажувач для формату, відмінного від RTF, скажімо, SGMLReader, і скористатися тими самими класами TextConverters для генерування уявлень SGML-документів у вигляді ASCII-тексту, TeX-тексту або текстового віджету;
- дає більш тонкий контроль за процесом конструювання. На відміну від патернів, що породжують, які відразу конструюють весь об'єкт цілком, будівельник робить це крок за кроком під управління розпорядника. І лише коли продукт завершено, розпорядник забирає його у будівельника. Тому інтерфейс будівельника більшою мірою відображає процес конструювання продукту, ніж інші породжувальні патерни. Це дозволяє забезпечити більш тонкий контроль над процесом конструювання, а отже, і над внутрішньою структурою готового продукту.

Реалізація.

```
#include <iostream>
#include <list>
#include <string>
#include <conio.h>

class Product
{
 std::list<std::string> _parts;
 public:
	void Add(std::string part)
	{
		_parts.push_back(part);
	}

	void Show()
	{
		std::cout << std::endl << "Product Parts: " << std::endl;
		for (std::list<std::string>::iterator it = _parts.begin(); it != _parts.end(); ++it) {
			std::cout << it->c_str() << std::endl;
```

```
                }
        }
};

class Builder
{
public:
  virtual void BuildPartA() = 0;
  virtual void BuildPartB() = 0;
  virtual Product * GetResult() = 0;
};

class ConcreteBuilder1 : public Builder
{
        Product * _product;
  public:
        ConcreteBuilder1() {
                _product = new Product();
        }

        ~ConcreteBuilder1() {
                delete _product;
        }

        void BuildPartA()
        {
                _product->Add("PartA");
        }

        void BuildPartB()
        {
                _product->Add("PartB");
        }

        Product * GetResult()
        {
                return _product;
        }
};

class ConcreteBuilder2 : public Builder
{
  Product * _product;
public:
  ConcreteBuilder2() {
```

```
        _product = new Product();
  }

  ~ConcreteBuilder2() {
        delete _product;
  }

  void BuildPartA()
  {
        _product->Add("PartX");
  }

  void BuildPartB()
  {
        _product->Add("PartY");
  }

  Product * GetResult()
  {
        return _product;
  }
};

class Director {
  public:
  void Construct(Builder &builder)
  {
        builder.BuildPartA();
        builder.BuildPartB();
  }
};



int main()
{
  Director director;
  ConcreteBuilder1 b1;
  ConcreteBuilder2 b2;
  // Construct two products
  director.Construct(b1);
  Product * p1 = b1.GetResult();
  p1->Show();
  director.Construct(b2);
  Product * p2 = b2.GetResult();
```

```
 p2->Show();
 _getch();
}
```

Наступний приклад демонструє конструювання листа електронної пошти для відсилання. Клас Email описує всі необхідні параметри поштового листа. Класс EmailBuilder призначений для коструювання цього класу.

```
#pragma once
#include <iostream>
#include <string>

class Email
{
public:
 friend class EmailBodyBuilder;
 friend class EmailHeaderBuilder;
 friend class EmailBuilder;
 friend std::ostream &operator<<(std::ostream &os, const Email &obj);
 static EmailBuilder create();
private:
 Email() {}
 std::string m_from;
 std::string m_to;
 std::string m_subject;
 std::string m_body;
 std::string m_attachment;
};


class AbstractEmailBuilder
{
protected:
 Email &m_email;
 AbstractEmailBuilder(Email &email) : m_email(email) {}
public:
 EmailHeaderBuilder header() const;
 EmailBodyBuilder body() const;
 Email & GetResult() { return m_email; }
};


class EmailBodyBuilder : public AbstractEmailBuilder
{
public:
 EmailBodyBuilder(Email &email)
		: AbstractEmailBuilder(email)
```

```
  {
  }
  EmailBodyBuilder &body(const std::string &body)
  {
          m_email.m_body = body;
          return *this;
  }
  EmailBodyBuilder &attachment(const std::string &attachment)
  {
          m_email.m_attachment = attachment;
          return *this;
  }
};

class EmailHeaderBuilder : public AbstractEmailBuilder
{
public:
  EmailHeaderBuilder(Email &email)
          : AbstractEmailBuilder(email)
  {
  }
  EmailHeaderBuilder &from(const std::string &from)
  {
          m_email.m_from = from;
          return *this;
  }
  EmailHeaderBuilder &to(const std::string &to)
  {
          m_email.m_to = to;
          return *this;
  }
  EmailHeaderBuilder &subject(const std::string &subject)
  {
          m_email.m_subject = subject;
          return *this;
  }
};

EmailHeaderBuilder AbstractEmailBuilder::header() const
{
  return EmailHeaderBuilder( m_email );
};
EmailBodyBuilder AbstractEmailBuilder::body() const
{
  return EmailBodyBuilder( m_email );
```

```
};

class EmailBuilder : public AbstractEmailBuilder
{
  Email m_email;
public:
  EmailBuilder() : AbstractEmailBuilder( m_email )
  {
  }
};

EmailBuilder Email::create()
{
  return EmailBuilder();
}

std::ostream &operator<<(std::ostream &os, const Email &obj)
{
  return os
          << "from: " << obj.m_from << std::endl
          << "to: " << obj.m_to << std::endl
          << "subject: " << obj.m_subject << std::endl
          << "body: " << obj.m_body << std::endl
          << "attachment: " << obj.m_attachment << std::endl;
}

class ReportEmailDirector {
public:
  void Construct(EmailBuilder &builder)
  {
          builder.header()
                  .from("test1@example.com")
                  .to("test2@example.com")
                  .subject("This is a test mail")
                  .body()
                  .body("This is a test body")
                  .attachment("This is a test attachment");
  }
};

int main()
{

  EmailBuilder builder = Email::create();
  ReportEmailDirector director;
```

```
director.Construct(builder);

Email & email = builder.GetResult();
std::cout << email;

}
```

2.7. Патерн Prototype.

Призначення. Задає види об'єктів, що створюються за допомогою екземпляра-прототипу і створює нові об'єкти шляхом копіювання цього прототипу.

Мотивація. Побудувати музичний редактор вдалося шляхом адаптації загального каркасу графічних редакторів і додавання нових об'єктів, що представляють ноти, паузи і нотний стан. У каркасі редактора може бути панель інструментів для додавання в партитуру цих музичних об'єктів. Палітра також може містити інструменти для вибору, переміщення та інших маніпуляцій з об'єктами. Так, користувач, клацнувши, наприклад, по значку чверті, помістив би її тим самим у партитуру. Або, застосувавши інструмент переміщення, зрушував би ноту на таборі вгору чи вниз, щоб змінити її висоту.

Припустимо, що каркас надає абстрактний клас Graphic для графічних компонентів на зразок нот та нотних станів, а також абстрактний клас Tool для визначення інструментів на палітрі. Крім того, в каркасі є зумовлений підклас GraphicTool для інструментів, які створюють графічні об'єкти та додають їх у документ.

Проте клас GraphicTool створює певну проблему для проектувальника каркасу. Класи нот і нотних станів специфічні для нашого додатку, а клас GraphicTool належить каркасу. Цьому класу нічого невідомо про те, як створювати екземпляри наших музичних класів та додавати їх у партитуру. Можна було б породити від GraphicTool підкласи для кожного виду музичних об'єктів, але тоді виявилося б занадто багато класів, які відрізняються лише тим, який музичний об'єкт вони інстанціюють. Ми знаємо, що гнучкою альтернативою породженню підкласів є композиція. Питання в тому, як каркас міг би скористатися нею для параметризації екземплярів GraphicTool класом того об'єкта Graphic, який передбачається створити.

Рішення - змусити GraphicTool створювати новий графічний об'єкт, копіюючи або «клонуючи» екземпляр підкласу класу Graphic. Цей екземпляр ми називатимемо прототипом. GraphicTool параметризується прототипом, який він повинен клонувати та додати до документа. Якщо всі підкласи Graphic підтримують операцію C1one, то GraphicTool може клонувати будь-який вид графічних об'єктів.

Отже, в нашому музичному редакторі кожен інструмент для створення музичного об'єкта - це екземпляр класу GraphicTool, ініціалізований тим чи іншим прототипом. Будь-який екземпляр GraphicTool створюватиме музичний об'єкт, клонуючи його прототип і додаючи клон у партитуру.

Можна скористатися патерном прототип, щоб ще більше скоротити кількість класів. Для цілих і половинних нот ми маємо окремі класи, але, можливо, це зайве. Натомість вони могли б бути екземплярами одного і того ж класу, ініціалізованого різними растровими зображеннями та тривалістю звучання. Інструмент для створення цілих нот стає просто об'єктом класу GraphicTool, в якому прототип MusicalNote ініціалізований цілою нотою. Це може значно зменшити кількість класів у системі. Заодно спрощується додавання нового виду нот до музичного редактора.

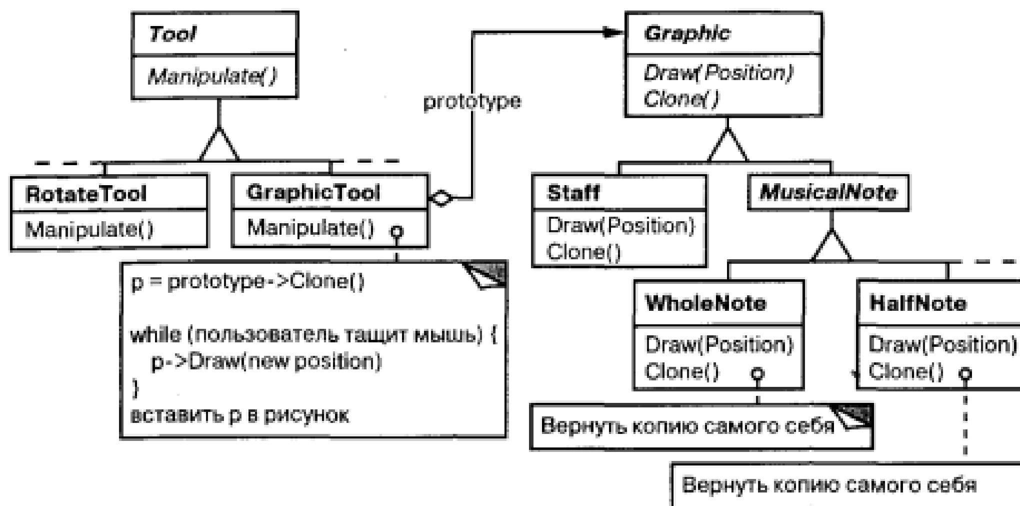

Рис. 2.8. Приклад взаємодії класів у патерні Prototype

Застосування.

Використовуйте патерн прототип, коли система не повинна залежати від того, як у ній створюються, компонуються та подаються продукти:

- класи, що інстанцуються, визначаються під час виконання, наприклад за допомогою динамічного завантаження;
- щоб уникнути побудови ієрархій класів або фабрик, паралельних ієрархії класів продуктів;
- екземпляри класу можуть перебувати в одному з невеликої кількості різних станів. Може виявитися зручнішим встановити відповідну кількість прототипів і клонувати їх, а не інстанціювати щоразу клас вручну у відповідному стані.

Учасники:

- **Prototype** (Graphic) – прототип, оголошує інтерфейс для клонування себе;
- **ConcretePrototype** (Staff – нотний стан, WholeNote – ціла нота,HalfNote – половинна нота) реалізує операцію клонування себе;
- **Client** (GraphicTool) – створює новий об'єкт, звертаючись до прототипу із запитом клонувати себе.

Структура.

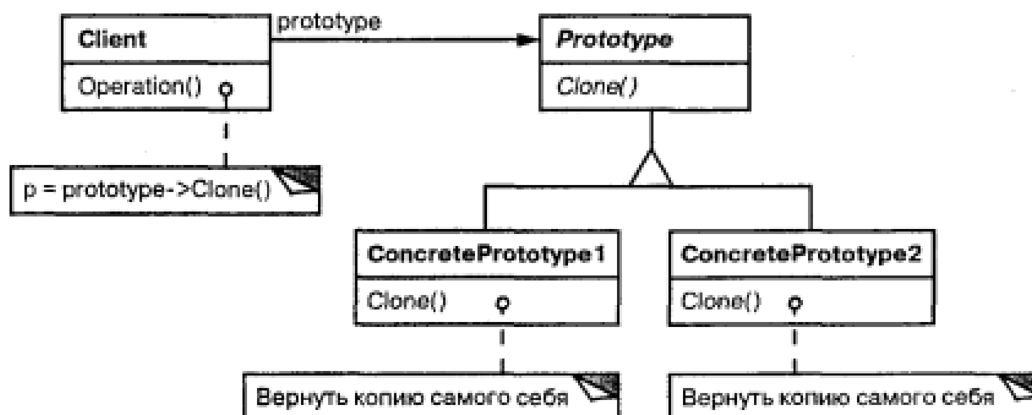

Рис. 2.9. Структура патерну прототип.

Відносини:

Клієнт звертається до **прототипу**, щоби той створив свою копію.

Результати.

У прототипу ті самі результати, що й у **абстрактної фабрики** і **будівельника**: він приховує від клієнта конкретні класи продуктів, зменшуючи цим число відомих клієнту імен. Крім того, всі ці патерни дозволяють клієнтам працювати зі специфічними для застосування класами без модифікацій.

Додаткові переваги прототипу такі:

- додавання та видалення продуктів під час виконання. Прототип дозволяє включати новий конкретний клас продуктів у систему, просто повідомивши клієнта про новий екземпляр-прототип. Це дещо гнучкіше рішення порівняно з тим, що вдасться зробити за допомогою інших патернів, що породжують, бо клієнт може встановлювати і видаляти прототипи під час виконання;
- специфікація нових об'єктів шляхом зміни значень. Динамічні системи дозволяють визначати поведінку з допомогою композиції об'єктів - наприклад, шляхом завдання значень змінних об'єкта, - а не з допомогою визначення нових класів. По суті, ви визначайте нові види об'єктів, інстанціюючи вже існуючі класи та реєструючи їх екземпляри як прототипи клієнтських об'єктів. Клієнт може змінити поведінку, делегуючи свої обов'язки прототипу. Такий дизайн дозволяє користувачам визначати нові класи без програмування. Фактично клонування об'єкта аналогічне створенню об’єкту класу. Патерн прототип може різко зменшити кількість необхідних класів системи. У нашому музичному редакторі за допомогою одного класу GraphivTool вдасться створити нескінченну різноманітність музичних об'єктів;
- специфікація нових об'єктів шляхом зміни структури. Багато додатків будують об'єкти з великих та дрібних складових. Наприклад, редактори для проектування плат створюють електричні схеми з підсхем. Такі програми часто дозволяють інстанціювати складні, визначені користувачем структури, скажімо, багаторазового використання деякої підсхеми. Патерн прототип підтримує таку можливість. Ми просто додаємо підсхему як прототип до палітри доступних

елементів схеми. За умови, що об'єкт, що представляє складову схему, реалізує операцію Clone як глибоке копіювання, схеми з різними структурами можуть виступати як прототипи;

- зменшення числа підкласів. Патерн фабричний метод часто породжує ієрархію класів Creator, паралельну до ієрархії класів продуктів. Прототип дозволяє клонувати прототип, а не вимагати фабричний метод створити новий об'єкт. Тому ієрархія класу Creator стає взагалі непотрібною;

- динамічне конфігурування програми класами. Деякі середовища дозволяють динамічно завантажувати класи в програму під час виконання. Патерн прототип – це ключ до застосування таких можливостей у мові типу C++. Додаток, який створює екземпляри класу, що динамічно завантажується, не може звертатися до його конструктора статично. Натомість середовище, що виконує, автоматично створює екземпляр кожного класу в момент його завантаження і реєструє екземпляр у диспетчері. Потім додаток може запросити у диспетчера прототипів екземпляри знову завантажених класів, які спочатку не були пов'язані з програмою.

Основний недолік патерну прототип полягає в тому, що кожен підклас класу Prototype повинен реалізовувати операцію Clone, а це далеко не завжди просто. Наприклад, складно додати операцію Clone, коли класи, що розглядаються, вже існують. Проблеми виникають і у випадку, якщо у внутрішньому поданні об'єкта є інші об'єкти або наявні кругові посилання.

Реалізація.

Прототип особливо корисний у статично типізованих мовах на зразок C++, де класи не є об'єктами, а під час виконання інформації про тип недостатньо чи ні. Основні питання, що виникають при реалізації прототипів:

- використання диспетчера прототипів. Якщо кількість прототипів у системі не фіксована (тобто вони можуть створюватися та знищуватись динамічно), ведіть реєстр доступних прототипів. Клієнти повинні не керувати прототипами самостійно, а зберігати та витягувати їх з реєстру. Клієнт запитує прототип із реєстру перед його клонуванням. Такий реєстр ми називатимемо диспетчером прототипів. Диспетчер прототипів – це асоціативне сховище, яке повертає прототип, що відповідає заданому ключу. У ньому є операції для реєстрації прототипу із зазначеним ключем та скасування реєстрації. Клієнти можуть змінювати і навіть переглядати реєстр під час виконання, а значить, розширювати систему і вести контроль над її станом без написання коду;

- реалізація операції Clone. Найважча частина патерну прототип - правильна реалізація операції Clone. Особливо складно це у разі, коли у структурі об'єкта є кругові посилання. Питання в наступному: чи повинні при клонуванні об'єкта клонуватися також його змінні екземпляра або клон просто розділяє з оригіналом ці змінні? Поверхневе копіювання просте, і часто його буває достатньо. У C++ копіювальний конструктор за замовчуванням виконує почленне копіювання, тобто вказівки розділяються копією та оригіналом. Але для клонування прототипів зі складною структурою зазвичай необхідно глибоке копіювання, оскільки клон має бути незалежним від оригіналу. Тому необхідно гарантувати, що компоненти клону є клонами компонентів прототипу. При

клонуванні вам доводиться вирішувати, що може розділятися і чи може взагалі.

- ініціалізація клонів. Хоча деяким клієнтам цілком достатньо клону, іншого потрібно ініціалізувати його внутрішній стан повністю або частково. Зазвичай передати початкові значення операції Clone неможливо, оскільки їхня кількість різна для різних класів прототипів. Для деяких прототипів потрібні багато параметрів ініціалізації, інші взагалі нічого не вимагають. Передача Clone параметрів заважає побудові одноманітного клонування інтерфейсу. Може виявитися, що у ваших класах прототипів вже визначаються операції встановлення та очищення деяких важливих елементів стану. Якщо так, то цими операціями можна скористатися одразу після клонування. В іншому випадку, можливо, знадобиться ввести операцію Initialize, яка приймає початкові значення як аргументи і відповідно встановлює внутрішній стан клону. Будьте обережними, якщо операція Clone реалізує глибоке копіювання: копії може знадобитися видаляти (явно або всередині Initialize) перед повторною ініціалізацією.

У загальному вигляді приклад прототипу виглядає так:

```
#include <iostream>
#include <string>
#include <unordered_map>
//
using std::string;

// Prototype Design Pattern
//
// Intent: Lets you copy existing objects without making your code dependent on
// their classes.

enum Type {
  PROTOTYPE_1 = 0,
  PROTOTYPE_2
};

/**
 * The example class that has cloning ability. We'll see how the values of field
 * with different types will be cloned.
 */

class Prototype {
protected:
  string prototype_name_;
  float prototype_field_;

public:
  Prototype() {}
  Prototype(string prototype_name)
```

```
        : prototype_name_(prototype_name) {
    }
    virtual ~Prototype() {}
    virtual Prototype *Clone() const = 0;
    virtual void Method(float prototype_field) {
        this->prototype_field_ = prototype_field;
        std::cout << "Call Method from " << prototype_name_ << " with field : "
<< prototype_field << std::endl;
    }
};

/**
 * ConcretePrototype1 is a Sub-Class of Prototype and implement the Clone
Method
 * In this example all data members of Prototype Class are in the Stack. If you
 * have pointers in your properties for ex: String* name_ ,you will need to
 * implement the Copy-Constructor to make sure you have a deep copy from the
 * clone method
 */

class ConcretePrototype1 : public Prototype {
private:
    float concrete_prototype_field1_;

public:
    ConcretePrototype1(string prototype_name, float concrete_prototype_field)
        :                    Prototype(prototype_name),
concrete_prototype_field1_(concrete_prototype_field) {
    }

    /**
     * Notice that Clone method return a Pointer to a new ConcretePrototype1
     * replica. so, the client (who call the clone method) has the responsability
     * to free that memory. I you have smart pointer knowledge you may prefer to
     * use unique_pointer here.
     */
    Prototype *Clone() const override {
        return new ConcretePrototype1(*this);
    }
};

class ConcretePrototype2 : public Prototype {
private:
    float concrete_prototype_field2_;
```

```
public:
  ConcretePrototype2(string prototype_name, float concrete_prototype_field)
      :                                         Prototype(prototype_name),
concrete_prototype_field2_(concrete_prototype_field) {
  }
  Prototype *Clone() const override {
      return new ConcretePrototype2(*this);
  }
};

/**
 * In PrototypeFactory you have two concrete prototypes, one for each concrete
 * prototype class, so each time you want to create a bullet , you can use the
 * existing ones and clone those.
 */

class PrototypeFactory {
private:
  std::unordered_map<Type, Prototype *, std::hash<int>> prototypes_;

public:
  PrototypeFactory() {
      prototypes_[Type::PROTOTYPE_1]            =            new
ConcretePrototype1("PROTOTYPE_1 ", 50.f);
      prototypes_[Type::PROTOTYPE_2]            =            new
ConcretePrototype2("PROTOTYPE_2 ", 60.f);
  }

  /**
   * Be carefull of free all memory allocated. Again, if you have smart pointers
   * knowelege will be better to use it here.
   */

  ~PrototypeFactory() {
      delete prototypes_[Type::PROTOTYPE_1];
      delete prototypes_[Type::PROTOTYPE_2];
  }

  /**
   * Notice here that you just need to specify the type of the prototype you
   * want and the method will create from the object with this type.
   */
  Prototype *CreatePrototype(Type type) {
      return prototypes_[type]->Clone();
  }
```

```
};

void Client(PrototypeFactory &prototype_factory) {
  std::cout << "Let's create a Prototype 1\n";

  Prototype *prototype =
prototype_factory.CreatePrototype(Type::PROTOTYPE_1);
  prototype->Method(90);
  delete prototype;

  std::cout << "\n";

  std::cout << "Let's create a Prototype 2 \n";

  prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_2);
  prototype->Method(10);

  delete prototype;
}

int main() {
  PrototypeFactory *prototype_factory = new PrototypeFactory();
  Client(*prototype_factory);
  delete prototype_factory;

  return 0;
}
```

Наступний приклад демонструє реалізацію клонування різних типів документів за допомогою диспетчера прототипів:

```
#include <iostream>
#include <vector>

using namespace std;
const int N = 3;

// Prototype
class Document
{
public:
  virtual Document* clone() const = 0;
  virtual void store() const = 0;
  virtual ~Document() {  }
};
```

```
// Concrete prototypes : xmlDoc, plainDoc, spreadsheetDoc

class xmlDoc : public Document
{
public:
  Document*   clone() const { return new xmlDoc; }
  void store() const { cout << "xmlDoc\n"; }
};

class plainDoc : public Document
{
public:
  Document* clone() const { return new plainDoc; }
  void store() const { cout << "plainDoc\n"; }
};

class spreadsheetDoc : public Document
{
public:
  Document* clone() const { return new spreadsheetDoc; }
  void store() const { cout << "spreadsheetDoc\n"; }
};

// create() calls Concrete Portotype's clone() method
// inherited from Prototype
class DocumentManager {
  Document* mDocTypes[N];
public:
  DocumentManager() {
          mDocTypes[0] = new xmlDoc;
          mDocTypes[1] = new plainDoc;
          mDocTypes[2] = new spreadsheetDoc;
  }
  ~DocumentManager() {
          for (int i = 1; i < size(); i++) {
                  delete mDocTypes[i];
          }
  }

  int size() { return N; }
  Document* create(int choice) { return mDocTypes[choice]->clone(); }


};
```

```
// Client
int main() {
  DocumentManager manager;
  vector<Document*> docs;
  int choice;
  cout << "quit(0), xml(1), plain(2), spreadsheet(3): \n";
  while (true) {
        cout << "Type in your choice (0-3)\n";
        cin >> choice;
        if (choice <= 0 || choice > manager.size())
              break;
        Document*newDoc = manager.create(choice - 1);
        docs.push_back(newDoc);
  }

  for (size_t i = 0; i < docs.size(); ++i)
        if (docs[i]) docs[i]->store();

  for (size_t i = 0; i < docs.size(); ++i) {
        if (docs[i]) delete docs[i];
  }

  return 0;
}
```

2.8. Патерн Object Pool.

Призначення. Пул об’єктів може значно підвищити продуктивність; він найбільш ефективний у ситуаціях, коли вартість ініціалізації екземпляра класу висока, швидкість створення екземплярів класу висока, а кількість екземплярів, що використовуються в будь-який момент часу, низька.

Мотивація. Пули об’єктів (інакше відомі як пули ресурсів) використовуються для керування кешуванням об’єктів. Клієнт, який має доступ до пулу об’єктів, може уникнути створення нових об’єктів, просто запросивши в пулу об’єкт, екземпляр якого вже створено. Зазвичай пул буде зростаючим, тобто сам пул створюватиме нові об’єкти, якщо пул порожній, або ми можемо мати пул, який обмежує кількість створюваних об’єктів.

Бажано зберігати всі повторно використовувані об’єкти, які зараз не використовуються, в одному пулі об’єктів, щоб ними можна було керувати за допомогою однієї узгодженої політики. Щоб досягти цього, клас Reusable Pool повинен бути розроблений як клас singleton.

Застосування.

Цей патерн застосовують, коли програмі потрібні об’єкти, створення яких є дорогим. Наприклад: потрібно відкрити забагато з’єднань для бази даних, тоді створення нового займе занадто багато часу, і сервер бази даних буде

перевантажений. Застосовується також, коли є кілька клієнтів, яким потрібен один і той же ресурс в різний час.

Пул об’єктів дозволяє іншим «вилучати» об’єкти з його пулу, коли ці об’єкти більше не потрібні їхнім процесам, вони повертаються до пулу для повторного використання.

Однак не потрібно, щоб процес чекав на звільнення певного об’єкта, тому пул об’єктів також створює екземпляри нових об’єктів, коли вони потрібні, але також має реалізовувати засіб для періодичного очищення невикористаних об’єктів.

Структура. Загальна ідея шаблону пулу підключень полягає в тому, що якщо екземпляри класу можна використовувати повторно, ви уникаєте створення екземплярів класу шляхом їх повторного використання.

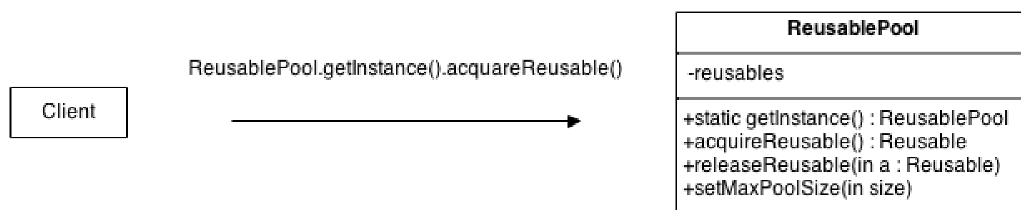

Рис. 2.10. Структура патерну «Пул об’єктів»

Учасники:

- **Reusable** – екземпляри класів у цій ролі співпрацюють з іншими об’єктами протягом обмеженого проміжку часу, після чого вони більше не потрібні для такої співпраці;
- **Client** – екземпляри класів у цій ролі використовують повторно використовувані об’єкти.
- **ReusablePool** – екземпляри класів у цій ролі керують повторно використовуваними об’єктами для використання клієнтськими об’єктами.

Результати.

Перевага шаблону проектування пулу об’єктів:

- значно підвищує продуктивність програми;
- найбільш ефективно в ситуації, коли швидкість ініціалізації екземпляра класу висока;
- керує підключеннями та надає спосіб повторного використання та спільного використання;
- може надати обмеження на максимальну кількість об’єктів, які можна створити.

Реалізація.

```
#include <iostream>
#include <list>
class Reusable
{
  int value;
```

```
public:
  Reusable()
  {
          value = 0;
  }
  void reset()
  {
          value = 0;
  }
  int getValue()
  {
          return value;
  }
  void setValue(int number)
  {
          value = number;
  }
};


template <class T>
class ReusablePool
{
  std::list<T*> _available;
  std::list<T*> _inUse;
  static ReusablePool<T>* instance;
  size_t maxPoolSize;
  ReusablePool() {
          maxPoolSize = 10;
  }
  ~ReusablePool() {

          for (auto it = _available.begin(); it != _available.end(); ++it) {

                  delete *it;
          }

          for (auto it = _inUse.begin(); it != _inUse.end(); ++it) {
                  delete *it;
          }

  }
public:
  static ReusablePool<T>* getInstance()
  {
```

```
        if (!ReusablePool<T>::instance)
        {
            ReusablePool<T>::instance = new ReusablePool<T>();
        }
        return ReusablePool<T>::instance;
    }

    static void free() {
        if (ReusablePool<T>::instance) {
            delete ReusablePool<T>::instance;
        }
    }

    void SetMaxPoolSize(size_t settingPoolSize)
    {
        maxPoolSize = settingPoolSize;
    }

    T *acquireReusable()
    {
        if (!_available.empty() && _available.size() < maxPoolSize)
        {
            T *item = _available.front();
            _inUse.push_back(item);
            _available.pop_front();
            return item;
        }
        T * obj = new T();
        _inUse.push_back(obj);
        return obj;
    }

    void releaseReusable(T *item)
    {
        if (_available.size() < maxPoolSize)
        {
            _available.push_back(item);
            _inUse.remove(item);
        }
        else
        {
            throw "To much object in pool!";
        }
    }
};
```

```
template <class T>
ReusablePool<T>* ReusablePool<T>::instance = 0;

int main()
{
  Reusable * r1 = ReusablePool<Reusable>::getInstance()->acquireReusable();
  Reusable * r2 = ReusablePool<Reusable>::getInstance()->acquireReusable();
  r1->setValue(14);
  r2->setValue(20);
  std::cout << r1->getValue() <<std::endl;
  std::cout << r2->getValue() << std::endl;
  ReusablePool<Reusable>::getInstance()->releaseReusable(r1);
  ReusablePool<Reusable>::getInstance()->releaseReusable(r2);
  Reusable * r3 = ReusablePool<Reusable>::getInstance()->acquireReusable();
  Reusable * r4 = ReusablePool<Reusable>::getInstance()->acquireReusable();
  std::cout << r3->getValue() << std::endl;
  std::cout << r4->getValue() << std::endl;

  ReusablePool<Reusable>::free();
}
```

2.9. Патерн Multiton.

Призначення. Multiton — це креативний шаблон дизайну. Це лише розширення шаблону Singleton.

Singleton дозволяє створити тільки один об'єкт класу. Конструктор робиться приватним, а об’єкти створюються лише за допомогою статичної функції, яка повертає екземпляр об’єкта. Коли ця функція викликається вперше, створюється новий об’єкт класу, а його посилання зберігається в класі. Подальший виклик функції повертає той самий об’єкт (посилання на яке зберігається).

Таким чином, створюється лише один об’єкт, тоді як multiton дозволяє зберігати кілька екземплярів (рівно n екземплярів, де n визначено), підтримуючи карту пов’язаних ключів і унікальних об’єктів.

Зауважте, що при реалізації шаблону на ключ може бути лише один екземпляр. Також зауважте, що ключ не обов’язково має бути рядковим значенням.

Реалізація.

```
#include <iostream>
#include <map>
#include <string>

template <typename T, typename Key = std::string>
```

```
class Multiton
{
protected:
  Multiton() {}
  ~Multiton() {}
private:
  Multiton(const Multiton&) {}
  Multiton& operator = (const Multiton&) { return *this; }
  static std::map<Key, T *> _instances;
public:
  static void destroy()
  {
          for (auto it = _instances.begin(); it != _instances.end(); ++it)
                  delete (*it    ).second;
          _instances.clear();
  }

  static void destroy(const Key &key)
  {
          auto it = _instances.find(key);

          if (it != _instances.end()) {
                  delete (*it).second;
                  _instances.erase(it);
          }
  }

  static T* getInstance(const Key &key)
  {
          const auto it = _instances.find(key);

          if (it != _instances.end())
                  return (T*)(it->second);

          T* instance = new T;
          _instances[key] = instance;
          return instance;
  }

  static T& getRef(const Key &key)
  {
          return *getPtr(key);
  }
};
template <typename T, typename Key>
```

```
std::map<Key, T*> Multiton<T, Key>::_instances;

class Resource: public Multiton<Resource>
{
  int value;
public:
  Resource()
  {
          value = 0;
  }
  void reset()
  {
          value = 0;
  }
  int getValue()
  {
          return value;
  }
  void setValue(int number)
  {
          value = number;
  }
};

int main()
{
  Resource * f1 = Resource::getInstance("first");
  f1->setValue(14);
  std::cout << f1->getValue() << std::endl;
  Resource * f2 = Resource::getInstance("second");
  f2->setValue(24);
  std::cout << f2->getValue() << std::endl;

  std::cout << Resource::getInstance("first")->getValue() << std::endl;
  std::cout << Resource::getInstance("second")->getValue() << std::endl;
  Resource::destroy();
}
```

Контрольні питання

1. Які переваги та недоліки патерну Sinleton?
2. Які переваги та недоліки патерну Abstract Factory?
3. Які переваги та недоліки патерну Factory Method?
4. Які переваги та недоліки патерну Builder?
5. Які переваги та недоліки патерну Prototype?

РОЗДІЛ 3
СТРУКТУРНІ ПАТЕРНИ

3.1. Характеристика структурних патернів.

У структурних патернах розглядається питання про те, як із класів та об'єктів утворюються більші структури. Структурні патерни рівня класу використовують успадкування для складання композицій з інтерфейсів та реалізацій. Простий приклад - використання множинного успадкування для об'єднання кількох класів на один. У результаті виходить клас, що має властивості всіх своїх батьків. Особливо корисний цей патерн, коли потрібно організувати спільну роботу кількох незалежно розроблених бібліотек. Інший приклад патерну рівня класу – адаптер. У випадку адаптер робить інтерфейс одного класу (адаптируемого) сумісним з інтерфейсом іншого, забезпечуючи цим уніфіковану абстракцію різнорідних інтерфейсів. Це досягається за рахунок закритого успадкування класу, що адаптується. Після цього адаптер висловлює свій інтерфейс у термінах операцій класу, що адаптується.

Замість композиції інтерфейсів чи реалізацій структурні патерни рівня об'єкта компонують об'єкти отримання нової функціональності. Додаткова гнучкість у разі пов'язані з можливістю змінити композицію об'єктів під час виконання, що неприпустимо для статичної композиції класів.

3.2. Патерн Adapter.

Призначення. Перетворює інтерфейс одного класу на інтерфейс іншого, на який очікують клієнти. Адаптер забезпечує спільну роботу класів із несумісними інтерфейсами, яка без нього була б неможлива.

Мотивація. Іноді клас із інструментальної бібліотеки, спроектований для повторного використання, не вдається використовувати лише тому, що його інтерфейс не відповідає тому, що потрібно конкретному додатку.

Розглянемо, наприклад, графічний редактор, завдяки якому користувачі можуть малювати на екрані графічні елементи (лінії, багатокутники, текст тощо) та організовувати їх у вигляді картинок та діаграм. Основною абстракцією графічного редактора є графічний об'єкт, який має форму, що змінюється і зображує сам себе. Інтерфейс графічних об'єктів визначений абстрактним класом Shape. Редактор визначає підклас класу Shape для кожного виду графічних об'єктів: LineShape для прямих, PolygonShape для багатокутників та т.д.

Класи для елементарних геометричних фігур, наприклад LineShape і PolygonShape, реалізувати порівняно не складно, оскільки закладені у яких можливості малювання і редагування вкрай обмежені. Але підклас TextShape, що вміє відображати та редагувати текст, вже значно складніший, оскільки навіть для найпростіших операцій редагування тексту потрібно нетривіальним чином оновлювати екран та керувати буферами. У той же час, можливо, існує вже готова бібліотека для розробки інтерфейсів, яка надає розвинений клас TextView, що дозволяє відображати і редагувати текст. В ідеалі ми хотіли б повторно

використовувати TextView для реалізації TextShape, але бібліотека розроблялася без урахування класів Shape, тому змусити об'єкти TextView та Shape працювати спільно не вдається.

То яким чином існуючі та незалежно розроблені класи на кшталт TextView можуть працювати у додатку, який спроектовано під інший, несумісний інтерфейс? Можна було б змінити інтерфейс класу TextView, щоб він відповідав інтерфейсу Shape, лише цього потрібен вихідний код. Але навіть якщо він доступний, то навряд чи розумно змінювати TextView; бібліотека не повинна пристосовуватись до інтерфейсів кожної конкретної програми.

Натомість ми могли б визначити клас TextShape так, що він буде адаптувати інтерфейс TextView до інтерфейсу Shape. Це допустимо зробити двома способами: успадковуючи інтерфейс від Shape, а реалізацію від TextView; включивши екземпляр TextView в TextShape та реалізувавши TextShape термінах інтерфейсу TextView. Два даних підходи відповідають варіантам патерна адаптер у його класовій та об'єктній іпостасях. Клас TextShape ми називатимемо адаптером.

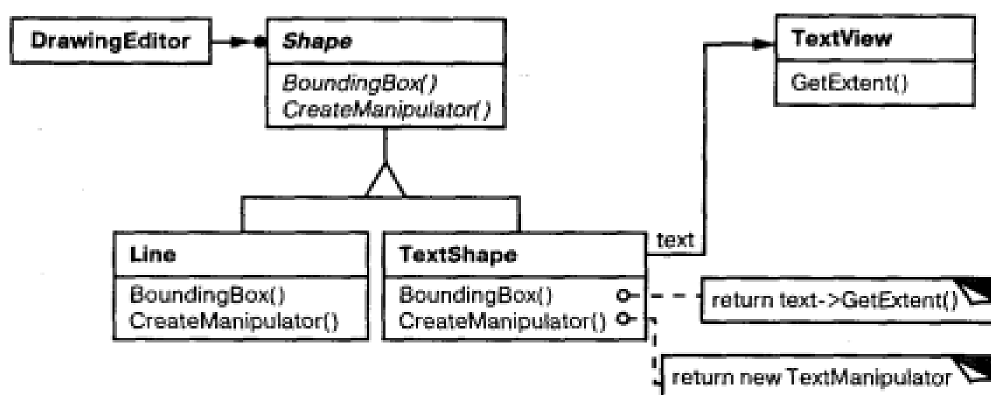

Рис. 3.1. Приклад взаємодії класів в патерні Adapter.

На рис. показано адаптер об'єкта. Видно, як запит BoundingBox, оголошений у класі Shape, перетворюється на запит GetExtent, визначений у класі TextView. Оскільки клас TextShape адаптує TextView до інтерфейсу Shape, графічний редактор може скористатися класом TextView, хоча той і має несумісний інтерфейс.

Часто адаптер відповідає за функціональність, яку не може надати клас, що адаптується. На рис. показано, як адаптер виконує такі функції. Користувач повинен мати можливість переміщати будь-який об'єкт класу Shape в інше місце, але в класі TextView така операція не передбачена. TextShape може додати функціональність, що бракує, самостійно реалізувавши операцію CreateManipulator класу Shape, яка повертає екземпляр відповідного підкласу Manipulator.

Manipulator - це абстрактний клас об'єктів, яким відомо, як анімувати Shape у відповідь на такі дії користувача, як перетягування фігури в інше місце. У класу Manipulator є підкласи для різних фігур. Наприклад, TextManipulator - підклас для TextShape. Повертаючи екземпляр TextManipulator, об'єкт класу TextShape

додає нову функціональність, якої в класі TextView немає, а класу Shape потрібно.

Застосування. Потрібно застосовувати Adapter коли:

- хочете використовувати існуючий клас, але його інтерфейс не відповідає вашим потребам;
- збираєтеся створити повторно використовуваний клас, який повинен взаємодіяти із заздалегідь невідомими або не пов'язаними з ним класами, що мають несумісні інтерфейси;
- (лише для адаптера об'єктів!) потрібно використовувати кілька існуючих підкласів, але непрактично адаптувати їх інтерфейси шляхом створення нових підкласів від кожного. У цьому випадку адаптер об'єктів може пристосовувати інтерфейс загального батьківського класу.

Структура. Адаптер класу використовує множинне наслідування для адаптації одного інтерфейсу до іншого. Адаптер об’єкту застосовує композицію об’єктів.

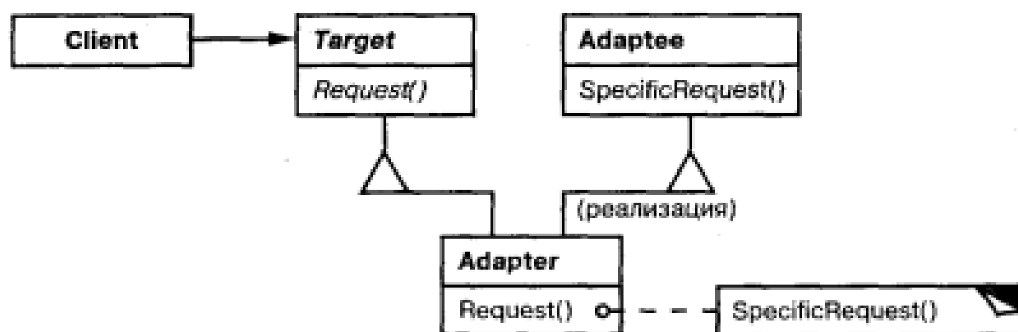

Рис. 3.2. Структура патерну Adapter.

Учасники:

- **Target** (Shape) - цільовий: визначає залежний від предметної області інтерфейс, яким користується Client;
- **Client** (DrawingEditor) - клієнт: вступає у взаємовідносини з об'єктами, що задовольняють інтерфейс Target;
- **Adaptee** (TextView) – адаптуємий: визначає існуючий інтерфейс, який потребує адаптації;
- **Adapter** (TextShape) - адаптер: адаптує інтерфейс Adaptee до інтерфейсу Target.

Відносини. Клієнти викликають операції екземпляра адаптера Adapter. У свою чергу адаптер викликає операції об'єкта, що адаптується, або класу Adaptee, який і виконує запит.

Результати.

Результати застосування адаптерів об'єктів та класів різні. Адаптер класу:

- адаптує Adaptee до Target, доручаючи дії конкретному класу Adaptee. Тому цей патерн не працюватиме, якщо ми захочемо одночасно адаптувати клас та його підкласи;

- дозволяє адаптеру Adapter заміститити деякі операції адаптованого класу Adaptee, оскільки Adapter є не що інше, як підклас Adaptee;
- вводить лише один новий об'єкт. Щоб дістатися до класу, що адаптується, не потрібно ніякого додаткового звернення за вказівкою.

Адаптер об'єктів:

- дозволяє одному адаптеру Adapter працювати з багатьма об'єктами Adaptee, що адаптуються, тобто з самим Adaptee та його підкласами (якщо такі є). Адаптер може додати нову функціональність відразу всім об'єктам, що адаптуються;
- ускладнює заміщення операцій класу Adaptee. Для цього потрібно породити від Adaptee підклас і змусити Adapter посилатися на цей підклас, а не на сам Adaptee.

Нижче наведено питання, які слід розглянути, коли ви вирішуєте застосувати адаптер:

- обсяг роботи з адаптації. Адаптери сильно відрізняються за тим обсягом роботи, який необхідний для адаптації інтерфейсу Adapter до інтерфейсу Target. Це може бути як найпростіше перетворення, наприклад, зміна імен операцій, так і підтримка зовсім іншого набору операцій. Обсяг роботи залежить від того, наскільки сильно відрізняються один від одного інтерфейси цільового та адаптованого класів;
- змінні адаптери. Ступінь повторної використовуваності класу тим вищий, чим менше припущень робиться про ті класи, які його застосовуватимуть. Вбудовуючи адаптацію інтерфейсу в клас, ви відмовляєтеся від припущення, що іншим класам стане доступний той самий інтерфейс. Іншими словами, адаптація інтерфейсу дозволяє включити ваш клас до існуючих систем, які спроектовані для класу з іншим інтерфейсом
- використання двосторонніх адаптерів для забезпечення прозорості. Адаптери непрозорі всіх клієнтів. Адаптований об'єкт вже не має інтерфейсу Adaptee, так що його не можна використовувати там, де Adaptee був застосований. Двосторонні адаптери здатні забезпечити таку прозорість. Точніше вони корисні в тих випадках, коли клієнт повинен бачити об'єкт по-різному.

Реалізація.

```
#include <iostream>
typedef int Coordinate;
typedef int Dimension;
// Desired interface
class Rectangle
{
public:
  virtual void draw() = 0;
};
// Legacy component
class LegacyRectangle
{
```

```
public:
  LegacyRectangle(Coordinate x1, Coordinate y1, Coordinate x2, Coordinate y2)
  {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle:  create.  (" << x1_ << "," << y1_ << ")
=> ("
              << x2_ << "," << y2_ << ")" << std::endl;
  }
  void oldDraw()
  {
        std::cout << "LegacyRectangle:  oldDraw.  (" << x1_ << "," << y1_ <<
              ") => (" << x2_ << "," << y2_ << ")" << std::endl;
  }
private:
  Coordinate x1_;
  Coordinate y1_;
  Coordinate x2_;
  Coordinate y2_;
};

// Adapter wrapper
class RectangleAdapter : public Rectangle, private LegacyRectangle
{
public:
  RectangleAdapter(Coordinate x, Coordinate y, Dimension w, Dimension h) :
        LegacyRectangle(x, y, x + w, y + h)
  {
        std::cout << "RectangleAdapter: create.  (" << x << "," << y <<
              "), width = " << w << ", height = " << h << std::endl;
  }
  virtual void draw(){
        std::cout << "RectangleAdapter: draw." << std::endl;
        oldDraw();
  }
};

int main(){
  Rectangle *r = new RectangleAdapter(120, 200, 60, 40);
  r->draw();
  delete r;
  return 0;
}
```

3.3. Патерн Bridge.

Призначення. Відокремити абстракцію від її реалізації так, щоб те й інше можна було змінювати незалежно.

Мотивація. Якщо для деякої абстракції можливе кілька реалізацій, то зазвичай застосовують успадкування. Абстрактний клас визначає інтерфейс абстракції, а його конкретні підкласи по-різному реалізують його. Але такий підхід не завжди має достатню гнучкість. Спадкування жорстко прив'язує реалізацію до абстракції, що ускладнює незалежну модифікацію, розширення та повторне використання абстракції та її реалізації.

Розглянемо реалізацію переносної абстракції вікна в бібліотеці для розробки інтерфейсів користувача. Написані з її допомогою програми повинні працювати в різних середовищах, наприклад під XW indows System і Presentation Managr (PM) від компанії 1BM. За допомогою успадкування ми могли б визначити абстрактний клас Window та його підкласи XWindow та PMWindow, що реалізують інтерфейс вікна для різних платформ. Але таке рішення має два недоліки:

1. Незручно розповсюджувати абстракцію Window на інші види вікон або нові платформи. Уявіть собі підклас IconWindow, який спеціалізує абстракцію вікна для піктограм. Щоб підтримати піктограми на обох платформах, нам доведеться реалізувати два нових підкласи XIconWindow і PMIconWindow. Більше того, по два підкласи потрібно визначати для кожного виду вікон. А для підтримки третьої платформи доведеться визначати для всіх видів вікон новий підклас Window.
2. Клієнтський код стає платформно-залежним. Під час створення вікна клієнт інстанціює конкретний клас, який має цілком певну реалізацію. Наприклад, створюючи об'єкт XWindow, ми прив'язуємо абстракцію вікна до її реалізації для системи X Window, отже, робимо код клієнта орієнтованим саме на цю віконну систему. Таким чином, ускладнюється перенесення клієнта на інші платформи.

Клієнти повинні мати можливість створювати вікно, не прив'язуючись до конкретної реалізації. Тільки сама реалізація вікна має залежати від платформи, де працює додаток. Тому в клієнтському коді не може бути жодних згадок про платформи.

За допомогою патерну міст ці проблеми вирішуються. Абстракція вікна та її реалізація поміщаються у окремі ієрархії класів. Таким чином, існує одна ієрархія для інтерфейсів вікон (Window, IconWindow, TransientWindow) та інша (з коренем WindowImp) - для платформно-залежних реалізацій. Так, підклас XWIndowImp надає реалізацію в системі X Window System.

Всі операції підкласів Window реалізовані в термінах абстрактних операцій з інтерфейсу WindowImp. Це відокремлює абстракцію вікна від різних її платформно-залежних реалізацій. Відношення між класами Window та WindowI,p ми називатимемо мостом, оскільки між абстракцією та реалізацією будується міст, і вони можуть змінюватися незалежно.

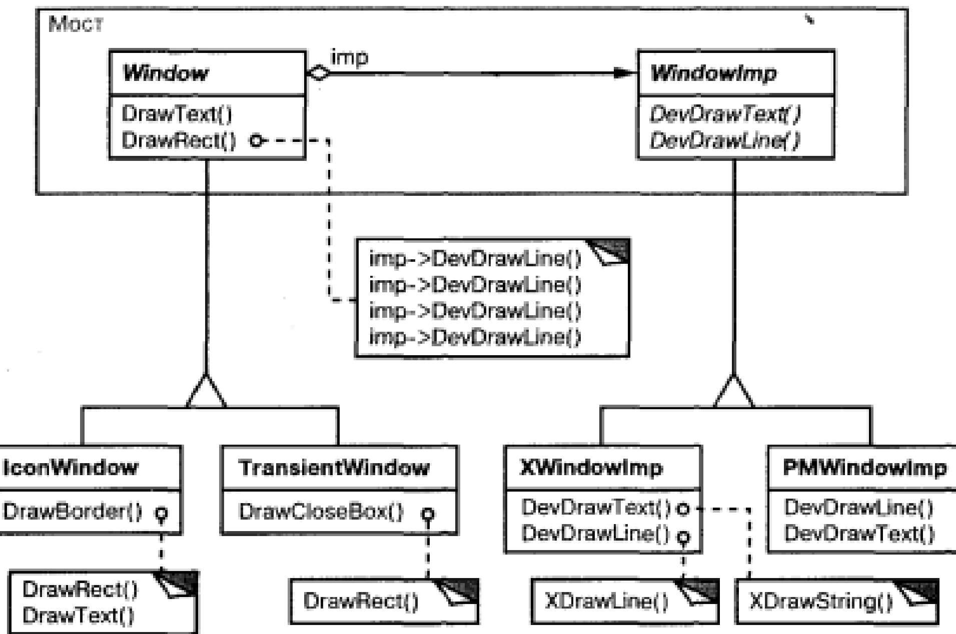

Рис. 3.3. Приклад реалізації патерну Bridge.

Структура.

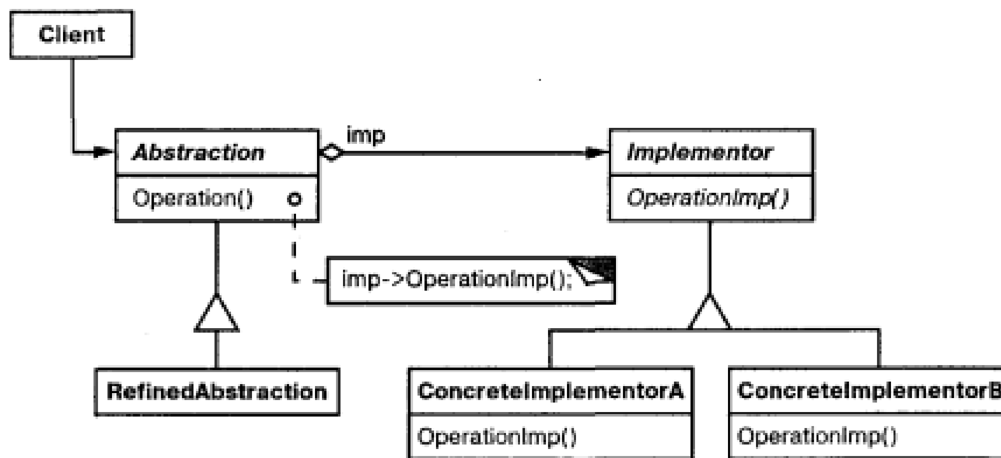

Рис. 3.4. Структура патерну Bridge.

Застосування. Патерн мост доцільно використовувати, коли:

- хочете уникнути постійної прив'язки абстракції до реалізації. Так, наприклад, буває, коли реалізацію необхідно обирати під час виконання програми;
- і абстракції, і реалізації мають розширюватись новими підкласами. У такому разі патерн міст дозволяє комбінувати різні абстракції та реалізації та змінювати їх незалежно;
- зміни в реалізації абстракції не повинні впливати на клієнтів, тобто клієнтський код не повинен перекомпілюватися;
- Ви хочете повністю приховати від клієнтів реалізацію абстракції;
- кількість класів починає швидко зростати;
- Ви хочете розділити одну реалізацію між кількома об'єктами (може

бути, застосовуючи підрахунок посилань), і цей факт необхідно приховати від клієнта.

Учасники:

- **Abstraction** (Window) - абстракція: визначає інтерфейс абстракції; зберігає посилання на об'єкт типу Implementor;
- **RefinedAbstraction** (IconWindow) - уточнена абстракція: розширює інтерфейс, визначений абстракцією Abstraction;
- **Implementor** (WindowImp) - реалізатор: визначає інтерфейс для класів реалізації. Він повинен точно відповідати інтерфейсу класу Abstraction. Насправді обидва інтерфейси можуть бути різними. Зазвичай інтерфейс класу Implementor надає лише примітивні операції, а клас Abstraction визначає операції вищого рівня, що базуються на цих примітивах;
- **ConcreteImplementor** (XWindowImp, PMWindowImp) - конкретний реалізатор: містить конкретну реалізацію інтерфейсу класу Implementor.

Відносини. Об'єкт Abstraction перенаправляє своєму об'єкту Implementor запити клієнта.

Результати:

- відокремлення реалізації від інтерфейсу. Реалізація більше немає постійної прив'язки до інтерфейсу. Реалізацію абстракції можна налаштувати під час виконання. Об'єкт може динамічно змінювати свою реалізацію. Розподіл класів Abstraction і Implementor усуває також залежності від реалізації, що встановлюються на етапі компіляції. Щоб змінити клас реалізації, зовсім не обов'язково перекомпілювати клас Abstraction та його клієнтів. Крім того, такий поділ полегшує розбиття системи на шари і, таким чином, дозволяє поліпшити її структуру. Високорівневі частини системи повинні знати тільки про класи Abstraction і Implementor;
- підвищення ступеня розширюваності. Можна розширювати незалежно ієрархії класів Abstraction і Implementor;
- приховування деталей реалізації від клієнтів. Клієнтів можна ізолювати від таких деталей реалізації, як поділ об'єктів класу Implementor та супутнього механізму підрахунку посилань.

Реалізація.

```
#include <iostream>
#include <iomanip>
#include <string>
class TimeImp {
public:
  TimeImp(int hr, int min) {
        hr_ = hr;
        min_ = min;
  }
  virtual void tell() {
        std::cout << "time is " << std::setw(2) << std::setfill('0') << hr_ << min_
<< std::endl;
  }
```

```
protected:
  int hr_, min_;
};

class CivilianTimeImp : public TimeImp {
public:
  CivilianTimeImp(int hr, int min, int pm) : TimeImp(hr, min)
{
        if (pm)
                strcpy_s(whichM_, 4, " PM");
        else
                strcpy_s(whichM_, 4, " AM");
  }

  /* virtual */
  void tell()
{
        std::cout << "time is " << hr_ << ":" << min_ << whichM_ << std::endl;
  }
protected:
  char whichM_[4];
};

class ZuluTimeImp : public TimeImp {
public:
  ZuluTimeImp(int hr, int min, int zone) : TimeImp(hr, min) {
        if (zone == 5)
                strcpy_s(zone_, 30, " Eastern Standard Time");
        else if (zone == 6)
                strcpy_s(zone_, 30, " Central Standard Time");
  }

  void tell() {
        std::cout << "time is " << std::setw(2) << std::setfill('0') << hr_ << min_
<< zone_ << std::endl;
  }
protected:
  char zone_[30];
};

class Time {
public:
  Time() { imp_ = 0; }
  Time(int hr, int min) {
        imp_ = new TimeImp(hr, min);
```

```
  }
  ~Time() {
          delete imp_;
  }
  virtual void tell() {
          imp_->tell();
  }
protected:
  TimeImp *imp_;
};

class CivilianTime : public Time {
public:
  CivilianTime(int hr, int min, int pm) {
          imp_ = new CivilianTimeImp(hr, min, pm);
  }
};

class ZuluTime : public Time {
public:
  ZuluTime(int hr, int min, int zone) {
          imp_ = new ZuluTimeImp(hr, min, zone);
  }
};

int main() {
  Time *times[3];
  times[0] = new Time(14, 30);
  times[1] = new CivilianTime(2, 30, 1);
  times[2] = new ZuluTime(14, 30, 6);
  for (int i = 0; i < 3; i++) {
          times[i]->tell();
  }
  for (int i = 0; i < 3; i++) {
          delete times[i];
  }
  return 0;
}
```

Паттерн абстрактна фабрика може створити та налаштувати міст.

Для забезпечення спільної роботи не пов'язаних між собою класів насамперед призначений патерн адаптер. Зазвичай він застосовується у вже готових системах. Міст же бере участь у проекті від початку і покликаний підтримати можливість незалежної зміни абстракцій та їх реалізацій.

3.4. Патерн Composite.

Призначення. Компонує об'єкти в деревоподібні структури для представлення ієрархій частина-ціле. Дозволяє клієнтам однаково трактувати індивідуальні та складові об'єкти.

Мотивація.

Такі програми, як графічні редактори та редактори електричних схем, дозволяють користувачам будувати складні діаграми з більш простих компонентів. Проектувальник може згрупувати дрібні компоненти для формування більших, які, у свою чергу, можуть стати основою для створення ще більших. У простій реалізації припустимо було б визначити класи графічних примітивів, наприклад тексту та ліній, а також класи, які виступають у ролі контейнерів для цих примітивів.

Але таке рішення має істотний недолік. Програма, в якій ці класи використовуються, повинна по-різному поводитися з примітивами та контейнерами, хоча користувач найчастіше працює з ними однаково. Необхідність розрізняти ці об'єкти ускладнює програму. Паттерн компонувальник описує, як можна застосувати рекурсивну композицію таким чином, що клієнту не доведеться проводити різницю між простими та складовими об'єктами.

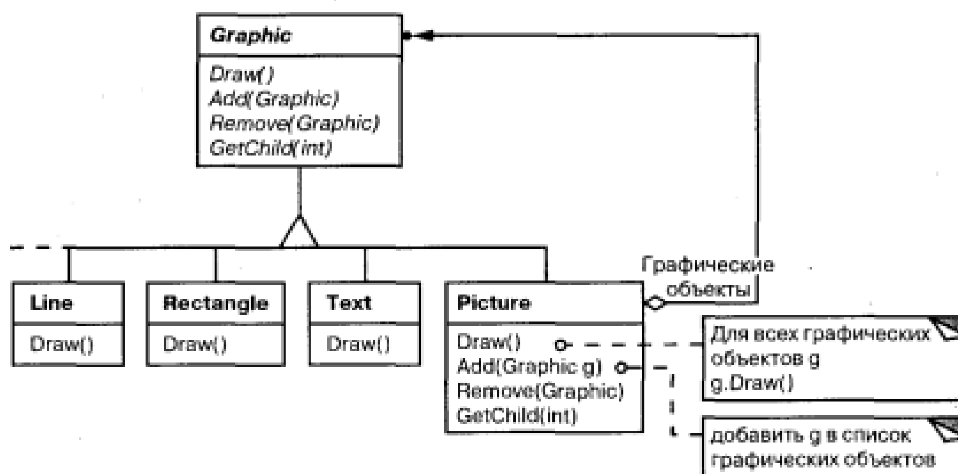

Рис. 3.5. Приклад взаємодії класів у патерні Composite

Ключем до патерну компонувальник є абстрактний клас, який представляє одночасно і примітиви, і контейнери. У графічній системі цей клас може називатися Graphic. У ньому оголошено операції, специфічні для кожного виду графічного об'єкта (такі як Draw) і загальні для всіх складних об'єктів, наприклад операції для доступу і управління нащадками.

Підкласи Line, Rectangle і Text визначають примітивні графічні об'єкти. В них операція Draw реалізована відповідно до малювання прямих, прямокутників і тексту. Оскільки примітивні об'єкти не мають нащадків, то жоден з цих підкласів не реалізує операції, що стосуються управління нащадками.

Клас Picture визначає агрегат, що складається з об'єктів Graphic. Реалізована

в ньому операція Draw викликає однойменну функцію для кожного нащадка, а операції для роботи з нащадками вже не порожні. Оскільки інтерфейс класу Picture відповідає інтерфейсу Graphic, то до складу об'єкта Picture можуть входити інші такі ж об'єкти.

Застосування. Використовуйте патерн компонувальник, коли:

- потрібно уявити ієрархію об'єктів виду частина-ціле;
- хочете, щоб клієнти однаково трактували складові та індивідуальні об'єкти.

Структура.

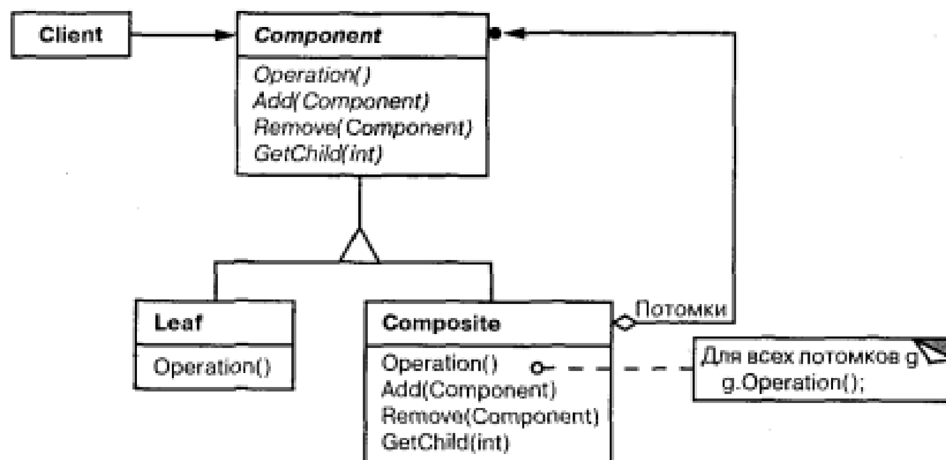

Рис. 3.6. Структура патерну Composite

Учасники:

- **Component** (Graphic) - компонент: оголошує інтерфейс для об'єктів, що компонуються; надає відповідну реалізацію операцій за умовчанням, що є загальною для всіх класів; оголошує інтерфейс для доступу до нащадків та управління ними; визначає інтерфейс для доступу до батька компонента в рекурсивній структурі і при необхідності реалізує його. Описана можливість необов'язкова;
- **Leaf** (Rectangle, Line, Text і т.п.) - лист: представляє листові вузли композиції та не має нащадків; визначає поведінку примітивних об'єктів у композиції;
- **Composite** (Picture) - складовий об'єкт: визначає поведінку компонентів, які мають нащадки; - зберігає компоненти-нащадки; реалізує операції, що належать до управління нащадками, в інтерфейсі класу Component;
- **Client** - клієнт: маніпулює об'єктами композиції через інтерфейс Component.

Відносини.

Клієнти використовують інтерфейс класу Component для взаємодії з об'єктами у складовій структурі. Якщо одержувачем запиту є листовий об'єкт Leaf, він і обробляє запит. Коли ж одержувачем є складовий об'єкт Composite, то він перенаправляє запит своїм нащадкам, можливо, виконуючи деякі додаткові операції до або після перенаправлення.

Результати. Патерн компонувальник:

- визначає ієрархії класів, що складаються з примітивних та складових об'єктів. З примітивних об'єктів можна складати складніші, які, своєю чергою, беруть участь у складніших композиціях тощо. Будь-який клієнт, який чекає на примітивний об'єкт, може працювати і зі складовим;
- спрощує архітектуру клієнта. Клієнти можуть однаково працювати з індивідуальними та об'єктами та зі складовими структурами. Зазвичай клієнту невідомо, чи взаємодіє він із листовим чи складовим об'єктом. Це спрощує код клієнта, оскільки немає необхідності писати функції, що розгалужуються залежно від того, з об'єктом якого класу вони працюють;
- полегшує додавання нових видів компонентів. Нові підкласи класів Composite або Leaf автоматично працюватимуть із вже існуючими структурами та клієнтським кодом. Змінювати клієнта при додаванні нових компонентів не потрібно;
- сприяє створенню загального дизайну. Однак така простота додавання нових компонентів має свої негативні сторони: стає важко накласти обмеження на те, які об'єкти можуть входити до складу композиції. Іноді бажано, щоб складовий об'єкт міг містити лише певні види компонентів. Паттерн компонувальник не дозволяє скористатися для таких обмежень статичною системою типів. Натомість слід проводити перевірки під час виконання.

Реалізація. Питання, що виникають при реалізації:

- явні посилання на батьків. Зберігання в компоненті посилання на свого батька може спростити обхід структури та керування нею. Наявність такого посилання полегшує пересування вгору структурою та видалення компонента. Крім того, посилання на батьків допомагають підтримати патерн «ланцюжок обов'язків». Зазвичай посилання на батька визначають у класі Component. Класи Leaf і Composite можуть успадкувати саме посилання та операції з нею.

За наявності посилання на батька важливо підтримувати наступний інваріант: якщо деякий об'єкт у складовій структурі посилається на інший складовий об'єкт як на свого батька, то для останнього перший нащадок. Найпростіший спосіб гарантувати дотримання цієї умови - змінювати батька компонента лише тоді, коли він додається або видаляється зі складеного об'єкта. Якщо це вдається один раз реалізувати в операціях Add та Remove, то реалізація буде успадкована всіма підкласами і, отже, інваріант підтримуватиметься автоматично;

- поділ компонентів. Часто буває корисно розділяти компоненти, наприклад для зменшення обсягу займаної пам'яті. Але якщо у компонента може бути більше одного з батьків, то поділ стає проблемою. Можливе рішення – дозволити компонентам зберігати посилання на кількох батьків. Однак у разі поширення запиту структурою можуть виникнути неоднозначності. Паттерн пристосуванець показує, як слід змінити дизайн, щоб відмовитися від зберігання батьків. Працює він у тих випадках, коли нащадки можуть не надсилати повідомлень своїм батькам, винісши за свої межі частину внутрішнього стану;
- максимізація інтерфейсу класу Component. Одна з цілей патерна компонувальник - позбавити клієнтів необхідності знати, чи працюють вони з листовим або складовим об'єктом. Для досягнення цієї мети клас Component

повинен зробити якнайбільше операцій загальними для класів Component та Leaf. Зазвичай клас Component надає для цих операцій реалізації за умовчанням, а підкласи Component і Leaf заміщають їх.

Однак іноді ця мета входить у конфлікт із принципом проектування ієрархії класів, згідно з яким клас має визначати лише логічні для всіх його підкласів операції. Клас Component підтримує багато операцій, які не мають сенсу для класу Leaf. Як же тоді надати їм реалізацію за умовчанням?

Іноді вдається перенести до класу Component операцію, яка, на перший погляд, має сенс лише для складових об'єктів. Наприклад, інтерфейс для доступу до нащадків є фундаментальною частиною класу Component, але зовсім не обов'язково класу Leaf. Однак якщо розглядати Leaf як Component, у якого ніколи не буває нащадків, то ми можемо визначити в класі СошропепР операцію доступу до нащадків як таку, яка ніколи не повертає нащадків. Тоді підкласи Leaf можуть використовувати цю реалізацію за умовчанням, а в підкласах Composite вона буде перевизначена, щоб повертати нащадків. Операції для управління нащадками досить клопіткі.

- оголошення операцій для управління нащадками. Хоча у класі Composite реалізовані операції Add і Remove для додавання і видалення нащадків, але для патерна компоновщик важливо, у яких класах ці операції оголошені. Чи треба оголошувати їх у класі Component і тим самим робити доступними в Leaf, чи їх слід оголосити та визначити лише у класі Composite та його підкласах?

Вирішуючи це питання, ми маємо обирати між безпекою та прозорістю:

- якщо визначити інтерфейс керувати нащадками докорінно ієрархії класів, ми домагаємося прозорості, оскільки всі компоненти вдається трактувати однаково. Однак розплачуватися доводиться безпекою, оскільки клієнт може спробувати виконати безглузду дію, наприклад, додати або видалити об'єкт з листового вузла;

- якщо керування нащадками зробити частиною класу Composite, то безпеку вдасться забезпечити, адже будь-яка спроба додати або видалити об'єкти з листів статично типізованою мовою на зразок C++ буде перехоплена на етапі компіляції. Але прозорість ми втрачаємо, бо у листових та складових об'єктів виявляються різні інтерфейси.

У патерні компонувальник ми надаємо особливого значення прозорості, а не безпеці. Якщо вам важливіше безпека, будьте готові до того, що іноді ви можете втратити інформацію про тип і доведеться перетворювати компонент до типу складеного об'єкта. Як це зробити, не вдаючись до небезпечних наведень типів?

Можна, наприклад, оголосити у класі Component операцію Composite* GetComposite(). Клас Component реалізує її за умовчанням, повертаючи нульовий покажчик. А в класі Composite ця операція перевизначається і повертає вказівку на сам об'єкт:

Зрозуміло, за такого підходу ми поводимося з усіма компонентами однаково, що погано. Знову доводиться перевіряти тип, перед тим як зробити ту чи іншу дію.

Єдиний спосіб забезпечити прозорість - це включити до класу Component

реалізації операцій Add і Remove за умовчанням. Але з'явиться нова проблема: не можна реалізувати Component::Add так, щоб вона ніколи не призводила до помилки. Можна, звичайно, зробити цю операцію порожньою, але тоді порушується важливе проектне обмеження: спроба додати щось до листового об'єкта, швидше за все, свідчить про помилку. Допустимо було б змусити її видаляти свій аргумент, але клієнт може бути не розрахованим на це.

Зазвичай найкращим рішенням є така реалізація Add і Remove за умовчанням, коли вони завершуються з помилкою (можливо, throw), якщо компоненту заборонено мати нащадків (для Add) чи аргумент не є чиїмось нащадком (для Remove).

Інша можливість - трохи змінити семантику операції «видалення». Якщо компонент зберігає посилання на батька, то можна було б вважати, що Component::Remove видаляє самого себе. Але для операції Add, як і раніше, немає розумної інтерпретації;

- чи Component повинен реалізовувати список компонентів. Може виникнути бажання визначити безліч нащадків у вигляді змінної екземпляра класу Component, в якому оголошено операції доступу та управління нащадками. Але розміщення вказівки на нащадків у базовому класі призводить до непродуктивного витрати пам'яті в усіх листових вузлах, хоча в листа нащадків бути не може. Такий прийом можна застосувати, тільки якщо в структурі не надто багато нащадків;

- упорядкування нащадків. У багатьох випадках порядок слідування нащадків складеного об'єкта важливий. У розглянутому вище прикладі класу Graphic під порядком можна розуміти Z-порядок розташування нащадків. У складових об'єктах, що описують дерева синтаксичного розбору, складові оператори можуть бути екземплярами класу Composite, порядок прямування нащадків яких відображає семантику програми.

Якщо порядок нащадків важливий, необхідно враховувати його при проектуванні інтерфейсів доступу та управління нащадками. У цьому може допомогти патерн ітератор;

- кешування для підвищення продуктивності. Якщо доводиться часто обходити композицію або проводити в ній пошук, то клас Composite може кешувати інформацію про обхід та пошук. Кешувати дозволяється або отримані результати, або тільки достатню інформацію для прискорення обходу або пошуку.

Будь-яка зміна компонента повинна робити кеші всіх батьків недійсними. Найбільш ефективний такий підхід у випадку, коли компонентам відомо про їхніх батьків. Тому, якщо ви вирішите скористатися кешуванням, необхідно визначити інтерфейс, що дозволяє повідомити складові об'єкти про недійсність кешів;

- хто повинен видаляти компоненти. У мовах, де немає збирача сміття, найкраще доручити класу Composite видаляти своїх нащадків у момент знищення. Винятком із цього правила є випадок, коли листові об'єкти постійні і, отже, можуть розділятися;

- яка структура даних найкраще підходить для зберігання компонентів.

Складові об'єкти можуть зберігати своїх нащадків у різних структурах даних, включаючи зв'язані списки, дерева, масиви і хеш-таблиці. Вибір структури даних визначається, як завжди, ефективністю. Власне, зовсім не обов'язково користуватися якоюсь із універсальних структур. Іноді в складових об'єктах кожен нащадок є окремою змінною. Щоправда, для цього кожен підклас Composite має реалізовувати свій власний інтерфейс управління пам'яттю.

Уявімо приклад, що ми створюємо веб-сервер, який на вимогу клієнта повинен віддавати html-text. Html сторінка містить форму, яка може мати дочірні елементи полів вводу тексту та олів вводу чисел. Тоді форма – це композитний єлемент, а полі вводу це компоненти. Наступний приклад демонструє отримання строки зі вмістом html-тексту, який містить форму, що складається з трьох полів введення.

```
#include <iostream>
#include <vector>

// Create an "interface" (lowest common denominator)
class Component
{
public:
  virtual ~Component() { }
  virtual std::string render() = 0;
};


class InputLeaf : public Component
{
  // 1. Scalar class   3. "isa" relationship
  std::string label;
public:
  InputLeaf(std::string l = "")
  {
        label = l;
  }

  virtual std::string render()
  {
        std::string result;
        if (label.length() > 0) {
              result += "<label>";
              result += label;
              result += "</label>";
        }
        result += "<input type='text' />";
        return result;
  }
```

```
};

class NumberInputLeaf : public Component
{
  // 1. Scalar class   3. "isa" relationship
  std::string label;
  int min, max;
public:
  NumberInputLeaf(std::string l = "")
  {
        label = l;
        min = max = -1;
  }

  void setMin(int m) { min = m; }
  void setMax(int m) { max = m; }

  virtual std::string render()
  {
        std::string result;
        if (label.length() > 0) {
              result += "<label>";
              result += label;
              result += "</label>";
        }

        char buf[30];
        result += "<input type='number'";
        if (min >= 0) {
              result += " min=";
              sprintf_s(buf,30, "%d", min);
              result += buf;
        }
        if (max >= 0) {
              result += " max=";
              sprintf_s(buf, 30, "%d", max);
              result += buf;
        }
        result += " />";
        return result;
  }
};

class FormComposite : public Component
{
```

```
  // 1. Vector class   3. "isa" relationship
  std::vector < Component * > children; // 4. "container" coupled to the interface
  std::string method;
  std::string action;
public:
  FormComposite() {
        method = "POST";
  }

  ~FormComposite() {
        for (int i = 0; i < children.size(); i++) {
              delete children[i];
        }
  }

  void setMethod(std::string m) { method = m; }
  void setAction(std::string a) { action = a; }

  // 4. "container" class coupled to the interface
  void add(Component *ele)
  {
        children.push_back(ele);
  }
  std::string render()
  {
        std::string result = "<form method='";
        result += method;
        result += "'";
        if (action.length() > 0) {
              result += " action='";
              result += action;
              result += "'";
        }
        result += ">\n";

        for (int i = 0; i < children.size(); i++) {
              // 5. Use polymorphism to delegate to children
              result += children[i]->render();
              result += "\n";
        }
        result += "</form>\n";
        return result;
  }
};
```

```
int main()
{
  FormComposite * form = new FormComposite();
  form->add(new InputLeaf("Firstname"));
  form->add(new InputLeaf("Lastname"));
  NumberInputLeaf * c = new NumberInputLeaf("Age");
  c->setMin(18);
  form->add(c);

  std::cout << form->render().c_str();
  delete form;
}
```

3.5. Патерн Decorator.

Призначення. Динамічно додає об'єкту нові обов'язки. Є гнучкою альтернативою породженню підкласів із метою розширення функціональності.

Мотивація. Іноді потрібно покласти додаткові обов'язки на окремий об'єкт, а не на клас в цілому. Так, бібліотека для побудови графічних інтерфейсів користувача повинна «вміти» додавати нову властивість, скажімо, рамку або нову поведінку (наприклад, можливість прокручування будь-якого елемента інтерфейсу).

Додати нові обов'язки можна за допомогою успадкування. При наслідуванні класу з рамкою навколо кожного екземпляра підкласу буде малюватись рамка. Однак це рішення статичне, а отже, недостатньо гнучке. Клієнт не може керувати оформленням компонента рамкою.

Гнучкішим є інший підхід: помістити компонент в інший об'єкт, званий декоратором, який якраз і додає рамку. Декоратор слідує інтерфейсу об'єкта, що декорується, тому його присутність прозора для клієнтів компонента. Декоратор переадресує запити внутрішньому компоненту, але може виконувати додаткові дії (наприклад, малювати рамку) до або після переадресації. Оскільки декоратори прозорі, вони можуть вкладатися один в одного, додаючи цим будь-яку кількість нових обов'язків.

Припустимо, що є об'єкт класу TextView, який відображає текст у вікні. За замовчуванням TextView не має смуг прокручування, оскільки вони не завжди потрібні. Але за потреби їх вдасться додати за допомогою декоратора ScrollDecorator. Припустимо, що ми хочемо додати жирну суцільну рамку навколо об'єкта TextView. Тут може допомогти декоратор BorderDecorator. Ми просто компонуємо обидва декоратори з BorderDecorator і отримуємо результат.

Класи ScrollDecorator та BorderDecorator є підкласами Decorator - абстрактного класу, який представляє візуальні компоненти, що застосовуються для оформлення інших візуальних компонентів.

VisualComponent - це абстрактний клас для представлення візуальних об'єктів. У ньому визначено інтерфейс для малювання та обробки подій. Зазначимо, що клас Decorator просто переадресує запити на малювання своєму

компоненту, а його підкласи можуть розширювати цю операцію.

Підкласи Decorator можуть додавати будь-які операції для забезпечення необхідної функціональності. Так, операція ScrollTo об'єкта ScrollDecorator дозволяє іншим об'єктам виконувати прокручування, якщо їм відомо про присутність об'єкта ScrollDecorator. Важлива особливість цього патерну полягає в тому, що декоратори можуть вживатися скрізь, де можлива поява самого об'єкта VisualComponent. Тому клієнт не може відрізнити декорований об'єкт від недекорованого, а отже, і аж ніяк не залежить від наявності чи відсутності оформлень.

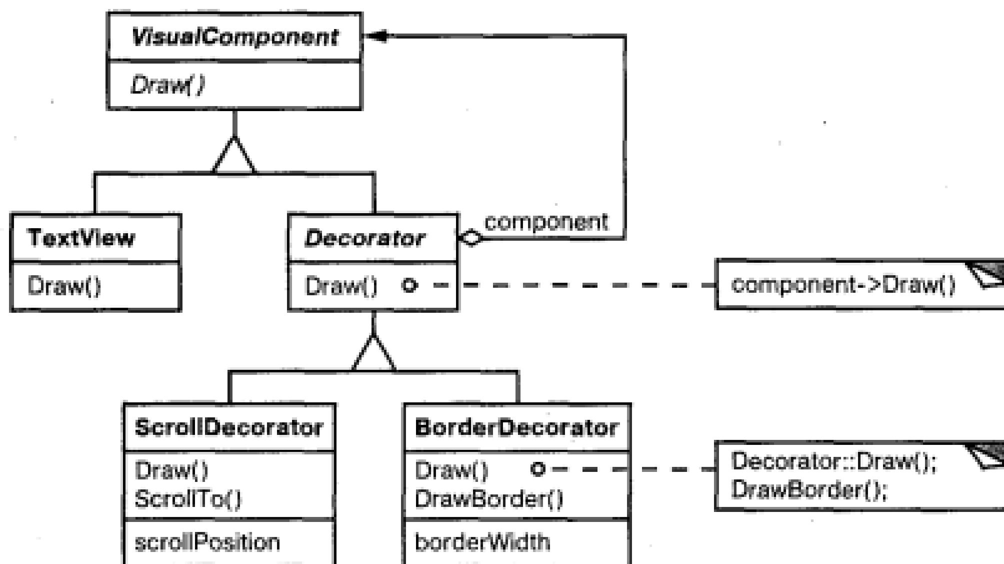

Рис. 3.7. Приклад взаємодії класів у патерні Decorator

Застосування. Використовуйте патерн декоратор:

- для динамічного, прозорого для клієнтів додавання обов'язків об'єктам;
- для реалізації обов'язків, які можуть бути зняті з об'єкта;
- коли розширення шляхом породження підкласів з якихось причин незручне чи неможливе. Іноді доводиться реалізовувати багато незалежних розширень, отже породження підкласів підтримки всіх можливих комбінацій призведе до комбінаторного зростання їх числа. В інших випадках визначення класу може бути приховане або ще недоступне, так що породити від нього підклас не можна.

Учасники:

Component (VisualComponent) - компонент: визначає інтерфейс для об'єктів, на які можуть бути динамічно покладені додаткові обов'язки;

ConcreteComponent (TextView) - конкретний компонент: визначає об'єкт, на який покладаються додаткові обов'язки;

Decorator - декоратор: зберігає посилання на об'єкт Component та визначає інтерфейс, що відповідає інтерфейсу Component;

ConcreteDecorator (BorderDecorator, ScrollDecorator) - конкретний декоратор: покладає додаткові обов'язки на компонент.

Структура.

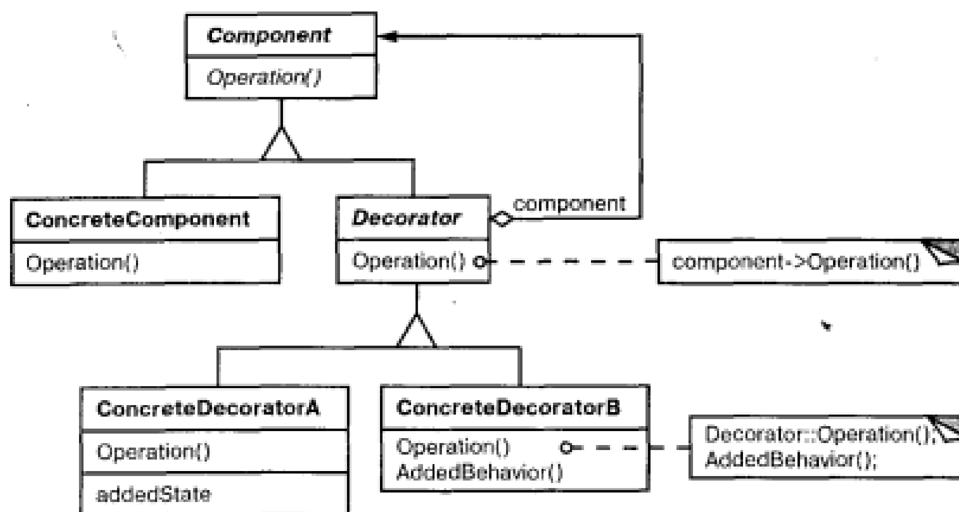

Рис. 3.8. Структура патерну Decorator

Відносини. Decorator переадресує запити об'єкту Component. Може виконувати додаткові операції до і після переадресації.

Результати.

Патерн декоратор має принаймні два плюси і два мінуси:

- більша гнучкість, ніж у статичного спадкування. Паттерн декоратор дозволяє гнучкіше додавати об'єкту нові обов'язки, ніж було б можливо у разі статичного (множинного) успадкування. Декоратор може додавати та видаляти обов'язки під час виконання програми. При використанні успадкування потрібно створювати новий клас для кожного додаткового обов'язку (наприклад, BorderScrollableDecorator, BorderedTextView), що веде до збільшення числа класів і, як наслідок, до зростання складності системи. Крім того, застосування кількох декораторів одного компоненту дозволяє довільним чином поєднувати обов'язки.

Декоратори дозволяють легко додати одну і ту ж властивість двічі. Наприклад, щоб оточити об'єкт TextView подвійною рамкою, потрібно просто додати два декоратори BorderDecorators. Подвійне успадкування класу Border у кращому разі загрожує помилками;

- дозволяє уникнути перевантажених функціями класів на верхніх рівнях ієрархії. Декоратор дозволяє додавати нові обов'язки за необхідності. Замість того, щоб намагатися підтримати всі можливі можливості в одному складному класі, що допускає різнобічне налаштування, ви можете визначити простий клас і поступово нарощувати його функціональність за допомогою декораторів. В результаті програма вже не платить за функції, що не використовуються. Неважко також визначати нові види декораторів незалежно від класів, які вони розширюють, навіть якщо такі розширення не планувалися. При розширенні ж складного класу зазвичай доводиться вдаватись в деталі, що не мають відношення до функції, що додається;

- декоратор та його компонент не ідентичні. Декоратор діє як прозоре обрамлення. Але декорований компонент все ж таки не ідентичний вихідному.

При використанні декораторів це слід пам'ятати;

- безліч дрібних об'єктів. При використанні в проекті патерну декоратор нерідко виходить система, складена з великої кількості дрібних об'єктів, які схожі один на одного і відрізняються лише способом взаємозв'язку, а не класом і значення своїх внутрішніх змінних. Хоча проектувальник, який знається на структурі такої системи, може легко налаштувати її, але вивчати і налагоджувати її дуже важко.

Реалізація. Застосування патерну декоратор потребує розгляду кількох питань:

- відповідність інтерфейсів. Інтерфейс декоратора повинен відповідати інтерфейсу компонента, що декорується. Тому класи ConcreteDecorator повинні успадковувати загальному класу;
- відсутність абстрактного класу Decorator. Немає необхідності визначати абстрактний клас Decorator, якщо планується додати лише один обов'язок. Так часто відбувається, коли ви працюєте з вже існуючою ієрархією класів, а не проектуєте нову. У такому разі відповідальність за переадресацію запитів, яку зазвичай несе клас Decorator, можна покласти безпосередньо на ConcreteDecorator;
- полегшені класи Component. Щоб можна було гарантувати відповідність інтерфейсів, компоненти та декоратори мають наслідувати загальний клас Component. Важливо, щоб цей клас був настільки легким, наскільки це можливо. Іншими словами, він повинен визначати інтерфейс, а не зберігати дані. В іншому випадку декоратори можуть стати дуже важкими, і застосовувати їх у великій кількості буде невигідно. Включення великого числа функцій у клас Component також збільшує ймовірність, що конкретним підкласам доведеться платити за те, що їм не потрібно;
- зміна вигляду, а не внутрішнього облаштування об'єкту. Декоратор можна розглядати як оболонку, що з'явилася у об'єкта, яка змінює його поведінку. Альтернатива - зміна внутрішнього устрою об'єкта, хорошим прикладом чого може бути патерн стратегія.

Стратегії краще підходять у ситуаціях, коли клас Component вже досить важкий, так що застосування патерну декоратор обходиться занадто дорого. У патерні стратегія компоненти передають частину своєї функціональності окремому об'єкту-стратегії, тому змінити чи розширити поведінку компонента допустимо, замінивши цей об'єкт.

Наприклад, ми можемо підтримати різні стилі рамок, доручивши малювання рамки спеціальному об'єкту Border. Об'єкт Border є прикладом об'єкта-стратегії: у разі він інкапсулює стратегію малювання рамки. Число стратегій може бути будь-яким, тому ефект такий самий, як від рекурсивної вкладеності декораторів.

При використанні підходу, що базується на стратегіях, може виникнути необхідність модифікувати компонент, щоб він відповідав новому розширенню. З іншого боку, стратегія може мати свій власний спеціалізований інтерфейс, тоді як інтерфейс декоратора повинен повторювати інтерфейс компонента. Наприклад, стратегії малювання рамки необхідно визначити лише інтерфейс для

цієї операції (DrawBorder, GetWidth і т.д.), тобто клас стратегії може бути легким, незважаючи на важкість компонента.

Наступний приклад демонструє як реалізувати інтерфейс-користувача за допомогою декоратора.

```
#include <iostream>

// 1. "lowest common denominator"
class VisualComponent
{
public:
  virtual ~VisualComponent() { }
  virtual void draw() = 0;
};

class TextField : public VisualComponent
{
  // 3. "Core" class & "is a"
  int width, height;
public:
  TextField(int w, int h)
  {
        width = w;
        height = h;
  }
  ~TextField() { std::cout << "111"; }

  /*virtual*/
  void draw() {
        std::cout << "TextField: " << width << ", " << height << std::endl;
  }
};

// 2. 2nd level base class
class Decorator : public VisualComponent  // 4. "is a" relationship
{
  VisualComponent *widget; // 4. "has a" relationship
public:
  Decorator(VisualComponent *w)
  {
        widget = w;
  }

  virtual ~Decorator() {
        delete widget;
  }
```

```
  /*virtual*/
  void draw()
  {
          widget->draw(); // 5. Delegation
  }
};

class BorderDecorator : public Decorator
{
public:
  // 6. Optional embellishment
  BorderDecorator(VisualComponent *w) : Decorator(w) {}

  /*virtual*/
  void draw() {
          // 7. Delegate to base class and add extra stuff
          Decorator::draw();
          std::cout << "   BorderDecorator" << std::endl;
  }
};

class ScrollDecorator : public Decorator
{
public:
  // 6. Optional embellishment
  ScrollDecorator(VisualComponent *w) : Decorator(w) {}

  /*virtual*/
  void draw() {
          // 7. Delegate to base class and add extra stuff
          Decorator::draw();
          std::cout << "   ScrollDecorator" << std::endl;
  }
};

int main()
{
  // 8. Client has the responsibility to compose desired configurations
  VisualComponent *widget = new BorderDecorator(new BorderDecorator(new
ScrollDecorator
  (new TextField(80, 24))));
  widget->draw();
  delete widget;
}
```

3.6. Патерн Facade.

Призначення. Надає уніфікований інтерфейс замість набору деяких інтерфейсів підсистеми. Фасад визначає інтерфейс вищого рівня, який полегшує використання підсистеми.

Мотивація. Розбиття на підсистеми полегшує проектування складної системи загалом. Загальна мета будь-якого проектування - звести до мінімуму залежність підсистем одна від одної та обмін інформацією між ними. Один із способів вирішення цього завдання - введення об'єкту фасаду, що надає єдиний спрощений інтерфейс до складніших системних засобів.

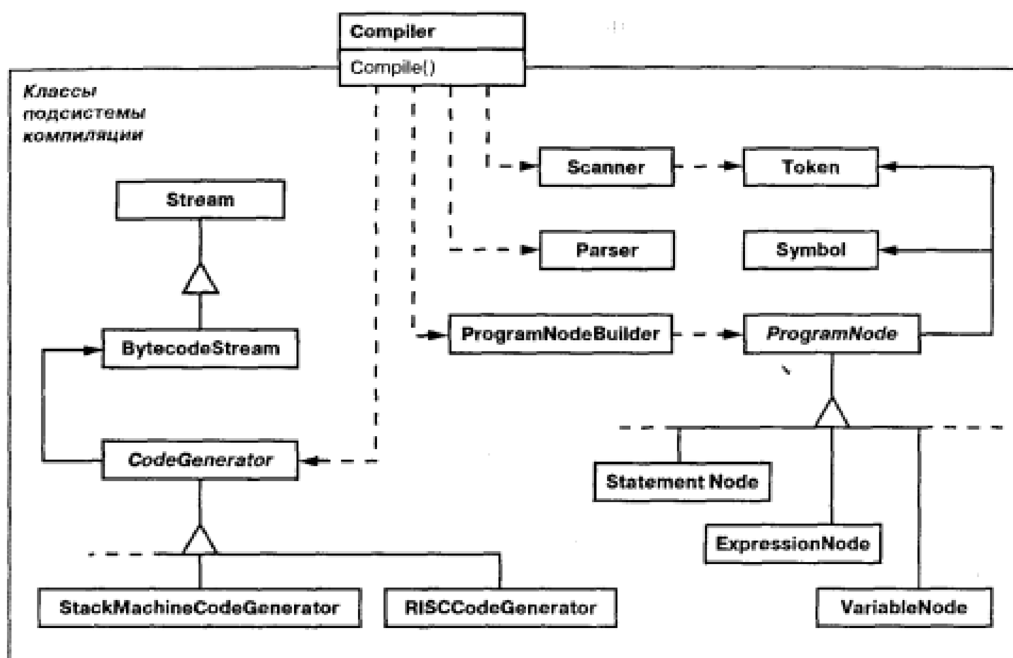

Рис. 3.9. Приклад взаємодії класів в патерні Facade

Розглянемо, наприклад, середовище програмування, яке дає програмам доступ до підсистеми компіляції. У цій підсистемі є такі класи, як Scanner (лексичний аналізатор), Parser (синтаксичний аналізатор), ProgramNode (вузол програми), BytecodeStream (потік байтових кодів) і ProgramNodeBuilder (будівельник вузла програми). Усі разом вони складають компілятор. Деяким спеціалізованим програмам, можливо, знадобиться прямий доступ до цих класів. Але більшість клієнтів компілятора такі деталі, як синтаксичний розбір і генерація коду, зазвичай не потрібні; їм просто потрібно відкомпілювати деяку програму. Для таких клієнтів застосування потужного, але низькорівневого інтерфейсу підсистеми компіляції лише ускладнює завдання.

Щоб надати інтерфейс вищого рівня, ізолюючий клієнта цих класів, в підсистему компіляції включено також клас Compiler (компілятор). Він визначає уніфікований інтерфейс для всіх можливостей компілятора. Клас Compiler виступає в ролі фасаду: пропонує простий інтерфейс до більш складної підсистеми. Він "склеює" класи, що реалізують функціональність компілятора, але не приховує їх повністю. Завдяки фасаду компілятора робота більшості

програмістів полегшується. При цьому ті, кому потрібний доступ до засобів низького рівня, не позбавляються його.

Застосування. Використовуйте патерн фасад, коли:

- хочете надати простий інтерфейс до складної системи. Часто підсистеми ускладнюються з розвитком. Застосування більшості патернів призводить до появи менших класів, але у більшій кількості. Таку підсистему простіше повторно використовувати та налаштовувати під конкретні потреби, але водночас застосовувати підсистему без налаштування стає складніше. Фасад пропонує деякий тип системи за замовчуванням, що влаштовує більшість клієнтів. І лише ті об'єкти, яким потрібні ширші можливості налаштування можуть звернутися безпосередньо до того, що знаходиться за фасадом;
- між клієнтами та класами реалізації абстракції існує багато залежностей. Фасад дозволить відокремити підсистему як від клієнтів, так і від інших підсистем, що, у свою чергу, сприяє підвищенню ступеня незалежності та переносимості;
- Ви бажаєте розкласти підсистему на окремі шари. Використовуйте фасад для визначення точки входу кожного рівня підсистеми. Якщо підсистеми залежать одна від одної, то залежність можна спростити, дозволивши підсистем обмінюватися інформацією лише через фасади.

Структура.

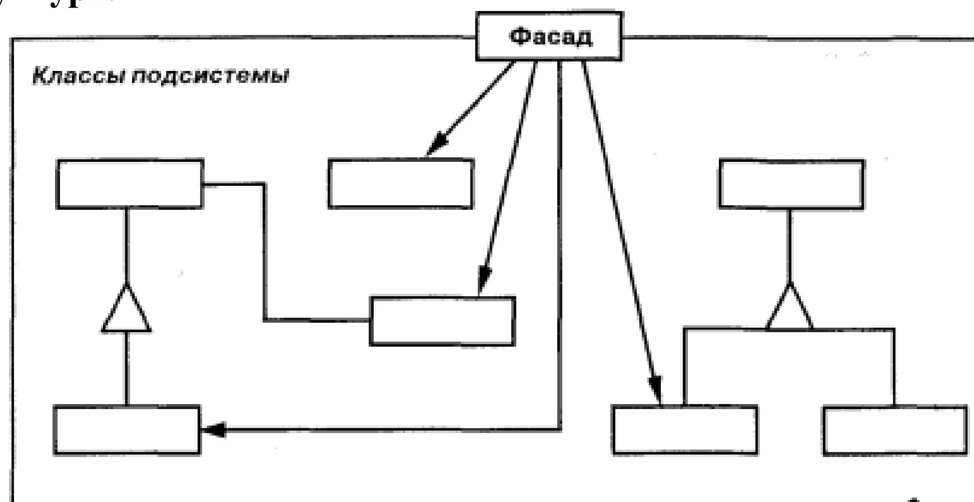

Рис. 3.10. Структура патерну Facade

Учасники:

- **Façade** (Compiler) - фасад: "знає", яким класам підсистеми адресувати запит; делегує запити клієнтів відповідним об'єктам усередині підсистеми;
- Класи підсистеми (Scaner, Parser, ProgramNode і т.д.): реалізують функціональність підсистеми; виконують роботу, доручену об'єктом Facade; нічого не знають про існування фасаду, тобто не зберігають посилань на нього.

Відносини.

Клієнти спілкуються з підсистемою, надсилаючи запити фасаду. Він переадресує їх відповідним об'єктам усередині підсистеми. Хоча основну роботу виконують саме об'єкти підсистеми, фасаду, можливо, доведеться перетворити

свій інтерфейс на інтерфейси підсистеми.

Клієнти, які мають фасад, не мають прямого доступу до об'єктів підсистеми.

Результати. Патерн фасад має такі переваги:

- ізолює клієнтів від компонентів підсистеми, зменшуючи тим самим кількість об'єктів, з якими клієнтам доводиться мати справу, та спрощуючи роботу з підсистемою;
- дозволяє послабити зв'язаність між підсистемою та її клієнтами. Найчастіше компоненти підсистеми дуже пов'язані. Слабка зв'язаність дозволяє видозмінювати компоненти, не торкаючись при цьому клієнтів. Фасади допомагають розкласти систему на шари та структурувати залежності між об'єктами, а також уникнути складних та циклічних залежностей. Це може бути важливим, якщо клієнт та підсистема реалізуються незалежно. Зменшення числа залежностей на стадії компіляції надзвичайно важливе у великих системах. Хочеться, звичайно, щоб час, що йде на перекомпіляцію після зміни класів підсистеми, був мінімальним. Скорочення числа залежностей за рахунок фасадів може зменшити кількість файлів, які потребують повторної компіляції, після невеликої модифікації якоїсь важливої підсистеми. Фасад може спростити процес перенесення системи на інші платформи, оскільки зменшується ймовірність того, що в результаті зміни однієї підсистеми знадобиться змінювати і всі інші;
- фасад не перешкоджає додаткам безпосередньо звертатися до класів підсистеми, якщо це необхідно. Таким чином, у вас є вибір між простотою та спільністю.

Ревлізація. При реалізації фасаду слід звернути увагу на такі питання:

- зменшення ступеня пов'язаності клієнта із підсистемою. Ступінь зв'язаності можна значно зменшити, якщо зробити клас Facade абстрактним. Його конкретні підкласи відповідатимуть різним реалізаціям підсистеми. Тоді клієнти зможуть взаємодіяти із підсистемою через інтерфейс абстрактного класу Facade. Це ізолює клієнтів від інформації, яка реалізація підсистеми використовується. Замість породження підкласів можна налаштувати об'єкт Facade різними об'єктами підсистем. Для налаштування фасаду достатньо замінити один або декілька таких об'єктів;
- відкриті та закриті класи підсистем. Підсистема схожа на клас у тому відношенні, що в обох є інтерфейси і обидва інкапсулюють. Клас інкапсулює стан та операції, а підсистема – класи. І якщо корисно розрізняти відкритий і закритий інтерфейси класу, то не менш розумно говорити про відкритий і закритий інтерфейс підсистеми. Відкритий інтерфейс підсистеми складається із класів, до яких мають доступ усі клієнти; закритий інтерфейс доступний лише розширення підсистеми. Клас Facade, звичайно, є частиною відкритого інтерфейсу, але це не єдина частина. Інші класи підсистеми можуть бути відкритими. Наприклад, у системі компіляції класи Parser та Scanner – частина відкритого інтерфейсу.

Наступний приклад демонструє фасад для системи компіляції.

```
#include <iostream>

class Scanner {
public:
  Scanner(std::istream& is): _inputStream (is) {   }
  virtual ~Scanner() {}

  virtual void Scan() { std::cout << "scan" << std::endl; }
private:
  std::istream& _inputStream;
};


class BytecodeStream {

};


class StatementNode {

};

class ExpressionNode {

};

class CodeGenerator {
public:
  virtual void Visit(StatementNode*) {}
  virtual void Visit(ExpressionNode*) {}
  // ...
protected:
  CodeGenerator(BytecodeStream& o): _output(o) {   }
protected:
  BytecodeStream& _output;
};

class RISCCodeGenerator : public CodeGenerator {
public:
  RISCCodeGenerator(BytecodeStream& o): CodeGenerator(o){}
};

class ProgramNode {
public:
  // program node manipulation
```

```
  virtual void GetSourcePosition(int& line, int& index) {}
  // ...

  // child manipulation
  virtual void Add(ProgramNode*) {}
  virtual void Remove(ProgramNode*) {}
  // ...

  virtual void Traverse(CodeGenerator&) { std::cout << "traverse" << std::endl; }

  ProgramNode() {}
};



class ProgramNodeBuilder {
public:
  ProgramNodeBuilder() { _node = new ProgramNode(); }

  virtual ProgramNode* NewVariable(
          const char* variableName
  ) const {
          return 0;
  }

  virtual ProgramNode* NewAssignment(
          ProgramNode* variable, ProgramNode* expression
  ) const {
          return 0;
  }

  virtual ProgramNode* NewReturnStatement(
          ProgramNode* value
  ) const {
          return 0;
  }

  virtual ProgramNode* NewCondition(
          ProgramNode* condition,
          ProgramNode* truePart, ProgramNode* falsePart
  ) const {
          return 0;
  }
  // ...
```

```
  ProgramNode* GetRootNode() { return _node; }
private:
  ProgramNode* _node;
};

class Parser {
public:
  Parser() {}
  virtual ~Parser() {}

  virtual void Parse(Scanner&s, ProgramNodeBuilder&) {
          std::cout << "parse" << std::endl;
          s.Scan();
  }
};

class Compiler {
public:
  Compiler() {}

  virtual void Compile(std::istream&, BytecodeStream&);
};

void Compiler::Compile(
  std::istream& input, BytecodeStream& output
) {
  Scanner scanner(input);
  ProgramNodeBuilder builder;
  Parser parser;

  parser.Parse(scanner, builder);

  RISCCodeGenerator generator(output);
  ProgramNode* parseTree = builder.GetRootNode();
  parseTree->Traverse(generator);
}

int main()
{
  Compiler c;
  BytecodeStream output;
  c.Compile(std::cin, output);
}
```

3.7. Патерн Flyweight.

Призначення. Використовує поділ для ефективної підтримки багатьох дрібних об'єктів.

Мотивація. У деяких додатках використання об'єктів могло бути дуже корисним, але прямолінійна реалізація виявляється неприпустимо марнотратною.

Наприклад, у більшості редакторів документів є засоби форматування та редагування текстів, в тій чи іншій мірі модульні. Об'єктно-орієнтовані редактори зазвичай застосовують об'єкти для представлення таких вбудованих елементів, як таблиці та малюнки. Але вони не використовують об'єкти для представлення кожного символу, незважаючи на те, що це збільшило б гнучкість на нижніх рівнях програми. Адже тоді до малювання та форматування символів та вбудованих елементів можна було б застосувати одноманітний підхід. І для підтримки нових наборів символів не довелося б зачіпати інші функції редактора. Та й загальна структура програми відображала б фізичну структуру документа. На наступній діаграмі показано, як редактор документів міг би скористатися об'єктами для представлення символів.

Такий дизайн має один недолік - вартість. Навіть у документі скромних розмірів було б кілька сотень тисяч об'єктів-символів, а це призвело б до витрачання величезного обсягу пам'яті та неприйнятних витрат під час виконання. Паттерн адаптер показує, як розділяти дуже дрібні об'єкти без неприпустимо високих витрат.

Пристосуванець - це об'єкт, що розділяється, який можна використовувати одночасно в декількох контекстах. У кожному контексті він виглядає як незалежний об'єкт, тобто не відрізняється від екземпляра, який не поділяється. Пристосуванці не можуть робити припущень про контекст, у якому працюють.

Ключова ідея тут – відмінність між внутрішнім та зовнішнім станами. Внутрішній стан зберігається в самому пристосуванці і складається з інформації, яка не залежить від його контексту. Саме тому вона може розділятися. Зовнішній стан залежить від контексту та змінюється разом з ним, тому не підлягає поділу. Об'єкти-клієнти відповідають за передачу зовнішнього стану пристосуванцю, коли в цьому виникає потреба.

Пристосуванці моделюють концепції або сутності, кількість яких занадто велика для представлення об'єктами. Наприклад, редактор документів міг би створити по одному пристосуванцю кожної літери алфавіту. Кожен пристосуванець зберігає код символу, але координати положення символу в документі та стиль його зображення визначаються алгоритмами розміщення тексту та командами форматування, що діють у тому місці, де символ з'являється. Код символу – це внутрішній стан, а решта – зовнішній.

Логічно для кожного входження даного символу документ існує об'єкт.

Фізично, однак, є лише по одному об'єкту-пристосуванцю для кожного символу, який з'являється у різних контекстах у структурі документа. Кожне входження даного об'єкта-символу посилається на один і той же екземпляр у пулі об'єктів-пристосуванців, що розділяється.

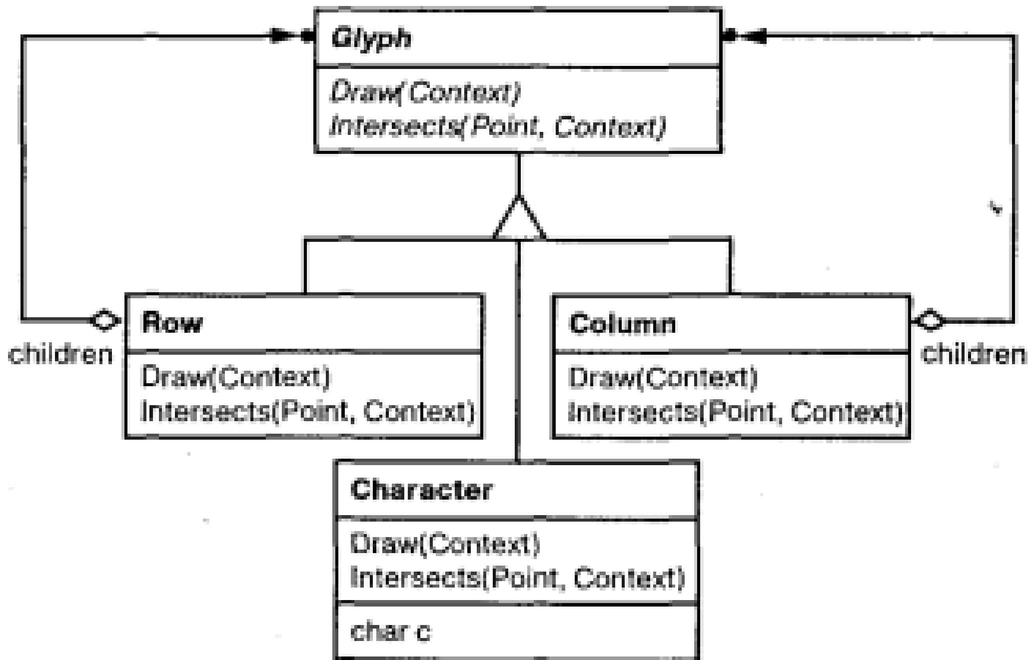

Рис. 3.11. Приклад взаємодії класів в патерні Flyweight

Glyph - це абстрактний клас для представлення графічних об'єктів (деякі з них можуть бути пристосуванцями). Операції, які можуть залежати від зовнішнього стану, передають його як параметр. Наприклад, операціям Draw (малювання) і Intersects (перетин) має бути відомо, в якому контексті зустрічається гліф, інакше вони не зможуть виконати те, що від них вимагається.

Пристосуванець, що представляє букву «а», містить лише відповідний код; ні положення, ні шрифт літери йому не треба зберігати. Клієнти передають пристосуванцю всю залежну від контексту інформацію, яка потрібна, щоб він міг зобразити себе. Наприклад, гліфу Row відомо, де його нащадки повинні себе показати, щоб це виглядало як горизонтальний рядок. Тому разом із запитом на малювання він може передавати кожному нащадку координати.

Оскільки кількість різних об'єктів-символів набагато менше, ніж кількість символів у документі, то й загальна кількість об'єктів істотно менша, ніж було б при простій реалізації. Документ, у якому всі символи зображуються одним шрифтом і кольором, створить близько 100 об'єктів-символів (це приблизно дорівнює кількості кодів у таблиці ASCII) незалежно від свого розміру. А оскільки у більшості документів застосовується не більше десятка різних комбінацій шрифту та кольору, то на практиці ця величина зросте неістотно. Тому абстракція об'єкта міє можливість для застосування і до окремих символів.

Застосування. Ефективність патерна пристосуванець багато в чому залежить від того, як і де він використовується. Застосовуйте цей патерн, коли виконані всі такі умови:

- у додатку використовується велика кількість об'єктів;
- через це накладні витрати на зберігання високі;
- більшу частину стану об'єктів можна винести назовні;
- багато груп об'єктів можна замінити відносно невеликою кількістю

об'єктів, що розділяються, оскільки зовнішній стан винесено;

- програма не залежить від ідентичності об'єкта. Оскільки об'єкти-пристосуванці можуть розділятися, то перевірка на ідентичність поверне «істину» концептуально різних об'єктів.

Структура.

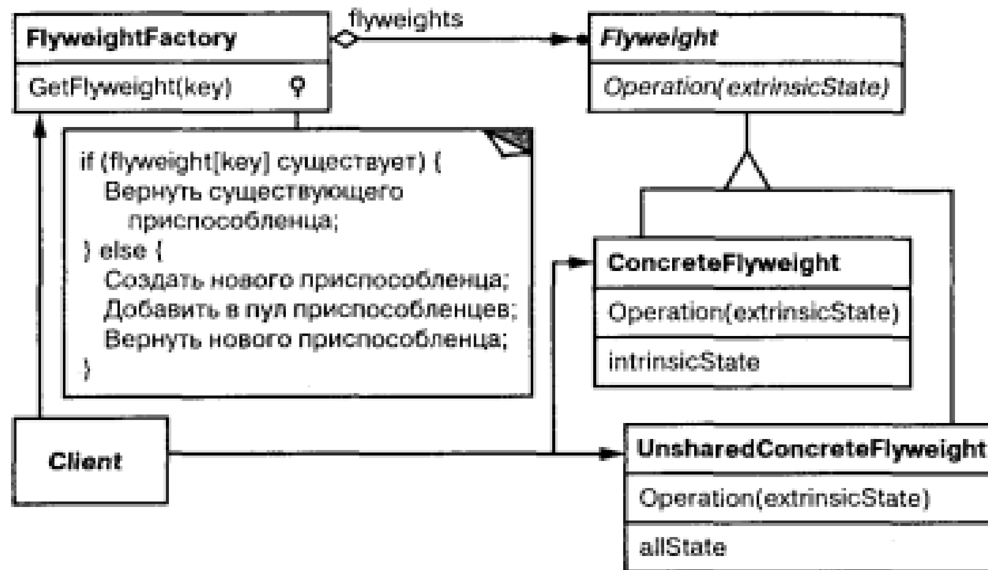

Рис. 3.12. Структура патерну Flyweight

Учасники:

- **Flyweight** (Glyph) - пристосуванець: оголошує інтерфейс, за допомогою якого пристосуванці можуть отримувати зовнішній стан або впливати на нього;
- **ConcreteFlyweight** (Character) - конкретний пристосуванець:реалізує інтерфейс класу Flyweight і додає за потреби внутрішній стан. Об'єкт класу ConcreteFlyweight; має бути розділеним. Будь-який стан, що зберігається ним, повинен бути внутрішнім, тобто не залежним від контексту;iiii
- **UnsharedConcreteFlyweight** (Row, Column) - конкретний нерозділений пристосуванець: не всі підкласи Flyweight обов'язково повинні бути поділені. Інтерфейс Flyweight допускає розподіл, але не нав'язує його. Часто у об'єктів UnsharedConcreteFlyweight на деякому рівні структури пристосуванця є нащадки у вигляді об'єктів класу ConcreteFlyweight, як, наприклад, в об'єктів класів Row і Column;
- **FlyweightFactory** - фабрика пристосуванців: створює об'єкти-пристосуванці та керує ними; забезпечує належне поділ пристосуванців. Коли клієнт запитує пристосуванця, об'єкт FlyweightFactory надає існуючий екземпляр або створює новий, якщо готового ще немає;
- **Client** - клієнт: зберігає посилання на одного або декількох пристосуванців; обчислює чи зберігає зовнішній стан пристосуванців.

Відносини.

− стан, необхідний для нормальної роботи, можна охарактеризувати як внутрішній або зовнішній. Перше зберігається в об'єкті ConcreteFlyweight.

Зовнішній стан зберігається чи обчислюється клієнтами. Клієнт передає його пристосуванцю під час виклику операцій;

– клієнти не повинні створювати екземпляри класу ConcreteFlyweight безпосередньо, а можуть отримувати їх тільки від об'єкта FlyweightFactory. Це дозволить гарантувати коректний поділ.

Результати.

При використанні пристосуванців не виключено витрати на передачу, пошук або обчислення внутрішнього стану, особливо якщо раніше воно зберігалося як внутрішній. Однак такі витрати з лишком компенсуються економією пам'яті за рахунок поділу об'єктів-пристосуванців.

Економія пам'яті виникає з низки причин:

- зменшення загальної кількості екземплярів;
- скорочення обсягу пам'яті, необхідного для зберігання внутрішнього стану;
- обчислення, а не зберігання зовнішнього стану (якщо це дійсно так).

Чим вищий ступінь поділу пристосуванців, тим істотніша економія.

Зі збільшенням обсягу поділюваного стану економія також зростає. Найбільшого ефекту вдається досягти, коли сумарний обсяг внутрішньої та зовнішньої інформації про стан великий, а зовнішній стан обчислюється, а чи не зберігається. Тоді поділ зменшує вартість зберігання внутрішнього стану, а рахунок обчислень скорочується пам'ять під зовнішній стан.

Паттерн пристосуванець часто застосовується разом з компонувальником для представлення ієрархічної структури у вигляді графу з листовими вузлами, що розділяються. Через поділ вказівка на батька не може зберігатися в листовому вузлі-пристосуванці, а повинна передаватися йому як частина зовнішнього стану. Це помітно впливає на спосіб взаємодії об'єктів ієрархії між собою.

Реалізація.

При реалізації пристосуванця слід звернути увагу на такі питання:

- винесення зовнішнього стану. Застосовність патерну значною мірою залежить від того, наскільки легко ідентифікувати зовнішній стан і винести його за межі об'єктів, що розділяються. Винесення зовнішнього стану не зменшує вартості зберігання, якщо різних зовнішніх станів так само багато, як об'єктів до поділу. Найкращий варіант - зовнішній стан обчислюється по об'єктах з іншою структурою, що вимагає значно меншої пам'яті.
- управління об'єктами, що розділяються. Оскільки об'єкти поділяються, клієнти не повинні інстанцувати їх безпосередньо. Фабрика FlyweightFactory дозволяє клієнтам знайти відповідного пристосуванця. В об'єктах цього класу часто є сховище, організоване як асоціативний масив, за допомогою якого можна швидко знаходити пристосуванця, потрібного клієнту.

```
#include <iostream>
#include <string>
#include <map>

// The 'Flyweight' interface
```

```
class IShape
{
public:
  virtual void Draw() = 0;
};

// A 'ConcreteFlyweight' class
class Rectangle : public IShape
{
  public:
          virtual void Draw() {
                  std::cout << "Draw Rectangle" << std::endl;
          }
};

// A 'ConcreteFlyweight' class
class Circle : public IShape
{
public:
  virtual void Draw() {
          std::cout << "Draw Circle" << std::endl;
  }
};

// The 'FlyweightFactory' class
class ShapeObjectFactory
{
protected:

  static std::map<std::string, IShape*> shapes;

  ShapeObjectFactory() {}

public:

  static int GetTotal() { return shapes.size(); }

  static IShape * GetShape(std::string ShapeName)
  {


          if (shapes.find(ShapeName) != shapes.end())
          {
                  return shapes[ShapeName];
          }
```

```
		IShape * shape = nullptr;
		if (ShapeName == "Rectangle") {
			shape = new Rectangle();
		} else if (ShapeName == "Circle") {
			shape = new Circle();
		}
		else {
			throw "Factory cannot create the object specified";
		}
		shapes[ShapeName] = shape;
		return shape;
	}

	static void free() {

		for (auto it = shapes.begin(); it != shapes.end(); ++it) {
			delete it->second;
		}
		shapes.clear();
	}
};

std::map<std::string, IShape*> ShapeObjectFactory::shapes;

int main()
{

	IShape * shape = ShapeObjectFactory::GetShape("Rectangle");
	shape->Draw();
	shape = ShapeObjectFactory::GetShape("Rectangle");
	shape->Draw();
	shape = ShapeObjectFactory::GetShape("Rectangle");
	shape->Draw();

	shape = ShapeObjectFactory::GetShape("Circle");
	shape->Draw();
	shape = ShapeObjectFactory::GetShape("Circle");
	shape->Draw();
	shape = ShapeObjectFactory::GetShape("Circle");
	shape->Draw();
	std::cout << ShapeObjectFactory::GetTotal();
	ShapeObjectFactory::free();
	return 0;
}
```

3.8. Патерн Proxy.

Призначення. Є сурогатом іншого об'єкта та контролює доступ до нього.

Мотивація. Розумно керувати доступом до об'єкта, оскільки тоді можна відкласти витрати на створення та ініціалізацію до моменту, коли об'єкт справді знадобиться. Розглянемо редактор документів, який допускає вбудовування в документ графічних об'єктів. Витрати створення деяких таких об'єктів, наприклад великих растрових зображень, можуть бути дуже значні. Але документ має відкриватися швидко, тому слід уникати створення всіх «важких» об'єктів на стадії відкриття (та й взагалі це зайве, оскільки не всі вони будуть виводитись одночасно).

У зв'язку з такими обмеженнями здається розумним створювати «важкі» об'єкти на вимогу. Це означає «коли зображення стає видимим». Але що помістити у документ замість зображення? І як, не ускладнюючи реалізації редактора, приховати те, що зображення створюється на вимогу? Наприклад, оптимізація не повинна відображатися на коді, який відповідає за малювання та форматування.

Рішення полягає в тому, щоб використовувати інший об'єкт – заступник зображення, який тимчасово підставляється замість реального зображення. Заступник поводиться так само, як саме зображення, і виконує при необхідності його інстанціювання.

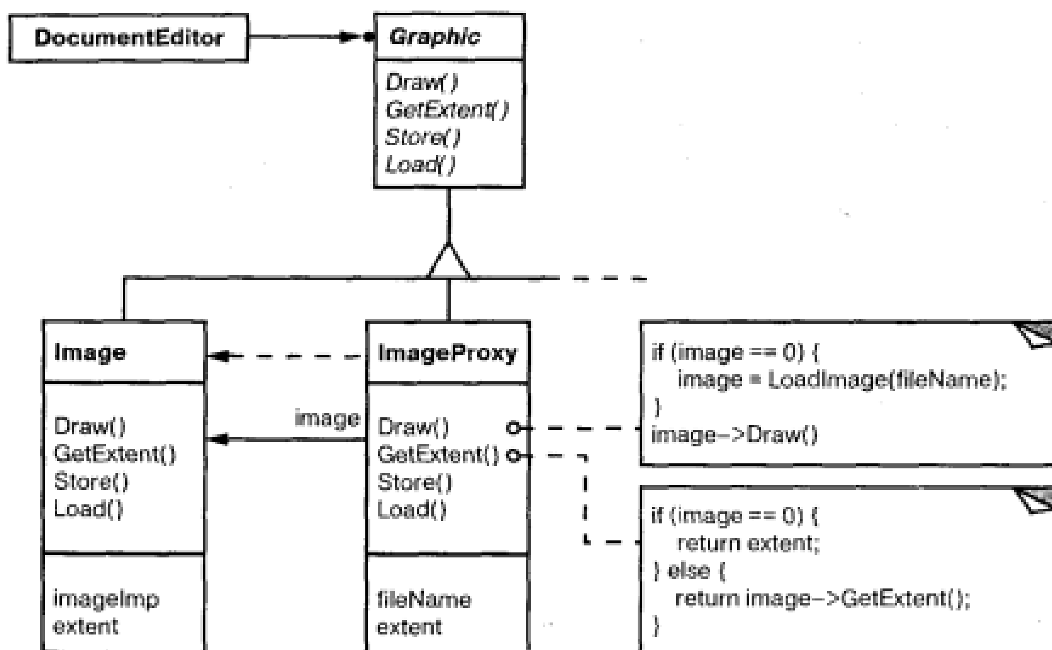

Рис. 3.13. Приклад взаємодії класів в патерні Proxy

Заступник створює справжнє зображення, лише якщо редактор документа викличе операцію Draw. Усі наступні запити заступник переадресує безпосередньо зображенню. Тому після створення зображення він має зберегти посилання на нього.

Припустимо, що зображення зберігаються у окремих файлах. У такому разі

ми можемо використовувати ім'я файлу як посилання реального об'єкта. Заступник зберігає також розмір зображення, тобто довжину та ширину. "Знаючи" її, заступник може відповідати на запити форматера про свій розмір, не інстанцюючи зображення.

Редактор документів отримує доступ до вбудованих зображень лише через інтерфейс, визначений абстрактним класом Graphic. ImageProxy - це клас для представлення зображень, створюваних на вимогу. У ImageProxy зберігається ім'я файлу, що відіграє роль посилання зображення, що знаходиться на диску. Ім'я файлу передається конструктору класу ImageProxy.

В об'єкті ImageProxy знаходяться також прямокутник зображення, що обмежує, і посилання на екземпляр реального об'єкта Image. Посилання залишається недійсним, доки заступник не інстанцує реальне зображення. Операцією Draw гарантується, що зображення буде створено до того, як заступник переадресує йому запит. Операція GetExtent переадресує запит зображенню, тільки якщо воно вже інстанційоване; в іншому випадку ImageProxy повертає розміри, які зберігає сам.

Застосування.

Паттерн заступник застосовується у всіх випадках, коли виникає необхідність послатися на об'єкт більш витончено, ніж це можливо, якщо використовувати просту вказівку. Ось кілька типових ситуацій, де заступник виявляється корисним:

- віддалений заступник надає локального представника замість об'єкта, що знаходиться в іншому адресному просторі;
- віртуальний заступник створює «важкі» об'єкти на вимогу;
- заступник, що захищає, контролює доступ до вихідного об'єкта. Такі заступники є корисними, коли для різних об'єктів визначено різні права доступу;
- «розумне» посилання – це заміна звичайної вказівки. Вона дозволяє виконувати додаткові дії під час доступу до об'єкта. До типових застосувань такого посилання можна віднести:
  - ✓ підрахунок числа посилань на реальний об'єкт, щоб займану їм пам'ять можна було звільнити автоматично, коли залишиться жодного посилання (такі посилання називають ще «розумними» вказівками);
  - ✓ завантаження об'єкта у пам’ять при першому зверненні до нього;
  - ✓ перевірку та встановлення блокування на реальний об'єкт при зверненні до нього, щоб жодний інший об'єкт не зміг у цей час змінити його.

Учасники:

- **Proxy** (ImageProxy) - заступник: зберігає посилання, яке дозволяє заступнику звернутися до реального суб'єкта. Об'єкт класу Proxy може звертатися до об'єкта класу Subject, якщо інтерфейси класів RealSubject та Subject однакові; надає інтерфейс, ідентичний інтерфейсу Subject, тому заступник завжди може бути підставлений замість реального суб'єкта; контролює доступ до реального суб'єкта і може відповідати за його створення та видалення;

Інші обов'язки залежать від виду заступника:

- ✓ віддалений заступник відповідає за кодування запиту та його аргументів та відправлення закодованого запиту реальному суб'єкту в іншому адресному просторі;
- ✓ віртуальний заступник може кешувати додаткову інформацію про реального суб'єкта, щоб відкласти його створення.
- ✓ захищаючий заступник перевіряє, чи має об'єкт, що викликає, необхідні для виконання запиту права;

- **Subject** (Graphic) - суб'єкт: визначає загальний для RealSubject та Proxy інтерфейс, так що клас Proxy можна використовувати скрізь, де очікується RealSubject;
- **RealSubject** (Image) - реальний суб'єкт: визначає реальний об'єкт, представлений заступником.

Структура.

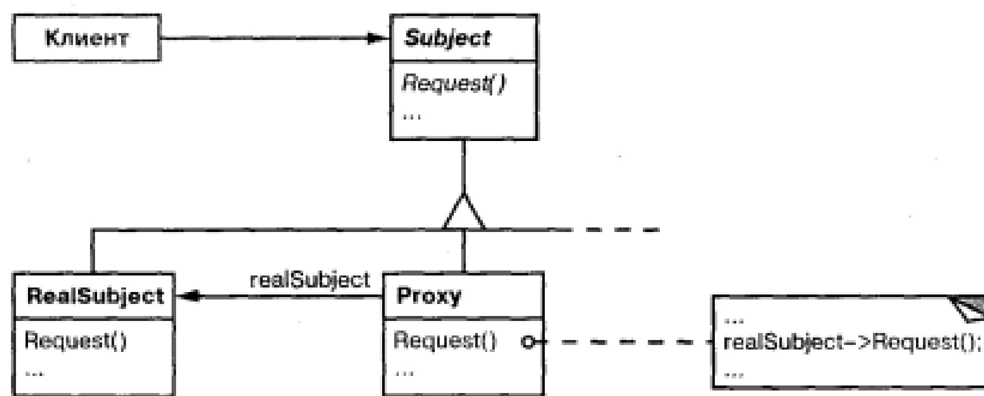

Рис. 3.14. Структура патерну Proxy

Відносини.

Proxy за потреби переадресує запити об'єкту RealSubject. Деталі залежить від виду заступника.

Результати.

За допомогою патерна замісник при доступі до об'єкта вводиться додатковий рівень опосередкованості. Цей підхід має багато варіантів залежно від виду заступника:

- віддалений заступник може приховати той факт, що об'єкт знаходиться в іншому адресному просторі;
- віртуальний заступник може виконувати оптимізацію, наприклад створення об'єкта на вимогу;
- захищаючий заступник та «розумне» посилання дозволяють вирішувати додаткові завдання при доступі до об'єкта.

Є ще одна оптимізація, яку патерн заступник іноді приховує від клієнта. Вона називається копіюванням при записі і має багато спільного зі створенням об'єкта на вимогу. Копіювання великого та складного об'єкта – дуже дорога операція. Якщо копія не модифікувалася, то немає сенсу платити цю ціну. Якщо відкласти процес копіювання, застосувавши заступник, то можна бути впевненим, що ця операція відбудеться лише тоді, коли він справді був змінений.

Щоб під час запису можна було скопіювати, необхідно підраховувати посилання на суб'єкт. Копіювання заступника просто збільшує лічильник посилань. І тільки тоді, коли клієнт запитує операцію, що змінює суб'єкт, заступник справді виконує копіювання. Одночасно заступник має зменшити лічильник посилань. Коли лічильник посилань стає рівним нулю, суб'єкт знищується.

Копіювання під час запису може значно зменшити плату за копіювання «важких» суб'єктів.

Реалізація. При реалізації можна використовувати наступні можливості:

- перевантаження оператора доступу до членів ->. Це дозволяє робити додаткові дії за будь-якого розіменування вказівки на об'єкт. Для деяких видів заступників це виявляється корисно, оскільки заступник поводиться як вказівка;
- заступнику не завжди має бути відомий тип реального об'єкта. Якщо клас Proxy може працювати з суб'єктом тільки через його абстрактний інтерфейс, то не потрібно створювати Proxy для кожного класу реального суб'єкта RealSubject; заступник може звертатися до будь-якого з них однаково. Але якщо заступник має інстанціювати реальних суб'єктів (як у випадку віртуальних заступників), то знання конкретного класу обов'язкове.

```
#include <iostream>
#include <windows.h>

class Graphic {
public:
  virtual ~Graphic() { }
  virtual void Draw(const POINT & p) = 0;
  virtual SIZE GetSize() = 0;
  virtual void Load() = 0;
};

class BitmapImage : public Graphic {
  std::string file;
public:
  BitmapImage(std::string f) { file = f; }
  virtual ~BitmapImage() { }
  virtual void Draw(const POINT & p) {
          std::cout << "Draw bmp Image: " << file.c_str() << std::endl;
  }

  virtual SIZE GetSize() {
          SIZE s = { 100,100 }; // for example
          return s;
  }
  virtual void Load() {
          std::cout << "Load bmp Image: " << file.c_str() << std::endl;
  }
```

```
};

class PNGImage : public Graphic {
  std::string file;
public:
  PNGImage(std::string f) { file = f; }
  virtual ~PNGImage() { }
  virtual void Draw(const POINT & p) {
        std::cout << "Draw Png Image: " << file.c_str() << std::endl;
  }

  virtual SIZE GetSize() {
        SIZE s = {100,100}; // for example
        return s;
  }
  virtual void Load() {
        std::cout << "Load png Image: " << file.c_str() << std::endl;
  }
};


class ImageProxy {
  Graphic * _image;
  public:
        ImageProxy(std::string f) {
              size_t pos = f.find_last_of(".");
              if (pos < 0) throw "Error extension";
              std::string ext = f.substr(pos + 1);
              if (ext == "png") {
                    _image = new PNGImage(f);
              }
              else if (ext == "bmp") {
                    _image = new BitmapImage(f);
              }
              else {
                    throw "Error extension";
              }
        }

        ~ImageProxy() { delete _image; }

        Graphic * operator ->() {
              return _image;
        }
};
```

```
int main()
{
  ImageProxy proxy("1.png");

  proxy->Load();
  POINT p = { 10,10 };
  proxy->Draw(p);
  return 0;
}
```

Контрольні питання

1. Які переваги та недоліки патерну Adapter?
2. Які переваги та недоліки патерну Bridge?
3. Які переваги та недоліки патерну Composite?
4. Які переваги та недоліки патерну Decorator?
5. Які переваги та недоліки патерну Facade?
6. Які переваги та недоліки патерну Flyweight?
7. Які переваги та недоліки патерну Proxy?

РОЗДІЛ 4
ПАТЕРНИ ПОВЕДІНКИ

4.1. Характеристика патернів поведінки.

Паттерни поведінки пов'язані з алгоритмами та розподілом обов'язків між об'єктами. Річ в них йде не лише про самі об'єкти та класи, а й про типові способи взаємодії. Паттерни поведінки характеризують помилковий потік управління, який важко простежити під час виконання програми. Увага акцентована не на потоці управління як такому, а на зв'язках між об'єктами.

У патернах поведінки рівня класу використовується спадкування - щоб розподілити поведінку між різними класами. З них більш простим і поширеним є шаблонний метод, який являє собою абстрактне визначення алгоритму. Алгоритм тут визначається покроково. На кожному кроці викликається або примітивна або абстрактна операція. Алгоритм «обростає м'ясом» за рахунок підкласів, де визначено абстрактні операції. Інший патерн поведінки рівня класу – інтерпретатор, який представляє граматику мови у вигляді ієрархії класів та реалізує інтерпретатор як послідовність операцій над екземплярами цих класів.

У патернах поведінки рівня об'єктів використовується не наслідування, а композиція. Деякі з них описують, як за допомогою кооперації безліч рівноправних об'єктів справляється із завданням, яке жодному з них не під силу. Важливо тут те, як об'єкти одержують інформацію про існування один одного. Об'єкти-колеги можуть зберігати посилання один на одного, але це збільшить ступінь зв'язаності системи. При максимальному ступені пов'язаності кожному об'єкту доведеться мати інформацію про всіх інших. Цю проблему вирішує посередник. Посередник, що між об'єктами-колегами, забезпечує опосередкованість посилань, необхідну для розривання зайвих зв'язків.

Паттерн ланцюжок обов'язків дозволяє й надалі зменшувати ступінь зв'язаності. Він дає можливість надсилати запити об'єкту не безпосередньо, а ланцюжком «об'єктів-кандидатів». Запит може виконати будь-який кандидат, якщо це допустимо в поточному стані виконання програми. Кількість кандидатів наперед не визначена, а підбирати учасників можна під час виконання.

Паттерн спостерігач визначає та відповідає за залежності між об'єктами.

Інші патерни поведінки пов'язані з інкапсуляцією поведінки в об'єкті та делегування йому запитів. Паттерн стратегія інкапсулює алгоритм об'єкта, спрощуючи його специфікацію та заміну. Паттерн команда інкапсулює запит у вигляді об'єкта, який можна передавати як параметр, зберігати у списку історії або використати якось інакше. Паттерн стан інкапсулює стан об'єкта таким чином, що при зміні стану об'єкт може змінювати поведінку. Паттерн відвідувач інкапсулює поведінку, яку інакше довелося б розподіляти між класами, а патерн ітератор абстрагує спосіб доступу та обходу об'єктів з деякого агрегату.

4.2. Патерн Chain of Responsibility.

Призначення. Дозволяє уникнути прив'язки відправника запиту до одержувача, даючи шанс обробити запит кільком об'єктам. Зв'язує об'єкти-одержувачі в ланцюжок і передає запит уздовж цього ланцюжка, доки його не обробулять.

Мотивація.

Розглянемо контекстно-залежну оперативну довідку в графічному інтерфейсі користувача, який може отримати додаткову інформацію з будь-якої частини інтерфейсу, просто клацнувши мишею. Зміст довідки залежить від того, яка частина інтерфейсу та в якому контексті вибрана. Наприклад, довідка по кнопці в діалоговому вікні може відрізнятися від довідки за аналогічною кнопкою в головному вікні програми. Якщо для деякої частини інтерфейсу довідки немає, система повинна показати інформацію про найближчий контекст, в якому вона знаходиться, наприклад про діалогове вікно в цілому.

Тому природно було б організувати довідкову інформацію від конкретніших розділів до загальніших. Крім того, ясно, що запит на отримання довідки обробляється одним з декількох об'єктів інтерфейсу користувача, яким саме - залежить від контексту і наявної інформації.

Проблема в тому, що об'єкт, який ініціює запит (наприклад, кнопка), не має інформації про те, який об'єкт зрештою надасть довідку. Нам потрібен якийсь спосіб відокремити кнопку-ініціатор запиту від об'єктів, які мають довідкову інформацію. Як цього досягти, показує патерн ланцюжок обов'язків.

Ідея полягає в тому, щоб розірвати зв'язок між відправниками та одержувачами, давши можливість обробити запит кільком об'єктам. Запит переміщається ланцюжком об'єктів, поки один з них не обробить його.

Перший об'єкт у ланцюжку отримує запит і або обробляє його сам, або направляє наступному кандидату в ланцюжку, який поводиться так само. Об'єкт, що надіслав запит, не має інформації про обробника. Ми говоримо, що у запита є анонімний одержувач.

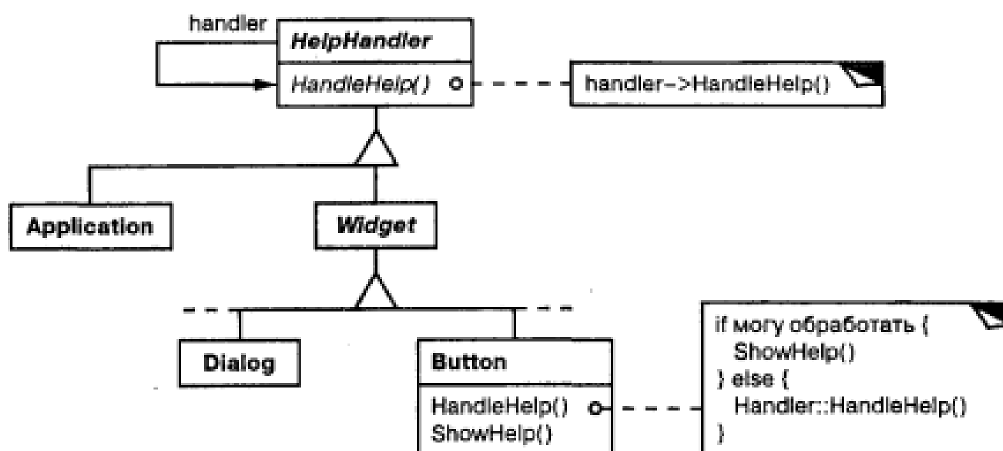

Рис. 4.1. Приклад взаємодії між класами в патерні Chain of Responsibility

Припустимо, що користувач запитує довідку на кнопці Print (друк). Вона знаходиться у діалоговому вікні PrintDialog, що містить інформацію про об'єкт програми, якому належить.

Щоб надіслати запит по ланцюжку і гарантувати анонімність одержувача, всі об'єкти в ланцюжку мають єдиний інтерфейс для обробки запитів і доступу до свого наступника (наступного об'єкта в ланцюжку). Наприклад, у системі оперативної довідки можна було б визначити клас HelpHandler (предок класів усіх об'єктів-кандидатів або клас, що підмішується (mixin class)) з операцією HandleHelp. Тоді класи, які опрацьовуватимуть запит, зможуть його передати своєму батькові.

Для обробки запитів на отримання довідки класи Button, Dialog та Application користуються операціями HelpHandler. За умовчанням операція HandleHelp просто перенаправляє запит своєму наступнику. У підкласах ця операція заміщується, тож за сприятливих обставин може видаватися довідкова інформація. Інакше запит надсилається далі за допомогою за умовчанням.

Застосування. Використовуйте ланцюжок обов'язків, коли:

- є більше одного об'єкта, здатного обробити запит, причому цей обробник заздалегідь невідомий і повинен бути знайдений автоматично;
- ви хочете надіслати запит одному з кількох об'єктів, не вказуючи явно якому саме;
- набір об'єктів, здатних обробити запит, має задаватися динамічно.

Структура.

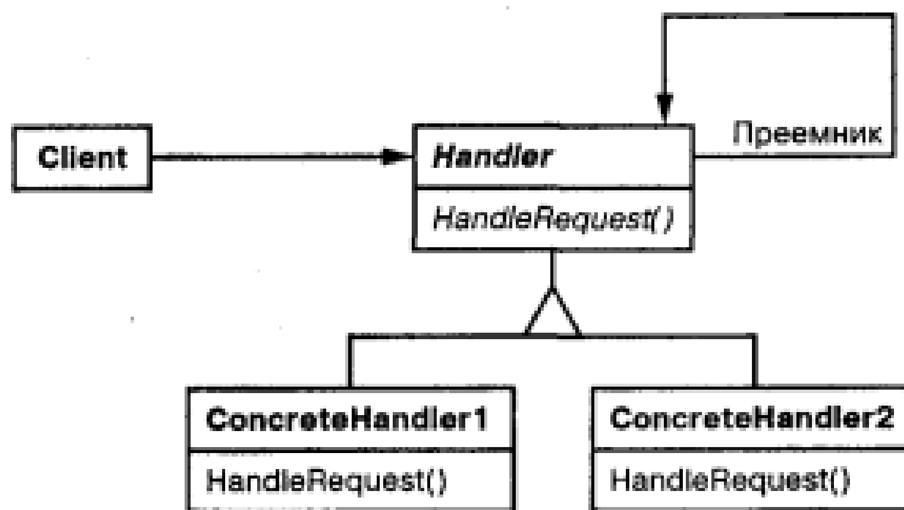

Рис. 4.2. Структура патерну Chain of Responsibility

Учасники:

- **Handler** (HelpHandler) - обробник: визначає інтерфейс для обробки запитів; реалізує зв'язок із наступником (необов'язково);
- **ConcreteHandler** (PrintButton, PrintDialog) - конкретний обробник: опрацьовує запит, за який відповідає; має доступ до свого наступника; якщо ConcreteHandler здатний обробити запит, то так і робить, якщо не може, то

спрямовує його – його своєму наступнику;

- **Client** - клієнт: надсилає запит деякому об'єкту ConcreteHandler у ланцюжку.

Відносини. Коли клієнт ініціює запит, він просувається ланцюжком, поки деякий об'єкт ConcreteHandler не візьме на себе відповідальність за його обробку.

Результати. Паттерн ланцюжок обов'язків має такі переваги та недоліки:

- ослаблення зв'язаності. Цей патерн звільняє об'єкт від необхідності «знати», хто опрацює його запит. Відправнику та одержувачу нічого невідомо один про одного, а включеному в ланцюжок об'єкту - про структуру ланцюжка. Таким чином, ланцюжок обов'язків допомагає спростити взаємозв'язки між об'єктами. Замість того, щоб зберігати посилання на всі об'єкти, які можуть стати одержувачами запиту, об'єкт повинен мати інформацію лише про свого найближчого наступника;
- додаткова гнучкість під час розподілу обов'язків між об'єктами. Ланцюжок обов'язків дозволяє підвищити гнучкість розподілу обов'язків між об'єктами. Додати або змінити обов'язки по обробці запиту можна, включивши в ланцюжок нових учасників або змінивши його іншим чином. Цей підхід можна поєднувати зі статичним породженням підкласів до створення спеціалізованих обробників;
- отримання не гарантоване. Оскільки запит не має явного одержувача, то немає і гарантій, що він взагалі буде оброблений: він може досягти кінця ланцюжка. Необробленим запит може бути і у разі неправильної конфігурації ланцюжка.

Реалізація. При реалізації патерну ланцюжок обов’язків слід звернути увагу на наступне:

- реалізація ланцюжка наступників. Є два способи реалізувати такий ланцюжок: визначити нові зв'язки (зазвичай це робиться в класі Handler, але можна і в ConcreteHandler); використовувати існуючі зв'язки.

Досі в наших прикладах визначалися нові зв'язки, проте можна скористатися посиланнями на об'єкти для формування ланцюжка наступників. Наприклад, посилання на батька в ієрархії "частина-ціле" може заодно визначати і наступника "частини". У структурі віджетів такі зв'язки також можуть існувати. Існуючі зв'язки можна використовувати, коли вони вже підтримують потрібний ланцюжок. Тоді ми уникнемо явного визначення нових зв'язків та заощадимо пам'ять. Але якщо структура не відображає пристрої ланцюжка обов'язків, то уникнути визначення надлишкових зв'язків не вдасться;

- з'єднання наступників. Якщо готових посилань, придатних для визначення ланцюжка, немає, їх доведеться запровадити. У такому випадку клас Handler не тільки визначає інтерфейс запитів, але й зберігає посилання на наступника. Отже у оброблювача з'являється можливість визначити реалізацію операції HandleRequest за умовчанням - перенаправлення запиту наступнику (якщо такий існує). Якщо підклас ConcreteHandler не зацікавлений у запиті, то йому і не треба замінювати цю операцію, оскільки за замовчуванням запит якраз і надсилається далі.
- подання запитів. Подавати запити можна по-різному. У найпростішій

формі, наприклад, у випадку HandleHelp, запит жорстко кодується як виклик деякої операції. Це зручно та безпечно, але переадресовувати тоді можна лише фіксований набір запитів, визначених у класі Handler.

Альтернатива – використовувати одну функцію-обробник, якою передається код запиту (скажімо, ціле число або рядок). Так можна підтримати заздалегідь невідому кількість запитів. Єдина вимога полягає в тому, що відправник та одержувач повинні домовитися про спосіб кодування запиту.

Це більш гнучкий підхід, але при реалізації потрібно використовувати умовні оператори для роздачі запитів на їх код. Крім того, не існує безпечного з погляду типів способу передачі параметрів, тому упаковувати та розпаковувати їх доводиться вручну. Очевидно, що це не так безпечно, як прямий виклик операції.

Щоб вирішити проблему передачі параметрів, можна використовувати окремі об'єкти-запити, в яких інкапсульовані параметри запиту. Клас Request може представляти деякі запити явно, а їх нові типи описуються в підкласах. Підклас може визначити інші параметри. Обробник повинен мати інформацію про тип запиту (який саме підклас Request використовується), щоб розібрати ці параметри.

Для ідентифікації запиту у класі Request можна визначити функцію доступу, яка повертає ідентифікатор класу. Натомість одержувач міг би скористатися інформацією про тип, доступною під час виконання, якщо мова програмування підтримує таку можливість.

```
#include <iostream>
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
  HelpHandler(HelpHandler *h = 0, Topic t = NO_HELP_TOPIC): _successor(h),
_topic(t){ }
  virtual bool HasHelp() { return _topic != NO_HELP_TOPIC; }
  virtual void SetHandler(HelpHandler *h, Topic t) { _successor = h; _topic = t;}
  virtual void HandleHelp() { if (_successor) _successor->HandleHelp();}
private:
  HelpHandler *_successor;
  Topic _topic;
};



class Widget: public HelpHandler {
      Widget *_parent;
  public:
      Widget(Widget *p = 0, Topic t = NO_HELP_TOPIC) : HelpHandler(p, t)
{ _parent = p; }
```

```
};

class Button : public Widget {
public:
  Button(Widget *p = 0, Topic t = NO_HELP_TOPIC) : Widget(p, t) {  }
  void HandleHelp() {
        if (HasHelp()) {
              std::cout << "Button Handle Help" << std::endl;
        }
        else {
              HelpHandler::HandleHelp();
        }
  }
};

class Dialog : public Widget {
public:
  Dialog(HelpHandler *p = 0, Topic t = NO_HELP_TOPIC) : Widget(0) {
        SetHandler(p, t);
  }
  void HandleHelp() {
        if (HasHelp()) {
              std::cout << "Dialog Handle Help" << std::endl;
        }
        else {
              HelpHandler::HandleHelp();
        }
  }
};


class Application : public HelpHandler {

public:
  Application(Topic t) : HelpHandler(0, t) {  }
  void HandleHelp() {
        std::cout << "Help" << std::endl;
  }
};

const Topic APPLICATION_TOPIC = 1;
const Topic PRINT_TOPIC = 2;
const Topic PAPER_ORIENTATION_TOPIC = 3;

int main()
```

```
{
 Application * app = new Application(APPLICATION_TOPIC);
 Dialog * dialog = new Dialog(app, PRINT_TOPIC);
 Button * button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
 button->HandleHelp();
 delete button;
 delete dialog;
 delete app;
 return 0;
}
```

4.3. Патерн Command.

Призначення. Інкапсулює запит як об'єкт, дозволяючи цим задавати параметри клієнтів для обробки відповідних запитів, ставити запити в чергу або протоколювати їх, а також підтримувати скасування операцій.

Мотивація.

Іноді необхідно надсилати об'єктам запити, нічого не знаючи про те, виконання якої операції запрошено і хто є одержувачем. Наприклад, у бібліотеках для побудови інтерфейсів користувача зустрічаються такі об'єкти, як кнопки і меню, які посилають запит у відповідь на дію користувача. Але в саму бібліотеку не закладено можливість обробляти цей запит, оскільки тільки додаток, що використовує її, має інформацію про те, що слід зробити. Проектувальник бібліотеки не має жодної інформації про одержувача запиту та про те, які операції той має виконати.

Паттерн команда дозволяє бібліотечним об'єктам надсилати запити невідомим об'єктам програми, перетворивши сам запит на об'єкт. Цей об'єкт можна зберігати та передавати, як і будь-який інший. В основі патерна, що описується, лежить абстрактний клас Command в якому оголошено інтерфейс для виконання операцій. У найпростішій формі цей інтерфейс складається з однієї абстрактної операції Execute. Конкретні підкласи Command. визначають пару «одержувач-дія», зберігаючи одержувача у змінній екземпляру, і реалізують операцію Execute, так щоб вона надсилала запит. Одержувач має інформацію, необхідну для виконання запиту.

За допомогою об'єктів Command легко реалізуються меню. Кожен пункт меню – це екземпляр класу MenuItem. Самі меню і всі їх пункти створює клас Application поряд з усіма іншими елементами інтерфейсу користувача. Клас Application відстежує також відкриті користувачем документи.

Додаток конфігурує кожен об'єкт MenuItem екземпляром конкретного підкласу Command. Коли користувач вибирає певний пункт меню, асоційований з ним об'єкт MenuItem викликає Execute для свого об'єкта-команди, а Execute виконує операцію. Об'єкти MenuItem не мають інформації, який підклас класу Command вони використовують. Підкласи Command зберігають інформацію про одержувача запиту та викликають одну або кілька операцій цього одержувача.

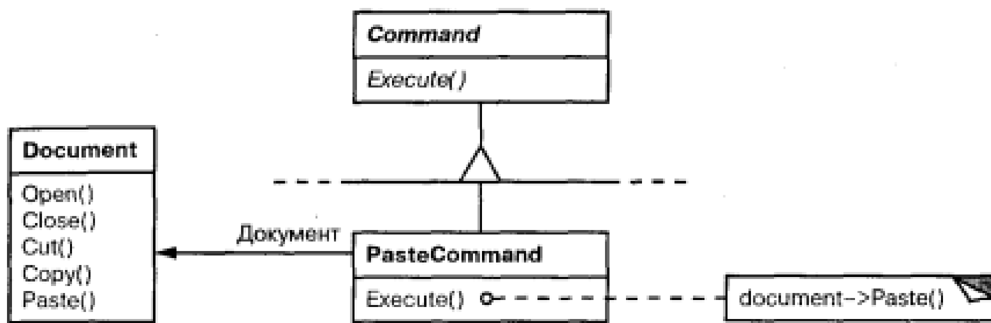

Рис. 4.3. Приклад взаємодії класів в патерні Command

Наприклад, підклас PasteCommand підтримує вставку тексту з буфера обміну в документ. Одержувачем для PasteCommand є Document, який був переданий при створенні об'єкта. Операція Execute викликає операцію Paste документа-отримувача.

Для підкласу OpenCommand операція Execute поводиться по-іншому: вона запитує у користувача ім'я документа, створює відповідний об'єкт Document, повідомляє про новий документ додаток-одержувач і відкриває цей документ.

Іноді об'єкт MenuItem повинен виконувати послідовність команд. Наприклад, пункт меню для центрування сторінки стандартного розміру можна було б сконструювати відразу з двох об'єктів: CenterDocumentCommand та NormalSizeCommand. Оскільки таке комбінування команд – явище звичайне, ми можемо визначити клас MacroComand, що дозволяє об'єкту MenuItem виконувати довільне число команд. MacroCommand - це конкретний підклас класу Command, який просто виконує послідовність команд. Він не має явного одержувача, оскільки для кожної команди визначено свій власний.

Зверніть увагу, що в кожному з наведених прикладів патерн команда відокремлює об'єкт, який ініціює операцію, від об'єкта, який знає, як її виконати. Це дозволяє досягти високої гнучкості при проектуванні інтерфейсу користувача. Пункт меню і кнопка одночасно можуть бути асоційовані в додатку з деякою функцією, для цього достатньо приписати обом елементам один і той самий екземпляр конкретного підкласу класу Command. Ми можемо динамічно підміняти команди, що дуже корисно для реалізації контекстно-залежних меню. Можна також підтримати сценарії, якщо компонувати прості команди у складніші. Все це можна здійснити тому, що об'єкт, який ініціює запит, повинен мати інформацію лише про те, як його відправити, а не про те, як його виконати.

Застосування. Використовуйте патерн команда, коли хочете:

- налаштувати об'єкти виконуваною дією, як у випадку з пунктами меню MenuItem;
- визначати, ставити в чергу та виконувати запити у різний час. Час життя об'єкта Command необов'язково має залежати від часу вихідного запиту. Якщо одержувача запиту вдається реалізувати так, щоб він не залежав від адресного простору, об'єкт-команду можна передати іншому процесу, який займеться його виконанням;
- підтримати скасування операцій. Операція Execute об'єкту Command

може зберегти стан, необхідний для відкату дій, виконаних командою. В цьому випадку в інтерфейсі класу Execute має бути додаткова операція Unexecute, яка скасовує дії, виконані попереднім зверненням до Execute. Виконані команди зберігаються у списку історії. Для реалізації довільного числа рівнів скасування та повтору команд потрібно обходити цей список відповідно у зворотному та прямому напрямках, викликаючи при відвідуванні кожного елемента команду Unexecute або Execute;

- підтримати протоколювання змін, щоб їх можна було виконати повторно після аварійної зупинки системи. Доповнивши інтерфейс класу Command операціями збереження та завантаження, ви зможете вести протокол змін у зовнішній пам'яті. Для відновлення після збою потрібно буде завантажити збережені команди з диска та повторно виконати їх за допомогою операції Execute;
- структурувати систему на основі високорівневих операцій, побудованих із примітивних. Така структура є типовою для інформаційних систем, що підтримують транзакції. Транзакція інкапсулює набір змін даних. Паттерн команда дозволяє моделювати транзакції. Всі команди мають спільний інтерфейс, що дає можливість працювати однаково з будь-якими транзакціями. За допомогою цього патерну можна легко додавати до системи нові види транзакцій.

Структура.

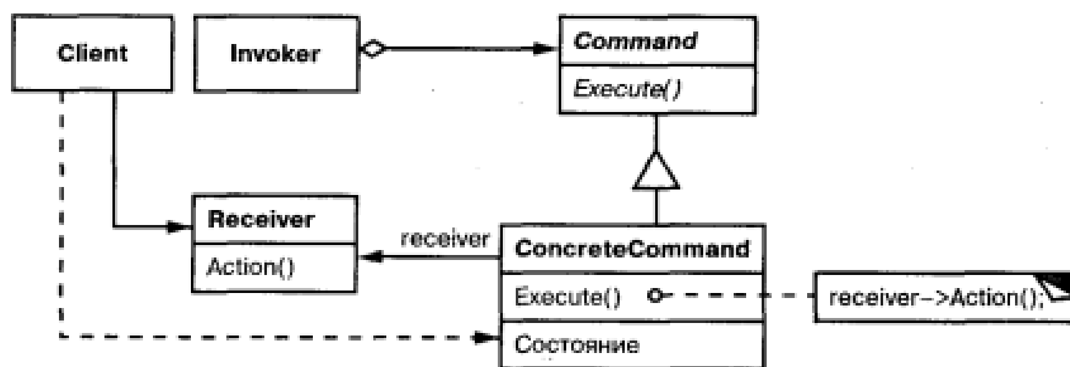

Рис. 4.4. Структура патерну Command

Учасники:

- **Command** - команда: оголошує інтерфейс для виконання операції;
- **ConcreteCommand** (PasteCommand, OpenCommand) – конкретна команда: визначає зв'язок між об'єктом-одержувачем Receiver та дією; реалізує операцію Execute шляхом виклику відповідних операцій об'єкта Receiver;
- **Client** - клієнт: створює об'єкт класу ConcreteCommand та встановлює його одержувача;
- **Invoker** (MenuItem) - ініціатор: звертається до команди для виконання запиту;
- **Receiver** (Document, Application) - одержувач: має у своєму розпорядженні інформацію про способи виконання операцій, необхідні для задоволення запиту. У ролі одержувача може бути будь-який клас.

Відносини:

- клієнт створює об'єкт ConcreteCommand та встановлює для нього одержувача;
- ініціатор Invoker зберігає об'єкт ConcreteCommand;
- ініціатор відправляє запит, викликаючи операцію команди Execute. Якщо підтримується скасування виконаних дій, то ConcreteCommand перед викликом Execute зберігає інформацію про стан, достатній для виконання відкату;
- об'єкт ConcreteCommand викликає операції одержувача до виконання запиту.

Результати. Результати застосування патерну команда такі:

- команда розриває зв'язок між об'єктом, який ініціює операцію, та об'єктом, який має інформацію про те, як її виконати;
- команди - це справжнісінькі об'єкти. Допускається маніпулювати ними і розширювати їх так само, як у випадку з будь-якими іншими об'єктами;
- з простих команд можна збирати складові, наприклад, клас MacroCommand, розглянутий вище. У загальному випадку складові команди описуються патерном компоновщик;
- додавати нові команди легко, оскільки жодні існуючі класи змінювати не потрібно.

Реалізація. При реалізації патерну команда слід звернути увагу на такі аспекти:

- наскільки «розумною» має бути команда. У команди може бути широке коло обов'язків. На одному полюсі стоїть просте визначення зв'язку між одержувачем та діями, які потрібно виконати для задоволення запиту. На іншому - реалізація всього самостійно, без звернення по допомогу до одержувача. Останній варіант корисний, коли ви хочете визначити команди, які не залежать від існуючих класів, коли відповідного одержувача немає або коли одержувач команді точно не відомий. Наприклад, команда, що створює нове вікно програми, може не розуміти, що саме вона створює, а трактувати вікно як будь-який інший об'єкт. Десь посередині між двома крайнощами знаходяться команди, які мають достатню інформацію для динамічного виявлення свого одержувача;
- підтримка скасування та повторення операцій. Команди можуть підтримувати скасування та повторення операцій, якщо є можливість скасувати результати виконання (наприклад, операцію Unexecute або Undo). У класі ConcreteCommand може зберігатися необхідна для цього додаткова інформація, зокрема:
  - ✓ об'єкт-одержувач Receiver, який виконує операції у відповідь на запит;
  - ✓ аргументи операції, виконаної одержувачем;
  - ✓ вихідні значення різних атрибутів одержувача, які можуть бути змінені в результаті обробки запиту. Одержувач повинен надати операції, що дозволяють команді повернутися у початковий стан.

Для підтримки всього одного рівня скасування додатку достатньо зберігати

лише останню виконану команду. Якщо ж потрібні багаторівневі скасування та повторення операцій, то доведеться вести список історії виконаних команд. Максимальна довжина цього списку визначає кількість рівнів скасування і повтору. Прохід за списком у зворотному напрямку і відкат результатів всіх команд, що зустрілися по дорозі, скасовує їхню дію; прохід у прямому напрямку і виконання команд, що зустрілися, призводить до повтору дій.

Команду, яка допускає скасування, можливо, доведеться скопіювати перед поміщенням до списку історії. Справа в тому, що об'єкт команди, використаний для доставки запиту, скажімо від пункту меню MenuItem, пізніше міг бути використаний для інших запитів. Тому копіювання необхідно визначити різні виклики однієї і тієї ж команди, якщо її стан при будь-якому виклику може змінюватися.

Наприклад, команда DeleteCommand, яка видаляє вибрані об'єкти, при кожному виклику повинна зберігати різні набори об'єктів. Тому об'єкт DeleteCommand необхідно скопіювати після виконання, а копію помістити до списку історії. Якщо в результаті виконання стан команди ніколи не змінюється, то копіювати не потрібно – до списку достатньо помістити лише посилання на команду. Команди, які обов'язково потрібно копіювати перед розміщенням до списку історії, поводяться подібно до прототипів;

- як уникнути накопичення помилок у процесі скасування. При виконанні, скасуванні та повторі команд іноді накопичуються помилки, внаслідок чого стан програми виявляється відмінним від початкового. Тому часом необхідно зберігати в команді більше інформації, щоб гарантувати, що об'єкти будуть повністю відновлені. Щоб надати команді доступ до цієї інформації, не розкриваючи внутрішнього пристрою об'єктів, можна скористатися патерном зберігач;

```
#include <iostream>
#include <vector>

class Command {
public:
  virtual ~Command() {}
  virtual void Execute() = 0;
protected:
  Command() {}
};

class Document {
public:

  void Paste() {
        std::cout << "Paste" << std::endl;
  }
};
```

```
class Application {
public:
  void Open() {
          std::cout << "Open doc" << std::endl;
  }

};

class OpenCommand: public Command {
  Application * _receiver;
public:
  OpenCommand(Application * d): _receiver(d) {}
  virtual void Execute() { _receiver->Open(); }
};

class PasteCommand : public Command {
  Document * _receiver;
public:
  PasteCommand(Document * d) : _receiver(d) {}
  virtual void Execute() { _receiver->Paste(); }
};

class MenuItem {
  Command *_command;
  std::string _title;
public:
  MenuItem(Command *command, std::string title):_command(command),
_title(title){}
  ~MenuItem() { delete _command; }
  void Execute() {
          _command->Execute();
  }
  std::string get() { return _title; }
};

class Menu {
  std::vector<MenuItem *> items; // 4. "container" coupled to the interface
public:
  ~Menu() {
          for (int i = 0; i < items.size(); i++) {
                  delete items[i];
          }
  }

  void Add(MenuItem *item) {
```

```
        items.push_back(item);
  }

  void Show() {
        for (int i = 0; i < items.size(); i++) {
              std::cout << i+1 << " " << items[i]->get().c_str() << std::endl;
        }
  }

  void Select(int m) {
        items[m]->Execute();
  }
};

int main()
{
  Document * doc = new Document;
  Application * app = new Application;
  Menu * menu = new Menu();

  menu->Add(new MenuItem(new OpenCommand(app), "Open"));
  menu->Add(new MenuItem(new PasteCommand(doc), "Paste"));

  menu->Show();
  int m;
  std::cin >> m;
  if (m>0 && m<3)
  menu->Select(m - 1);

  delete menu;
  delete doc;
  delete app;
  return 0;
}
```

4.4. Патерн Interpreter.

Призначення. Для заданої мови визначає подання її граматики, а також інтерпретатор речень цієї мови.

Мотивація.

Якщо деяке завдання виникає часто, то є сенс представити її конкретні прояви у вигляді речень простою мовою. Потім можна буде створити інтерпретатор, який вирішує завдання, аналізуючи речення цієї мови.

Наприклад, пошук рядків на зразок - дуже поширене завдання. Регулярні вирази - це стандартна мова для зразків пошуку. Замість програмувати

спеціалізовані алгоритми для зіставлення рядків з кожним зразком, чи не простіше побудувати алгоритм пошуку так, щоб він міг інтерпретувати регулярне вираз, що описує безліч рядків-зразків?

Паттерн інтерпретатор визначає граматику простої мови, подає пропозиції цією мовою та інтерпретує їх. Для наведеного прикладу патерн визначає визначення граматики та інтерпретації мови регулярних виразів.

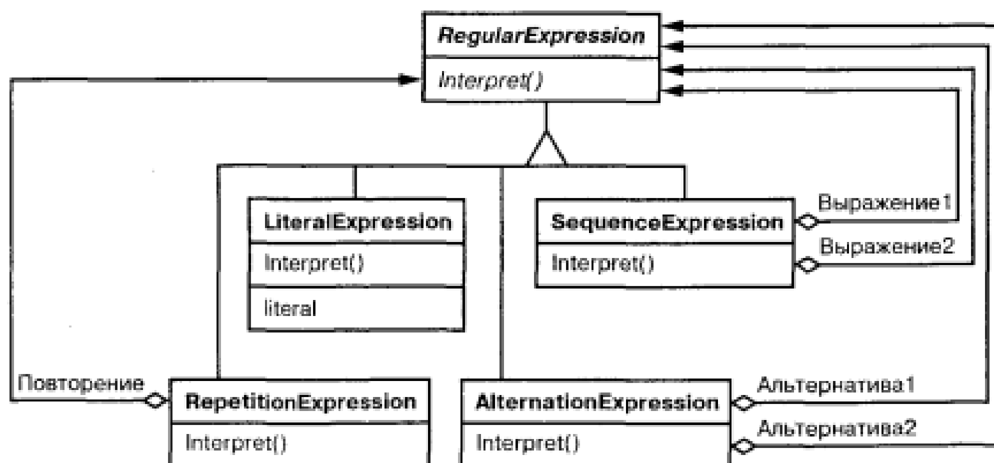

Рис. 4.5. Приклад взаємодії класів в патерні Interpreter

Ми можемо створити інтерпретатор регулярних виразів, визначивши в кожному підкласі LiteralExpression операцію Interpret, яка приймає як аргумент контекст, де потрібно інтерпретувати вираз. Контекст складається з вхідного рядка та інформації про те, як далеко по ньому ми вже просунулися. У кожному підкласі RegularExpression операція Interpret здійснює зіставлення з частиною вхідного рядка, що залишилася. Наприклад:

LiteralExpression перевіряє, чи відповідає вхідний рядок літералу, що зберігається в об'єкті підкласу;

RegularExpression перевіряє, чи відповідає рядок однієї з альтернатив;

RepetitionExpression перевіряє, якщо в рядку повторювані входження виразу, що збігається з тим, що зберігається в об'єкті.

І так далі.

Застосування.

Використовуйте патерн інтерпретатор, коли є мова для інтерпретації, речення якої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще цей патерн працює, коли:

- граматика проста. Для складних граматик ієрархія класів стає надто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо, і час;
- ефективність не є головним критерієм. Найбільш ефективні інтерпретатори зазвичай працюють безпосередньо з деревами, а спочатку транслюють в іншу форму. Так, регулярний вираз часто перетворять на кінцевий

автомат. Але навіть у цьому випадку сам транслятор можна реалізувати за допомогою патерну інтерпретатора.

Структура.

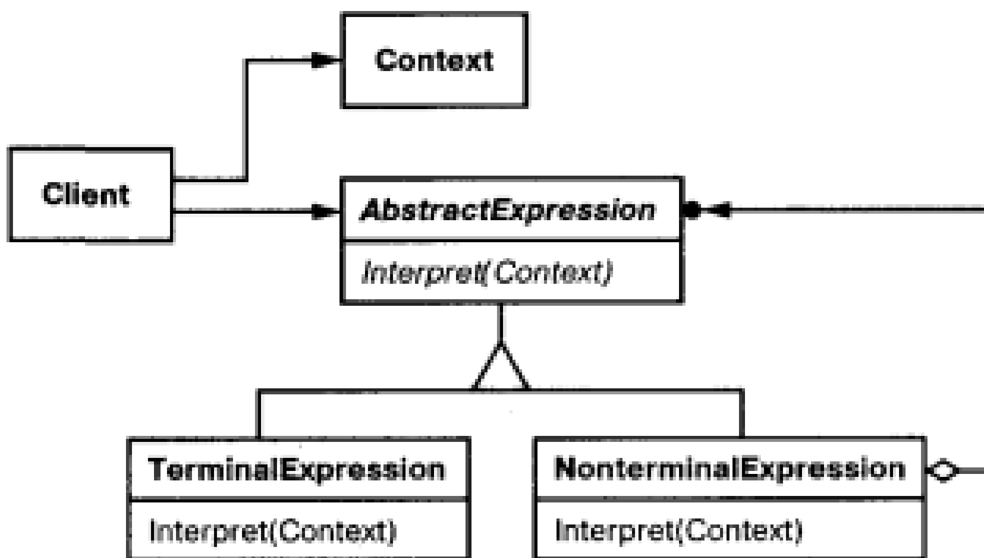

Рис. 4.6. Структура патерну Interpreter

Учасники:

- **AbstractExpression** (RegularExpression) - абстрактний вираз: оголошує абстрактну операцію Interpret, загальну для всіх вузлів в абстрактному синтаксичному дереві;
- **TerminalExpression** (LiteralExpression) - термінальний вираз: реалізує операцію 1пЬегрreР для термінальних символів граматики; необхідний окремий екземпляр для кожного термінального символу у реченні;
- **NonterminalExpression** (AlternationExpression, RepetitionExpression, SequrnceExpression) - нетермінальний вираз: по одному такому класу потрібно для кожного граматичного правила; зберігає змінні екземпляра типу AbstractExpression кожного символу; реалізує операцію Interpret для нетермінальних символів граматики. Ця операція рекурсивно викликає себе для змінних.
- **Context** - контекст: містить інформацію, глобальну щодо інтерпретатора;
- **Client** - клієнт: - будує (або одержує у готовому вигляді) абстрактне синтаксичне дерево, що представляє окрему пропозицію мовою з даною граматикою. Дерево складено з екземплярів класів TerminalExpression та NonterminalExpression; викликає операцію Interpret.

Відносини:

- клієнт будує (або отримує у готовому вигляді) пропозицію у вигляді абстрактного синтаксичного дерева, у вузлах якого знаходяться об'єкти класів TerminalExpression та NonterminalExpression. Потім клієнт ініціалізує контекст і викликає операцію Interpret;
- у кожному вузлі виду NonterminalExpression через операції Interpret

визначається операція Interpret для кожного підвиразу. Для класу TerminalExpression операція Interpret визначає базу рекурсії;

- операції Interpret у кожному вузлі використовують контекст для збереження та доступу до стану інтерпретатора.

Результати. Патерн інтерпретатор має такі переваги і недоліки:

- граматику легко змінювати та розширювати. Оскільки для представлення граматичних правил у патерні використовуються класи, то для зміни чи розширення граматики можна застосовувати успадкування. Існуючі вирази можна модифікувати поступово, а нові визначати як варіації старих;
- проста реалізація граматики. Реалізації класів, що описують вузли абстрактного синтаксичного дерева, схожі. Такі класи легко кодувати, а часто може автоматично згенерувати компілятор або генератор синтаксичних аналізаторів;
- складні граматики важко супроводжувати. У патерні інтерпретатор визначається щонайменше один клас для кожного правила граматики. Тому супровід граматики з великою кількістю правил іноді виявляється важким завданням. Для її вирішення можуть бути застосовані інші патерни, але якщо граматика дуже складна, краще вдатися до інших методів, наприклад, скористатися генератором компіляторів або синтаксичних аналізаторів;
- додавання нових способів інтерпретації виразів. Паттерн інтерпретатор дозволяє легко змінити спосіб обчислення виразів. Наприклад, реалізувати гарний друк виразу замість перевірки типів, що входять до нього, можна, просто визначивши нову операцію в класах виразів. Якщо вам доводиться часто створювати нові способи інтерпретації виразів, подумайте про застосування патерну відвідувач. Це допоможе уникнути зміни класів, що описують граматику.

Реалізація. У реалізацій патернів інтерпретатор та компонувальник є багато спільного. Наступні питання стосуються лише інтерпретатора:

створення абстрактного синтаксичного дерева. Паттерн інтерпретатор не пояснює, як створювати дерево, тобто розбір виразу не входить до його завдання. Створити дерево розбору може таблично-керований чи написаний вручну (зазвичай методом рекурсивного спуску) аналізатор, і навіть клієнт;

визначення операції Interpret. Визначати операцію у класах виразів необов'язково. Якщо створювати нові інтерпретатори доводиться часто, то краще скористатися патерном відвідувач і помістити операцію Interpret в окремий об'єкт-відвідувач;

поділ термінальних символів за допомогою патерна пристосуванець. Для граматик, пропозиції яких містять багато входжень одного і того ж термінального символу, може бути корисним поділ цього символу.

У термінальних вузлах зазвичай не зберігається інформація про положення в абстрактному синтаксичному дереві. Необхідний для інтерпретації контекст надають батьківські вузли. В наявності різниця між поділеним (внутрішнім) і переданим (зовнішнім) станами, так що цілком можна застосувати патерн пристосуванець.

```
#include <iostream>
#include <string>
#include <map>

class Context;
class VariableExp;

class AbstractExp
{
 public:
        virtual bool interpret(Context *context) = 0;
};


class Context
{
 std::map<std::string, bool> _pool;
public:
 bool lookUp(std::string name)
 {
        if (_pool.find(name) == _pool.end()) {
                throw "no exist variable";
        }

        return _pool[name];
 }

 void assign(VariableExp * variable, bool val);
};

class VariableExp: public AbstractExp
{
 std::string name;
public:
 VariableExp(std::string n): name(n)
 {
 }

 bool interpret(Context *context)
 {
        return context->lookUp(name);
 }

 std::string getName()
 {
```

```
        return name;
  }
};

class AndExp : public AbstractExp
{
  AbstractExp *first;
  AbstractExp *second;
public:
  AndExp(AbstractExp *f, AbstractExp *s): first(f), second(s)
  {

  }

  bool interpret(Context *context)
  {
        return first->interpret(context) && second->interpret(context);
  }
};

class OrExp : public AbstractExp
{
  AbstractExp *first;
  AbstractExp *second;
public:
  OrExp(AbstractExp *f, AbstractExp *s) : first(f), second(s)
  {

  }

  bool interpret(Context *context)
  {
        return first->interpret(context) || second->interpret(context);
  }
};

void Context::assign(VariableExp * variable, bool val)
{
  _pool[variable->getName()] = val;
}

int main()
{
  Context * context = new Context();
  VariableExp * a = new VariableExp("A");
```

```
  VariableExp * b = new VariableExp("B");
  VariableExp * c = new VariableExp("C");
  context->assign(a, false);
  context->assign(b, false);
  context->assign(c, true);
  OrExp *exp1 = new OrExp(a, b);
  if (exp1->interpret(context)) {
          std::cout << "A || B true" << std::endl;
  }
  else {
          std::cout << "A || B false" << std::endl;
  }
  OrExp *exp2 = new OrExp(exp1, c);
  if (exp2->interpret(context)) {
          std::cout << "(A || B) || C true" << std::endl;
  }
  else {
          std::cout << "(A || B) || C false" << std::endl;
  }

  AndExp *exp3 = new AndExp(a, b);
  if (exp3->interpret(context)) {
          std::cout << "A && B true" << std::endl;
  }
  else {
          std::cout << "A && B false" << std::endl;
  }
  AndExp *exp4 = new AndExp(exp1, c);
  if (exp4->interpret(context)) {
          std::cout << "(A && B) && C true" << std::endl;
  }
  else {
          std::cout << "(A && B) && C false" << std::endl;
  }

  delete exp4;
  delete exp3;
  delete exp2;
  delete exp1;
  delete c;
  delete b;
  delete a;
  delete context;
  return 0;
}
```

4.5. Патерн Iterator.

Призначення. Надає спосіб послідовного доступу до всіх елементів складового об'єкта, не розкриваючи його внутрішнього уявлення.

Мотивація.

Складовий об'єкт, скажімо список, повинен надавати спосіб доступу до своїх елементів, не розкриваючи їх внутрішню структуру. Більше того, іноді потрібно обходити список по-різному, залежно від завдання, що вирішується. Але навряд чи ви захочете засмічувати інтерфейс класу List операціями для різних варіантів обходу, навіть якщо всі їх можна передбачати заздалегідь. З іншого боку, іноді необхідно, щоб у той самий момент було визначено кілька активних обходів списку.

Все це дозволяє зробити патерн ітератор. Основна його ідея в тому, щоб за доступ до елементів і спосіб обходу відповідав не сам список, а окремий об'єкт-ітератор. У класі Iterator визначено інтерфейс доступу до елементів списку. Об'єкт цього класу відстежує поточний елемент, тобто він має у своєму розпорядженні інформацію, які елементи вже відвідувалися.

Перш ніж створювати екземпляр класу ListIterator, необхідно мати список, що підлягає обходу. З об'єктом ListIterator можна послідовно відвідати всі елементи списку. Операція CurrentItem повертає поточний елемент списку, операція First ініціалізує поточний елемент першим елементом списку, Next робить поточним наступний елемент, а IsDone перевіряє, чи ми опинилися за останнім елементом, якщо так, то обхід завершено.

Відділення механізму обходу від об'єкта List дозволяє визначати ітератори, реалізують різні стратегії обходу, не перераховуючи в інтерфейсі класу List. Наприклад, FilteringListIterator міг би надавати доступ тільки до тих елементів, які задовольняють умовам фільтрації.

Зауважимо: між ітератором та списком є тісний зв'язок, клієнт повинен мати інформацію, що він обходить саме список, а не якусь іншу агреговану структуру. Тому клієнт прив'язаний до конкретного способу агрегування. Було б краще, якби ми могли змінювати клас агрегату, не чіпаючи код клієнта. Це можна зробити, узагальнивши концепцію ітератора та розглянувши поліморфну ітерацію.

Наприклад, припустимо, що ми маємо ще клас SkipList, що реалізує список. Список з проаусками - це ймовірнісна структура даних, що за характеристиками нагадує збалансоване дерево. Нам потрібно навчитися писати код, здатний працювати з об'єктами як класу List, і класу SkipList.

Визначимо клас AbstractList, у якому оголошено загальний інтерфейс для маніпулювання списками. Ще нам знадобиться абстрактний клас Iterator, що визначає загальний інтерфейс ітерації. Потім ми змогли визначити конкретні підкласи класу Iterator для різних реалізацій списку. У результаті механізм ітерації виявляється незалежним від конкретних агрегованих класів.

Залишається зрозуміти, як створюється ітератор. Оскільки ми хочемо написати код, який не залежить від конкретних підкласів List, то не можна просто інстанцувати конкретний клас. Натомість ми доручимо самим об'єктам-спискам

створювати для себе відповідні ітератори, ось чому буде потрібна операція CreateIterator, за допомогою якої клієнти зможуть запитувати об'єкт-ітератор.

CreateIterator - це приклад використання патерну фабричний метод. У цьому випадку він служить для того, щоб клієнт міг запросити у об'єкта-списку відповідний ітератор. Застосування фабричного методу призводить до появи двох ієрархій класів – однієї для списків, інший для ітераторів. Фабричний метод CreateIterator «пов'язує» ці дві ієрархії.

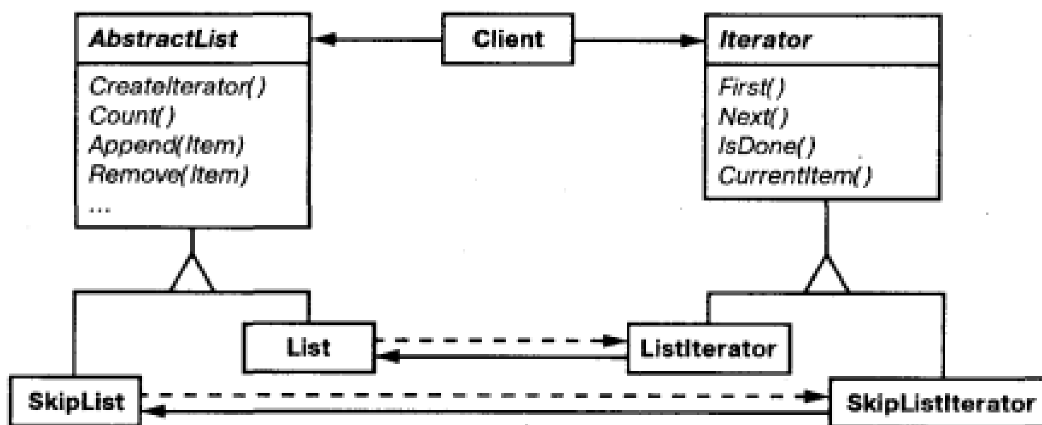

Рис. 4.7. Приклад взаємодії класів в патерні Iterator

Структура.

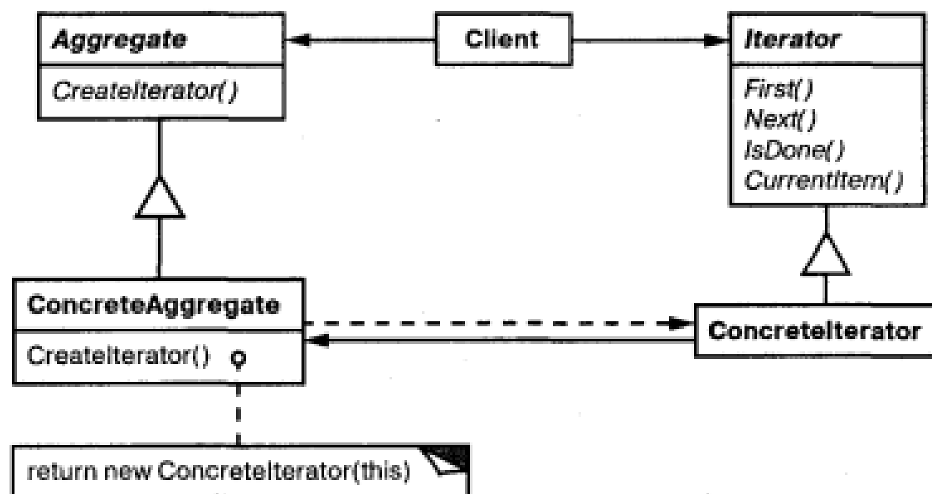

Рис. 4.8. Структура патерну Iterator

Застосування. Використовуйте патерн ітератор:

- для доступу до вмісту агрегованих об'єктів без розкриття їх внутрішнього уявлення;
- для підтримки кількох активних обходів одного й того ж агрегованого об'єкта;
- для надання одноманітного інтерфейсу з метою обходу різних агрегованих структур (тобто підтримки поліморфної ітерації).

Учасники:

- **Iterator** - ітератор: визначає інтерфейс для доступу та обходу елементів;
- **ConcreteIterator** - конкретний ітератор: реалізує інтерфейс класу Iterator; стежить за поточною позицією при обході агрегату;
- **Aggregate** - агрегат: визначає інтерфейс для створення об'єкта-ітератора;
- **ConcreteAggregate** - конкретний агрегат: реалізує інтерфейс створення ітератора та повертає екземпляр відповідного класу ConcreteIterator.

Відносини.

ConcreteIterator відстежує поточний об'єкт в агрегаті і може обчислити той, хто йде за ним.

Результати. Патерн ітератор має такі важливі особливості:

- підтримує різні види обходу агрегату. Складні агрегати можна оминати по-різному. Наприклад, для генерації коду та семантичних перевірок необхідно обходити дерева синтаксичного аналізу. Генератор коду може обходити дерево у внутрішньому чи прямому порядку. Ітератори спрощують зміну алгоритму обходу – досить просто замінити один екземпляр ітератора іншим. Для підтримки нових видів обходу можна визначити і підкласи класу Iterator;
- ітератори спрощують інтерфейс класу Aggregate. Наявність інтерфейсу для обходу в класі Iterator робить зайвим дублювання цього інтерфейсу в класі Aggregate. Тим самим інтерфейс агрегату спрощується;
- одночасно для цього агрегату може бути активно кілька обходів. Ітератор стежить за інкапсульованим у ньому станом обходу. Тому одночасно дозволяється здійснювати кілька обходів агрегату.

Реалізація. Існує безліч варіантів реалізації ітератора. Нижче перераховані найбільш уживані. Рішення про те, який спосіб вибрати, часто залежить від структур, що підтримуються мовою програмування.

- який учасник керує ітерацією. Найважливіше питання у тому, що управляє ітерацією: сам ітератор чи клієнт, який ним користується. Якщо ітерацією керує клієнт, то ітератор називається зовнішнім, в іншому випадку - внутрішнім. Навпаки, у разі внутрішнього ітератора клієнт передає ітератору деяку операцію, а ітератор вже сам застосовує цю операцію до кожного відвіданого під час обходу елемента агрегату. Зовнішні ітератори мають більшу гнучкість, ніж внутрішні. Наприклад, порівняти дві колекції на рівність за допомогою зовнішнього ітератора дуже легко, а за допомогою внутрішнього практично неможливо. Але, з іншого боку, внутрішні ітератори простіше у використанні, оскільки вони замість вас визначають логіку обходу;
- що визначає алгоритм обходу. Алгоритм обходу можна визначити у ітераторі. Його може визначити сам агрегат та використовувати ітератор лише для зберігання стану ітерації. Такі ітератор ми називаємо курсором, оскільки він лише вказує на поточну позицію в агрегаті. Клієнт викликає операцію Next агрегату, передаючи їй курсор як аргумент. А операція Next: змінює стан курсору.

Якщо за алгоритм обходу відповідає ітератор, то для того самого агрегату можна використовувати різні алгоритми ітерації, і, крім того, простіше застосувати один алгоритм до різних агрегатів. З іншого боку, алгоритму обходу

може знадобитися доступ до закритих змінних агрегатів. Якщо це так, то перенесення алгоритму до ітератора порушує інкапсуляцію агрегату;

- наскільки ітератор стійкий. Модифікація агрегату в той час, як відбувається його обхід, може виявитися небезпечною. Якщо при цьому додаються або видаляються елементи, то не виключено, що деякий елемент буде відвіданий двічі або взагалі жодного разу. Просте рішення - скопіювати агрегат і оминати копію, але зазвичай це дуже дорого.

Стійкий ітератор гарантує, що ні вставки, на видалення не завадять обходу, причому досягається без копіювання агрегату. Існує багато способів реалізації стійких ітераторів. У більшості їх ітератор реєструється в агрегаті. При вставці чи видаленні агрегат або підправляє внутрішній стан всіх створених ним ітераторів, або організує внутрішню інформацію те щоб обхід виконувався правильно.

- додаткові операції ітератора. Мінімальний інтерфейс класу Iterator складається з операцій First, Next, IsDone і CurrentItem. Але можуть виявитися корисними деякі додаткові операції. Наприклад, упорядковані агрегати можуть надавати операцію Previous, що позиціонує ітератор на попередній елемент. Для відсортованих або індексованих колекцій інтерес представляє операція SkipTo, яка позиціонує ітератор на об'єкт, що відповідає певному критерію;

- використання поліморфних ітераторів у C++. Із поліморфними ітераторами пов'язані певні накладні витрати. Необхідно, щоб об'єкт-ітератор створювався в динамічній пам'яті фабричним методом. Тому використовувати їх варто лише тоді, коли є потреба у поліморфізмі. В іншому випадку застосовуйте конкретні ітератори, які можна розподіляти в стеку.

У поліморфних ітераторів є ще один недолік: за їх видалення відповідає клієнт. Тут відкривається великий простір помилок, оскільки дуже легко забути про звільнення розподіленого з купи об'єкта-ітератора після завершення роботи з ним. Особливо велика ймовірність цього, якщо операція має кілька точок виходу.

Цю ситуацію допомагає виправити патерн заступник. Замість справжнього ітератора ми використовуємо його заступника, пам'ять якого виділено в стеку. Заступник знищує ітератора у своєму деструкторі. Тому, як тільки заступник виходить із сфери дії, разом з ним знищується і справжній ітератор. Заступник гарантує виконання належного очищення навіть у разі винятків;

- ітератори можуть мати привілейований доступ. Ітератор можна розглядати як розширення агрегату, що створив його. Ітератор та агрегат тісно пов'язані. У C++ таке ставлення можна висловити, зробивши ітератор другом свого агрегату. Тоді не потрібно визначати в агрегаті операції, єдиною метою яких є дозволити ітераторам ефективно виконати обхід. Однак наявність такого привілейованого доступу може ускладнити визначення нових способів обходу, оскільки потрібно змінити інтерфейс агрегату, додавши нового друга. Для того, щоб вирішити цю проблему, клас Iterator може включати захищені операції для доступу до важливих, але не відкритих членів агрегату. Підкласи класу Iterator (і тільки підкласи) можуть скористатися цими захищеними операціями для отримання привілейованого доступу до агрегату;

- ітератори для складових об'єктів. Реалізувати зовнішні агрегати для рекурсивно агрегованих структур (таких, наприклад, які виникають в результаті застосування патерну компонувальник) може бути важко, оскільки опис положення в структурі іноді охоплює кілька рівнів вкладеності. Тому, щоб відстежити позицію поточного об'єкта, зовнішній ітератор повинен зберігати шлях через складовий об'єкт Composite. Іноді простіше користуватися внутрішнім ітератором. Він може запам'ятати поточну позицію, рекурсивно викликаючи себе самого, тому шлях буде неявно зберігатися в стеку викликів.

Якщо вузли складового об'єкта Composite мають інтерфейс для переміщення від вузла до його братів, батьків та нащадків, то найкраще рішення дає ітератор курсорного типу. Курсору слід стежити лише за поточним вузлом, а обхід складового об'єкта може покластися на інтерфейс цього вузла.

Складові об'єкти часто потрібно оминати кількома способами. Найпоширеніші - це обхід у прямому, зворотному та внутрішньому порядку, а також обхід завширшки. Кожен вид обходу можна підтримати окремим ітератором;

- порожні ітератори. Порожній ітератор NullIterator - це вироджений ітератор, корисний під час обробки граничних умов. За визначенням, NullIterator завжди вважає, що обхід завершено, тобто його операція з IsDone незмінно повертає істину.

Застосування порожнього ітератора може спростити обхід деревоподібних структур (наприклад, об'єктів Composite). У кожній точці обходу ми запитуємо поточного елемента ітератор для його нащадків. Елементи-агрегати, як завжди, повертають конкретний ітератор. Але листові елементи повертають екземпляр NullIterator. Це дозволяє реалізувати обхід усієї структури однаково.

Наступні приклади демонструють реалізацію ітераторів для класу зв’язаного списку та стеку.

```
#include <iostream>

template <class T>
class Stack {
  T *ar;
  int count;
  int size;
public:
  Stack(int s = 999) { count = 0; size = s; ar = new T[size]; }
  ~Stack() { delete[]ar; }
  bool IsEmpty() { return !count; }
  bool IsFull() { return count == size; }
  bool push(T v) {
        if (IsFull()) return false;
        ar[count++] = v;
        return true;
  }
```

```
  T pop() { return ar[--count]; }
  void clear() { count = 0; }
  bool operator !() { return !IsEmpty(); }


  class Iterator {
          Stack<T> *stack;
          int index;
  public:
          Iterator(Stack<T> *s): stack(s), index(0) {}
          Iterator(const Iterator & i): stack(i.stack), index(i.index) {          }


          Iterator & operator =(const Iterator & i) {
                  stack = i.stack;
                  index = i.index;
                  return *this;
          }


          Iterator & begin() { index = 0; return *this; }
          //prefix
          Iterator & operator ++() { return next();  }
          //postfix
          Iterator & operator ++(int) { return next(); }
          operator T() { return stack->ar[index]; }
          bool operator !() { return index < stack->count; }

          Iterator & next() { index++; return *this; }
  };

  Iterator createIterator() { return Iterator(this); }
  operator Iterator() { return createIterator(); }

};



typedef void * LISTPOSITION;

template <class T>
class List {
public:
  class Node {
  public:
```

```
		T data;
		Node *prev, *next;
		Node(T d, Node *p = 0, Node*n = 0): data(d), prev(p), next(n) {}
	};
private:
	Node *head, *tail;
	int count;
public:
	List() { head = tail = 0; count = 0; }
	List(List &list) { head = tail = 0; count = 0; AddTail(List &list); }
	~List() { RemoveAll(); }
	int GetCount() { return count; }
	bool IsEmpty() { return head == 0; }
	LISTPOSITION GetHead() { return (LISTPOSITION)head; }
	LISTPOSITION GetTail() { return (LISTPOSITION)tail; }
	LISTPOSITION FindNode(int i);
	List & AddTail(T data);
	List & AddTail(List &list);
	List & AddHead(T data);
	T & GetNext(LISTPOSITION &p);
	T & GetPrev(LISTPOSITION &p);
	T & operator[] (int i) { return ((Node*)FindNode(i))->data; }
	T & FindAt(int i) { return ((Node*)FindNode(i))->data; }
	T RemoveHead();
	T RemoveTail();
	T RemoveAt(int i);
	List & RemoveAll();
	class Iterator {
		List<T> *list;
		Node *current;
	public:
		Iterator(List<T> *s) : list(s) { begin(); }
		Iterator(const Iterator & i) : list(i.list), current(i.current) {	}


		Iterator & operator =(const Iterator & i) {
			list = i.list;
			current = i.current;
			return *this;
		}


		Iterator & begin() { current = (Node *)list->GetHead(); return *this; }
		//prefix
		Iterator & operator ++() { return next(); }
```

```
      //postfix
      Iterator & operator ++(int) { return next(); }
      operator T() { return current->data; }
      bool operator !() { return current > 0; }

      Iterator & next() { current = current->next; return *this; }
  };

  Iterator createIterator() { return Iterator(this); }
  operator Iterator() { return createIterator(); }
};


template <class T>
List<T> & List<T>::AddTail(T data) {
  Node *t = new Node(data, tail);
  if (tail) tail->next = t;
  tail = t;
  if (!head) head = t;
  count++;
  return *this;
}

template <class T>
List<T> & List<T>::AddTail(List &list) {
  LISTPOSITION p = list.GetHead();
  while (p) AddTail(list.GetNext(p));
  return *this;
}

template <class T>
List<T> & List<T>::AddHead(T data) {
  Node *t = new Node(data, 0, head);
  if (head) head->prev = t;
  head = t;
  if (!tail) tail = t;
  count++;
  return *this;
}

template <class T>
T &List<T>::GetNext(LISTPOSITION &p) {
  Node *t = (Node*)p;
  p = (LISTPOSITION)t->next;
  return t->data;
```

```
}

template <class T>
T & List<T>::GetPrev(LISTPOSITION &p) {
  Node *t = (Node*)p;
  p = (LISTPOSITION)t->prev;
  return t->data;
}

template <class T>
LISTPOSITION List<T>::FindNode(int i) {
  if (i < 0 || i >= count) throw 1;
  int j = 0;
  Node *t = head;
  while (t) {
          if (i == j) return (LISTPOSITION)t;
          t = t->next;
          j++;
  }
  throw 1;
}

template <class T>
T List<T>::RemoveAt(int i) {
  Node *t = FindNode(i);
  T data = t->data;
  if (t->prev) t->prev->next = t->next;
  if (t->next) t->next->prev = t->prev;
  if (t == head) head = t->next;
  if (t == tail) tail = t->prev;
  delete t;
  count--;
  return data;
}

template <class T>
List<T> & List<T>::RemoveAll() {
  if (!head) return *this;
  Node *t = head;
  while (t) {
          Node *v = t->next;
          delete t;
          t = v;
  }
  head = tail = 0;
```

```
  count = 0;
  return *this;
}


template <class T>
T List<T>::RemoveHead() {
  if (!head) throw 1;
  T data = head->data;
  if (head->next) head->next->prev = 0;
  Node *v = head;
  head = head->next;
  if (v == tail) tail = 0;
  delete v;
  count--;
  return data;
}

template <class T>
T List<T>::RemoveTail() {
  if (!tail) throw 1;
  T data = tail->data;
  if (tail->prev) tail->prev->next = 0;
  Node *v = tail;
  tail = tail->prev;
  if (v == head) head = 0;
  delete v;
  count--;
  return data;
}


int main()
{
  Stack<int> s;

  s.push(5);
  s.push(7);
  s.push(10);

  /*
  while (!s)
  {
          std::cout << s.pop() << std::endl;
  }
```

```
    */

    Stack<int>::Iterator it = s;
    for (it.begin(); !it; ++it) {
            std::cout << it << std::endl;
    }

    List<int> list;

    list.AddTail(15);
    list.AddTail(17);
    list.AddTail(110);

    LISTPOSITION p = list.GetHead();

    while (p)
    {
            int data = list.GetNext(p);
            std::cout << data << std::endl;
    }

    std::cout << std::endl;

    List<int>::Iterator it1 = list;
    for (it1.begin(); !it1; ++it1) {
            std::cout << it1 << std::endl;
    }

}
```

4.6. Патерн Mediator.

Призначення. Визначає об'єкт, що інкапсулює спосіб взаємодії безлічі об'єктів. Посередник забезпечує слабку пов'язаність системи, позбавляючи об'єкти необхідності явно посилатися один на одного і дозволяючи цим незалежно змінювати взаємодії з-поміж них.

Мотивація.

Об'єктно-орієнтоване проектування сприяє розподілу певної поведінки між об'єктами. Але при цьому в структурі об'єктів, що вийшла, може виникнути багато зв'язків або (у гіршому випадку) кожному об'єкту доведеться мати інформацію про всіх інших.

Незважаючи на те, що розбиття системи на безліч об'єктів у загальному випадку підвищує ступінь повторного використання, проте ріноманітність взаємозв'язків призводить до зворотного ефекту. Якщо взаємозв'язків занадто багато, тоді система подібна до моноліту і малоймовірно, що об'єкт зможе

працювати без підтримки інших об'єктів. Більше того, суттєво змінити поведінку системи практично неможливо, оскільки вона розподілена між багатьма об'єктами. Якщо ви спробуєте, то для налаштування поведінки системи вам доведеться визначати безліч підкласів.

Розглянемо реалізацію діалогових вікон у графічному інтерфейсі користувача. Тут розташовується ряд віджетів: кнопки, меню, поля введення тощо, як показано малюнку.

Часто між різними віджетами в діалоговому вікні є залежності. Наприклад, якщо одне з полів введення порожнє, певна кнопка недоступна. Вибір зі списку може змінювати вміст поля введення. І навпаки, введення тексту в поле може автоматично призвести до вибору одного або декількох елементів списку. Якщо в полі введення є якийсь текст, то можуть бути активізовані кнопки, що дозволяють зробити певну дію над цим текстом, наприклад, змінити або видалити його.

У різних діалогових вікнах залежність між віджетами може бути різною. Тому, незважаючи на те, що у всіх вікнах зустрічаються однотипні віджети, просто взяти і повторно використовувати готові класи віджетів не вдасться, доведеться робити налаштування з метою врахування залежностей. Індивідуальне налаштування кожного віджету - стомлююче заняття, бо класів, що беруть участь, занадто багато.

Всіх цих проблем можна уникнути, якщо інкапсулювати колективну поведінку в окремому посереднику. Посередник відповідає за координацію взаємодій між групою об'єктів. Він позбавляє об'єкти, що входять до групи, необхідності явно посилатися один на одного. Всі об'єкти мають інформацію лише про посередника, тому кількість взаємозв'язків скорочується.

Так, наприклад, клас FontDialogDirector може служити посередником між віджетами в діалоговому вікні. Об'єкт цього класу «знає» про всі віджети у вікні та координує взаємодію між ними, тобто виконує функції центру комунікацій.

Об'єкти кооперуються один з одним, реагуючи на зміну обраного елемента списку.

Послідовність подій, в результаті яких інформація про вибраний елемент списку передається в поле введення, така:

1. Список інформує розпорядника про зміни, що відбулися в ньому.
2. Розпорядник отримує від списку вибраний елемент.
3. Розпорядник передає обраний елемент полю введення.
4. Тепер, коли поле введення містить інформацію, розпорядник активізує кнопки, що дозволяють виконати певну дію (наприклад, змінити шрифт на жирний або курсив).

Зверніть увагу, як розпорядник здійснює посередництво між списком і полем введення. Віджети спілкуються один з одним не безпосередньо, а через розпорядник. Їм взагалі не потрібно володіти інформацією один про одного, вони знають лише про існування розпорядника. А якщо поведінка локалізована в одному класі, то її нескладно модифікувати або зробити зовсім іншим шляхом розширення або заміни цього класу.

DialogDirector - це абстрактний клас, який визначає поведінку діалогового

вікна в цілому. Клієнти викликають операцію ShowDialog для відображення вікна на екрані. CreateWidgets - це абстрактна операція для створення віджетів у діалоговому вікні. WidgetChanged – ще одна абстрактна операція; з її допомогою віджети повідомляють розпоряднику про зміни. Підкласи DialogDirector заміщають операції CreateWidgets.йде1;з (для створення потрібних віджетів) та WidgetChanged (для обробки повідомлень про зміни).

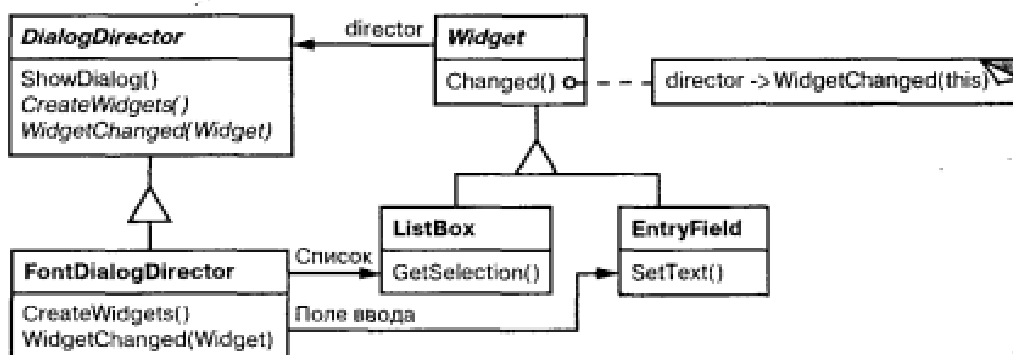

Рис. 4.9. Приклад взаємодії класів в патерні Mediator

Застосування. Використовуйте патерн посередник, коли

- є об'єкти, зв'язки між якими складні та чітко визначені. Взаємозалежності, що виходять при цьому, не структуровані і важкі для розуміння;
- не можна повторно використовувати об'єкт, оскільки він обмінюється інформацією з багатьма іншими об'єктами;
- поведінка, розподілена між кількома класами, повинна піддаватися налаштуванню без породження безлічі підкласів.

Структура.

Рис. 4.10. Структура патерну Mediator

Учасники:

- **Mediator** (DialogDirector) - посередник: визначає інтерфейс для обміну інформацією з об'єктами Colleague;
- **ConcreteMediator** (FontDialogDirector) - конкретний посередник: реалізує кооперативну поведінку, координуючи дії об'єктів Colleague; володіє інформацією про колег та підраховує їх;
- **Класи Colleague** (ListBox, EntryField) – колеги:- кожен клас Colleague

«знає» про свій об'єкт Mediator; всі колеги обмінюються інформацією лише з посередником, оскільки за його відсутності їм довелося спілкуватися між собою напряму.

Відносини.

Колеги надсилають запити посереднику та отримують запити від нього. Посередник реалізує кооперативну поведінку шляхом переадресації кожного запиту до відповідного колеги (або кількох колег).

Результати. У патерна посередник є такі переваги та недоліки:

- знижує кількість підкласів, що породжуються. Посередник локалізує поведінку, яку інакше довелося б розподіляти між кількома об'єктами. Для зміни поведінки потрібно породити підкласи тільки від класу посередника Mediator, класи колег Colleague можна використовувати повторно без будь-яких змін;
- усуває пов'язаність між колегами. Посередник забезпечує слабку пов'язаність колег. Змінювати класи Colleague та Mediator можна незалежно один від одного;
- спрощує протоколи взаємодії об'єктів. Посередник замінює дисципліну взаємодії «все з усіма» дисципліною «один із усіма», тобто один посередник взаємодіє з усіма колегами. Відносини виду "один до багатьох" простіше для розуміння, супроводу та розширення;
- абстрагує спосіб кооперування об'єктів. Виділення механізму посередництва в окрему концепцію та інкапсуляція її в одному об'єкті дозволяє зосередитися саме на взаємодії об'єктів, а не на їхній індивідуальній поведінці. Це дозволяє прояснити наявні у системі взаємодії;
- централізує керування. Паттерн посередник переносить складність взаємодії у клас-посередник. Оскільки посередник інкапсулює протоколи, він може бути складніше окремих колег. В результаті сам посередник стає монолітом, який важко супроводжувати.

Реалізація.

Майте на увазі, що при реалізації патерну посередник може відбуватися:

- звільнення від абстрактного класу Mediator. Якщо колеги працюють тільки з одним посередником, то немає необхідності визначати абстрактний клас Mediator. Абстракція, що забезпечується класом Mediator, дозволяє колегам працювати з різними підкласами класу Mediator і навпаки;
- обмін інформацією між колегами та посередником. Колеги повинні обмінюватися інформацією зі своїм посередником тільки тоді, коли виникає подія, що представляє інтерес. Одним із підходів до реалізації посередника є застосування патерну спостерігач. Тоді класи колег діють як суб'єкти, що посилають повідомлення посереднику про зміну свого стану. Посередник реагує на них, повідомляючи про це інших колег.

Інший підхід: у класі Mediator визначається спеціалізований інтерфейс сповіщення, який дозволяє колегам обмінюватися інформацією більш вільно.

```
#include <iostream>

class FileSelectionDialog;
```

```
class Widget
{
public:
  Widget(FileSelectionDialog *mediator, const char *name)
  {
          _mediator = mediator;
          strcpy_s(_name, 20, name);
  }
  virtual void changed();
  virtual void updateWidget() = 0;
  virtual void queryWidget() = 0;
protected:
  char _name[20];
private:
  FileSelectionDialog *_mediator;
};

class List : public Widget
{
public:
  List(FileSelectionDialog *dir, const char *name) : Widget(dir, name) { }
  void queryWidget()
  {
          std::cout << "   " << _name << " list queried" << std::endl;
  }
  void updateWidget()
  {
          std::cout << "   " << _name << " list updated" << std::endl;
  }
};

class Edit : public Widget
{
public:
  Edit(FileSelectionDialog *dir, const char *name) : Widget(dir, name) { }
  void queryWidget()
  {
          std::cout << "   " << _name << " edit queried" << std::endl;
  }
  void updateWidget()
  {
          std::cout << "   " << _name << " edit updated" << std::endl;
  }
};
```

```
class FileSelectionDialog
{
public:
  enum Widgets
  {
          FilterEdit, DirList, FileList, SelectionEdit
  };
  FileSelectionDialog()
  {
          _components[FilterEdit] = new Edit(this, "filter");
          _components[DirList] = new List(this, "dir");
          _components[FileList] = new List(this, "file");
          _components[SelectionEdit] = new Edit(this, "selection");
  }
  virtual ~FileSelectionDialog();
  void handleEvent(int which)
  {
          _components[which]->changed();
  }
  virtual void widgetChanged(Widget *theChangedWidget)
  {
          if (theChangedWidget == _components[FilterEdit])
          {
                  _components[FilterEdit]->queryWidget();
                  _components[DirList]->updateWidget();
                  _components[FileList]->updateWidget();
                  _components[SelectionEdit]->updateWidget();
          }
          else if (theChangedWidget == _components[DirList])
          {
                  _components[DirList]->queryWidget();
                  _components[FileList]->updateWidget();
                  _components[FilterEdit]->updateWidget();
                  _components[SelectionEdit]->updateWidget();
          }
          else if (theChangedWidget == _components[FileList])
          {
                  _components[FileList]->queryWidget();
                  _components[SelectionEdit]->updateWidget();
          }
          else if (theChangedWidget == _components[SelectionEdit])
          {
                  _components[SelectionEdit]->queryWidget();
                  std::cout << "   file opened" << std::endl;
          }
```

```
  }
private:
  Widget *_components[4];
};

FileSelectionDialog::~FileSelectionDialog()
{
  for (int i = 0; i < 4; i++)
        delete _components[i];
}

void Widget::changed()
{
  _mediator->widgetChanged(this);
}

int main()
{
  FileSelectionDialog fileDialog;
  int i;

  std::cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
  std::cin >> i;

  while (i)
  {
        fileDialog.handleEvent(i - 1);
        std::cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
        std::cin >> i;
  }
  return 0;
}
```

4.7. Патерн Memento.

Призначення. Не порушуючи інкапсуляції, фіксує і виносить межі об'єкта його внутрішній стан так, щоб пізніше можна було відновити у ньому об'єкт.

Мотивація.

Іноді необхідно в той чи інший спосіб зафіксувати внутрішній стан об'єкта. Така потреба виникає, наприклад, при реалізації контрольних точок та механізмів відкату, що дозволяють користувачеві скасувати пробну операцію або відновити стан після помилки. Його необхідно десь зберегти, щоб пізніше відновити у ньому об'єкт. Але зазвичай об'єкти інкапсулюють усі свій стан або хоча б його частину, роблячи його недоступним для інших об'єктів, тому зберегти стан ззовні неможливо. Розкриття ж стану було б порушенням

принципу інкапсуляції і поставило під загрозу надійність і розширюваність програми.

Розглянемо, наприклад, графічний редактор, що підтримує пов'язаність об'єктів. Користувач може з'єднати два прямокутники лінією, і вони залишаться в такому положенні за будь-яких переміщень. Редактор сам перемальовує лінію, зберігаючи пов'язаність зміни.

Система дозволу обмежень – добре відомий спосіб підтримки зв'язаності між об'єктами. Її функції можуть виконуватися об'єктом класу ConstraintSolver, який реєструє новостворені сполуки і генерує математичні рівняння, що їх описують. Коли користувач якимось чином модифікує діаграму, об'єкт вирішує ці рівняння. Результати обчислень об'єкт ConstraintSolver використовує для перемалювання графіки так, щоб були збережені всі з'єднання.

Підтримка відкату операцій у додатках не така проста, як може здатися на перший погляд.

Очевидний спосіб відкотити операцію переміщення - це зберегти відстань між старим та новим становищем, а потім перемістити об'єкт на таку саму відстань назад. Однак при цьому не гарантується, що всі об'єкти будуть там же, де знаходилися. Припустимо, що у способі розташування сполучної лінії є певна свобода. Тоді, перемістивши прямокутник на колишнє місце, ми можемо не досягти бажаного ефекту.

Відкритого інтерфейсу ConstraintSolver іноді не вистачає для точного відкату всіх змін суміжних об'єктів. Механізм відкату повинен працювати в тісній взаємодії з ConstraintSolver для відновлення попереднього стану, але необхідно також подбати про те, щоб внутрішні деталі ConstraintSolver не були доступні цьому механізму.

Паттерн зберігач допоможе вирішити цю проблему. Зберігач – це об'єкт, у якому зберігається внутрішній стан іншого об'єкта – господаря зберігача. Для роботи механізму відкату потрібно, щоб господар надав зберігач, коли виникне потреба записати контрольну точку стану господаря. Тільки господареві дозволено поміщати в зберігач інформацію та витягати її звідти, для інших об'єктів зберігач непрозорий.

У прикладі графічного редактора, який обговорювався вище, у ролі господаря може виступати об'єкт СопзРгахпЬЗо^ег. Процес відкату характеризується такою послідовністю подій:

1. Редактор запитує зберігач у об'єкта ConstraintSolver у процесі виконання операції переміщення.

2. ConstraintSolver створює і повертає зберігач, у даному прикладі екземпляр класу SolverState. Зберігач SolverState містить структури даних, що описують поточний стан внутрішніх рівнянь та змінних ConstraintSolver.

3. Пізніше, коли користувач скасовує операцію переміщення, редактор повертає SolverState об'єкту ConstraintSolver.

4. На основі інформації, що зберігається в об'єкті SolverState, ConstraintSolver змінює свої внутрішні структури, повертаючи рівняння та змінні в початковий стан.

Така організація дозволяє об'єкту ConstraintSolver «знайомить» інші об'єкти

з інформацією, яка йому необхідна для повернення до попереднього стану, не розкриваючи водночас свою структуру та подання.

Застосування. Використовуйте патерн зберігач, коли:

- необхідно зберегти миттєвий знімок стану об'єкта (або його частини), щоб згодом об'єкт можна було відновити у тому ж стані;
- пряме отримання цього стану розкриває деталі реалізації та порушує інкапсуляцію об'єкта.

Структура.

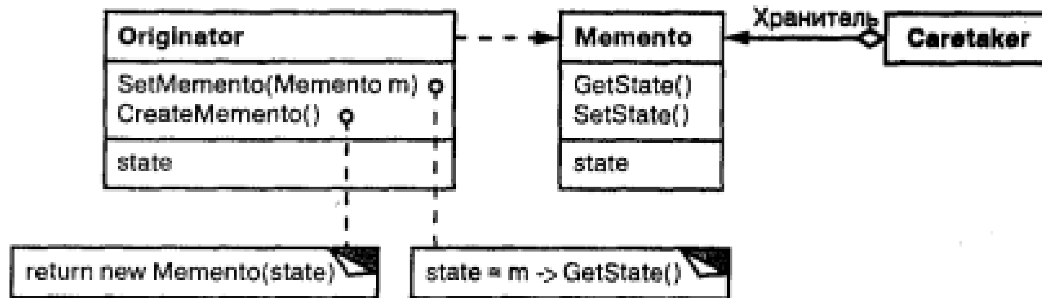

Рис. 4.11. Структура патерну Memento

Учасники:

- **Memento** (SolverState) - зберігач: зберігає внутрішній стан об'єкта Originator. Обсяг інформації може бути різним і визначається потребами господаря; забороняє доступ до всіх інших об'єктів, крім господаря. По суті, зберігачі мають два інтерфейси. «Посильний» Caretaker «бачить» лише «вузький» інтерфейс зберігача - може лише передавати зберігача іншим об'єктам. Навпаки, господареві доступний «широкий» інтерфейс, який забезпечує доступ до всіх даних, необхідних для відновлення в колишньому стані. Ідеальний варіант - коли лише господареві, який створив зберігач, відкритий доступ до внутрішнього стану останнього;
- **Originator** (ConstraintSolver) - господар: створює зберігач, що містить знімок поточного внутрішнього стану; використовує зберігач відновлення внутрішнього стану;
- **Caretaker** (механізм відкату) - посильний: відповідає за збереження зберігача; не здійснює жодних операцій над зберігачем і не досліджує його внутрішній вміст.

Відносини:

- посильний запитує зберігач у господаря, якийсь час тримає його в себе, а потім повертає господареві.

Іноді цього немає, оскільки останньому не потрібно відновлювати колишній стан;

- зберігачі пасивні. Тільки господар, який створив зберігач, має доступ до інформації про стан.

Результати. Характерні особливості патерну зберігач:

- збереження меж інкапсуляції. Зберігач дозволяє уникнути розкриття інформації, якою повинен розпоряджатися лише господар, але яку, проте, необхідно зберігати поза господарем. Цей патерн екранує об'єкти від потенційно

складного внутрішнього устрою господаря, не змінюючи межі інкапсуляції;

- спрощення структури господаря. При інших варіантах дизайну, спрямованого на збереження меж інкапсуляції, господар зберігає в собі версії внутрішнього стану, який запитували клієнти. Таким чином, вся відповідальність за керування пам'яттю лежить на господареві. При перекладанні турботи про запит на клієнтів спрощується структура господаря, а клієнтам дається можливість не інформувати господаря про те, що вони закінчили роботу;
- значні витрати під час використання зберігачів. Зі зберігачами можуть бути пов'язані помітні витрати, якщо господар повинен копіювати великий обсяг інформації для занесення на згадку зберігача або якщо клієнти створюють і повертають зберігачів досить часто. Якщо плата за інкапсуляцію та відновлення стану господаря велика, цей патерн не завжди підходить;
- визначення «вузького» та «широкого» інтерфейсів. У деяких мовах складно гарантувати, що лише господар має доступ до стану зберігача;
- прихована плата за утримання зберігача. Посильний відповідає за видалення зберігача, проте не має інформації, який обсяг інформації про стан прихований у ньому. Тому невибагливий до ресурсів посильний може витрачати дуже багато пам'яті під час роботи з хранителем.

Реалізація. При реалізації патерну зберігач слід пам'ятати:

- мовну підтримку. У зберігачів є два інтерфейси; "широкий" для господарів і "вузький" для всіх інших об'єктів. В ідеалі мову реалізації має підтримувати два рівні статичного контролю доступу. У C++ це можливо, якщо оголосити господаря другом зберігача і зробити закритим «широкий» інтерфейс останнього. Відкритим залишається лише «вузький» інтерфейс;
- збереження інкрементних змін. Якщо зберігачі створюються та повертаються своєму господареві в передбачуваній послідовності, то зберігач може зберегти лише зміни у внутрішньому стані господаря. Наприклад, команди, які допускають скасування, у списку історії можуть користуватися зберігачами для відновлення початкового стану. Список історії призначений лише для скасування та повтору команд. Це означає, що зберігачі можуть працювати лише зі змінами, зробленими командою, а не з повним станом об'єкта.

```
#include <iostream>

class State
{
 public:
	enum StateStatus {
		Empty = 0,
		Created,
		Opened,
		Assigned,
		Closed,

		ErrorStatus
	};
```

```
  private:
        StateStatus state;

  public:
        State(StateStatus s)
        {
              ensureIsValidState(s);
              state = s;
        }

        State(const State& s)
        {
              ensureIsValidState(s.state);
              state = s.state;
        }

        State  & operator = (const State & s) {
              ensureIsValidState(s.state);
              state = s.state;
              return *this;
        }

        void ensureIsValidState(StateStatus state)
        {
              if (state <= Empty || state >= ErrorStatus) {
                    throw "Invalid state given";
              }
        }

        int getState() { return state; }
};

class Memento
{
        friend class Ticket;
        State state;
        Memento(State s) : state(s) {}
  public:
        Memento(const Memento & m) : state(m.state) {}
        Memento  operator = (const Memento & m) {
              state = m.state;
              return *this;
        }
        State getState() {return state;    }
};
```

```
//originator
class Ticket
{
  State currentState;
public:
  Ticket():currentState(State::Created) {}

  void open()
  {
          currentState = State(State::Opened);
  }

  void assign()
  {
          currentState = State(State::Assigned);
  }

  void close()
  {
          currentState = State(State::Closed);
  }

  Memento save()
  {
          return Memento(currentState);
  }

  void restore(Memento memento)
  {
          currentState = memento.getState();
  }

  int getState()
  {
          return currentState.getState();
  }
};

int main()
{

  Ticket ticket;
  ticket.open();
  std::cout << ticket.getState() << std::endl;
```

```
  ticket.assign();
  std::cout << ticket.getState() << std::endl;
  Memento m = ticket.save();
  ticket.close();
  std::cout << ticket.getState() << std::endl;
  ticket.restore(m);
  std::cout << ticket.getState() << std::endl;

  //error. only for originator
  //Memento m1;

  return 0;
}
```

4.8. Патерн Observer.

Призначення. Визначає залежність типу "один до багатьох" між об'єктами таким чином, що при зміні стану одного об'єкта всі, що залежать від нього, сповіщаються про це і автоматично оновлюються.

Мотивація.

В результаті розбиття системи на безліч спільно працюючих класів виникає потреба підтримувати узгоджений стан взаємопов'язаних об'єктів. Але не хотілося б, щоб за узгодженість треба було платити жорсткою зв'язаністю класів, оскільки це певною мірою зменшує можливості повторного використання.

Наприклад, у багатьох бібліотеках для побудови графічних інтерфейсів користувача презентаційні аспекти інтерфейсу відокремлені від даних програми. З класами, що описують дані та їх подання, можна працювати автономно. Електронна таблиця та діаграма не мають інформації один про одного, тому ви маєте право використовувати їх окремо. Але поводяться вони так, ніби «знають» одне про одного. Коли користувач працює з таблицею, всі зміни негайно відбиваються на діаграмі, і навпаки.

При такій поведінці мається на увазі, що і електронна таблиця, і діаграма залежать від даних об'єкта і тому повинні повідомлятись про будь-які зміни в його стані. І немає жодних причин, що обмежують кількість залежних об'єктів; для роботи з одними і тими ж даними може існувати будь-яке число інтерфейсів користувача.

Паттерн спостерігач визначає, як встановлювати такі відносини. Ключовими об'єктами у ньому є суб'єкт та спостерігач. У суб'єкта може бути скільки завгодно залежних від нього спостерігачів. Усі спостерігачі повідомляються про зміни у стані суб'єкта. Отримавши повідомлення, спостерігач опитує суб'єкта, щоб синхронізувати свій стан.

Така взаємодія часто називається ставленням видавець-передплатник. Суб'єкт видає або публікує повідомлення та розсилає їх, навіть не маючи інформації про те, які об'єкти є передплатниками. На отримання повідомлень може підписатися необмежену кількість спостерігачів.

Застосування. Використовуйте патерн спостерігач у таких ситуаціях:

- коли абстракція має два аспекти, один з яких залежить від іншого. Інкапсуляції цих аспектів у різні об'єкти дозволяють змінювати та повторно використовувати їх незалежно;
- коли при модифікації одного об'єкта потрібно змінити інші, і ви не знаєте, скільки саме об'єктів потрібно змінити;
- коли один об'єкт повинен сповіщати інших, не роблячи припущень про повідомлені об'єкти. Іншими словами, ви не хочете, щоб об'єкти були тісно пов'язані між собою.

Структура.

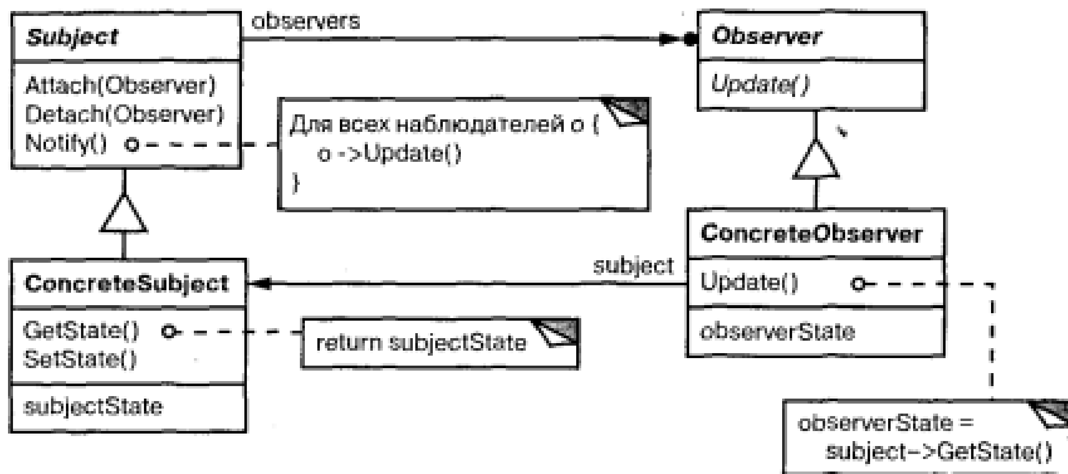

Рис. 4.12. Структура патерну Observer

Учасники:

Subject - суб'єкт: має у своєму розпорядженні інформацію про своїх спостерігачів. За суб'єктом може «стежити» будь-яка кількість спостерігачів; надає інтерфейс для приєднання та відділення спостерігачів;

Observer - спостерігач: визначає інтерфейс оновлення для об'єктів, які повинні бути повідомлені про зміну суб'єкта;

ConcreteSubject - конкретний суб'єкт: зберігає стан, що представляє інтерес для конкретного спостерігача ConcreteObserver; надсилає інформацію своїм спостерігачам, коли відбувається зміна;

ConcreteObserver - конкретний спостерігач: зберігає посилання на об'єкт класу ConcreteSubject; зберігає дані, які мають бути узгоджені з даними суб'єкта; реалізує інтерфейс оновлення, визначений у класі Observer, щоб підтримувати узгодженість із суб'єктом.

Відносини:

- об'єкт ConcreteSubject повідомляє своїх спостерігачів про будь-яку зміну, яка могла б призвести до неузгодженості станів спостерігача та суб'єкта;
- після отримання від конкретного суб'єкта повідомлення про зміну об'єкт ConcreteObserver може запитати у суб'єкта додаткову інформацію, яку використовує у тому, щоб опинитися у стані, узгодженому зі станом суб'єкта.

Результати.

Паттерн спостерігач дозволяє змінювати суб'єкти та спостерігачі незалежно

один від одного. Суб'єкти дозволяється повторно використовувати без участі спостерігачів і навпаки. Це дозволяє додавати нових спостерігачів без модифікації суб'єкта чи інших спостерігачів.

Розглянемо деякі переваги та недоліки патерну спостерігач:

- абстрактна пов'язаність суб'єкта та спостерігача. Суб'єкт має інформацію лише про те, що він має низку спостерігачів, кожен із яких підпорядковується простому інтерфейсу абстрактного класу Observer. Суб'єкту невідомі конкретні класи спостерігачів. Таким чином, зв'язки між суб'єктами та спостерігачами носять абстрактний характер і зведені до мінімуму.

Оскільки суб'єкт і спостерігач є тісно пов'язаними, всі вони можуть перебувати на різних рівнях абстракції системи. Суб'єкт нижчого рівня може повідомляти спостерігачів на верхніх рівнях, не порушуючи ієрархії системи. Якби суб'єкт і спостерігач являли собою єдине ціле, то об'єкт, що виходить, або перетинав би межі рівнів (порушуючи принцип їх формування), або повинен був перебувати на якомусь одному рівні (компрометуючи абстракцію рівня);

- підтримка широкомовних комунікацій. На відміну від звичайного запиту для повідомлення, що надсилається суб'єктом, не потрібно задавати певного одержувача. Повідомлення автоматично надходить всім об'єктам, що підписалися на нього. Суб'єкту не потрібна інформація про кількість таких об'єктів, від нього потрібно лише повідомити своїх спостерігачів. Тому ми можемо у будь-який час додавати та видаляти спостерігачів. Спостерігач сам вирішує, обробити отримане повідомлення чи ігнорувати його;

- несподівані оновлення. Оскільки спостерігачі не мають інформації один про одного, їм невідомо і про те, у що обходиться зміна суб'єкта. Нешкідлива, на перший погляд, операція над суб'єктом може викликати цілу низку оновлень спостерігачів та об'єктів, що залежать від них. Більше того, нечітко визначені або погано підтримувані критерії залежності можуть спричинити непередбачені оновлення, відстежити які дуже складно.

Ця проблема посилюється ще й тим, що простий протокол оновлення не містить жодної інформації про те, що саме змінилося в суб'єкті. Без додаткового протоколу, який допомагає з'ясувати характер змін, спостерігачі будуть змушені зробити складну роботу для непрямого отримання такої інформації.

Реалізація.

У цьому розділі обговорюються питання щодо реалізації механізму залежностей:

- відображення суб'єктів на спостерігачів. За допомогою цього найпростішого способу суб'єкт може відстежити всіх спостерігачів, яким він повинен надсилати повідомлення, тобто зберігати на них явні посилання. Проте за наявності великої кількості суб'єктів і лише кількох спостерігачів може виявитися накладно. Один із можливих компромісів на користь економії пам'яті за рахунок часу полягає в тому, щоб використовувати асоціативний масив (наприклад, хеш-таблицю) для зберігання відображення між суб'єктами та спостерігачами. Тоді суб'єкт, який не має спостерігачів, не дарма витрачатиме пам'ять. З іншого боку, за такого підходу збільшується час пошуку спостерігачів;

- спостереження більш ніж за одним суб'єктом. Іноді спостерігач може

залежати від одного суб'єкта. Наприклад, в електронній таблиці буває більше одного джерела даних. У таких випадках необхідно розширити інтерфейс Update, щоб спостерігач міг дізнатися, який суб'єкт надіслав повідомлення. Суб'єкт може просто передати себе як параметр операції Update, тим самим повідомляючи спостерігачеві, що саме потрібно обстежити;

- хто ініціює оновлення. Щоб зберегти узгодженість, суб'єкт та його спостерігачі покладаються на механізм повідомлень. Але який об'єкт викликає операцію Notify для ініціювання оновлення? Є два варіанти:
  - ✓ операції класу Subject, що змінили стан, викликають Notify для повідомлення про цю зміну. Перевага такого підходу полягає в тому, що клієнтам не треба пам'ятати про необхідність викликати операцію Notify суб'єкта. Недолік полягає в наступному: при виконанні кожної з кількох послідовних операцій будуть проводитися оновлення, що може стати причиною неефективної роботи програми;
  - ✓ відповідальність за своєчасний виклик Notify покладається на клієнта. Перевага: клієнт може відкласти ініціювання оновлення до завершення серії змін, виключивши цим непотрібні проміжні оновлення. Недолік: клієнти мають додатковий обов'язок. Це збільшує ймовірність помилок, оскільки клієнт може забути викликати Notify;
- висячі посилання на віддалені суб'єкти. Видалення суб'єкта має призводити до появи висячих посилань в спостерігачів. Уникнути цього можна, наприклад, доручивши суб'єкту повідомляти всі свої спостерігачі про своє видалення, щоб вони могли знищити посилання, що зберігаються у себе. У загальному випадку просте видалення спостерігачів не годиться, оскільки на них можуть посилатися інші об'єкти та під їх спостереженням можуть бути інші суб'єкти;
- гарантії несуперечності стану суб'єкта перед відправкою повідомлення. Важливо бути впевненим, що перед викликом операції Notify стан суб'єкта несуперечливий, оскільки в процесі оновлення власного стану спостерігачі опитуватимуть стан суб'єкта. Правило несуперечності дуже легко порушити, якщо операції одного з підкласів класу Subject викликають успадковані операції.
- як уникнути залежності протоколу оновлення від спостерігача: моделі витягування та проштовхування. У реалізаціях патерну спостерігач суб'єкт часто транслює всім передплатникам додаткову інформацію про характер зміни. Вона передається як аргументу операції Update, і її обсяг змінюється у широких діапазонах.

На одному полюсі знаходиться так звана модель проштовхування, коли суб'єкт надсилає спостерігачам детальну інформацію про зміну незалежно від того, чи потрібно їм це. На іншому - модель витягування, коли суб'єкт не посилає нічого крім мінімального повідомлення, а спостерігачі запитують деталі пізніше.

У моделі витягування підкреслюється неінформованість суб'єкта про своїх спостерігачів, а в моделі проштовхування передбачається, що суб'єкт має певну інформацію про потреби спостерігачів. У разі застосування моделі

проштовхування ступінь повторного їх використання може знизитися, тому що класи Subject припускають про класи Observer, які не завжди можуть бути вірними. З іншого боку, модель витягування може виявитися неефективною, бо спостерігачам без допомоги суб'єкта необхідно з'ясовувати, що змінилося;

- явне специфікування модифікацій, що представляють інтерес модифікацій. Ефективність оновлення можна підвищити, розширивши інтерфейс реєстрації суб'єкта, тобто надавши можливість реєстрації спостерігача вказати, які події його цікавлять. Коли подія відбувається, суб'єкт інформує лише тих спостерігачів, які виявили щодо нього інтерес. Щоб отримувати конкретну подію, спостерігачі приєднуються до своїх суб'єктів так:

```
void Subject::Attach(Observer *, Aspect &interest)
```

де interes визначає подію, що представляє інтерес. У момент посилання повідомлення суб'єкт передає своїм спостерігачам аспект, що змінився у вигляді параметра операції Update. Наприклад:

```
void Observer::Update(Subject *, Aspect &interest)
```

- інкапсуляція складної семантики оновлення. Якщо відносини залежності між суб'єктами та спостерігачами стають особливо складними, то може знадобитися об'єкт, що інкапсулює ці відносини. Будемо називати його ChangeManager (менеджер змін). Він служить для мінімізації обсягу роботи, необхідної для того, щоб спостерігачі змогли відобразити зміни суб'єкта. Наприклад, якщо певна операція тягне за собою зміни в кількох незалежних суб'єктах, то хотілося б, щоб спостерігачі повідомлялися після того, як будуть модифіковані всі суб'єкти, щоб не повідомляти одного й того ж спостерігача кілька разів.

Клас ChangeManager має три обов'язки:

- ✓ будувати відображення між суб'єктом та його спостерігачами та надавати інтерфейс для підтримки відображення в актуальному стані. Це звільняє суб'єктів від необхідності зберігати посилання своїх спостерігачів і навпаки;
- ✓ визначати конкретну стратегію оновлення;
- ✓ оновлювати всіх залежних спостерігачів на запит від суб'єкта.

- комбінування класів Subject і Observer. У бібліотеках класів, які написані мовами, що не підтримують множинного успадкування, зазвичай не визначаються окремі класи Subject і Observer. Їх інтерфейси комбінуються в одному класі. Це дозволяє визначити об'єкт, що виступає в ролі одночасно суб'єкта та спостерігача, без множинного успадкування.

```
#include <iostream>
#include <list>
#include <string>

class Observer;
class Subject;

class Observer {
public:
```

```
  virtual void update(Subject *subject) = 0;
};

class Subject {
public:
  virtual void notify() = 0;
  virtual void attach(Observer *observer) = 0;
  virtual void detach(Observer *observer) = 0;
};



class ObserverStorage {
  std::list<Observer *> list;
public:
  void attach(Observer *observer)
  {
        std::list<Observer *>::iterator it = std::find(list.begin(), list.end(),
observer);
        if (list.end() == it) {
              list.push_back(observer);
        }
  }

  void detach(Observer *observer)
  {
        std::list<Observer *>::iterator it = std::find(list.begin(), list.end(),
observer);
        if (list.end() == it) {
              list.remove(observer);
        }
  }
  class Iterator {
        ObserverStorage *storage;
        std::list<Observer *>::iterator it;
  public:
        Iterator(ObserverStorage *s) : storage(s) { it = storage->list.begin();  }
        Iterator(const Iterator & i) : storage(i.storage) { it = storage->list.begin();
}


        Iterator & operator =(const Iterator & i) {
              storage = i.storage;
              it = storage->list.begin();
              return *this;
```

```
        }


        Iterator & begin() { it = storage->list.begin(); return *this; }
        //prefix
        Iterator & operator ++() {  return next(); }
        //postfix
        Iterator & operator ++(int) { return next(); }
        operator Observer *() { return *it; }
        bool operator !() { return it != storage->list.end(); }

        Iterator & next() { ++it; return *this; }
  };

  Iterator createIterator() { return Iterator(this); }
  operator Iterator() { return createIterator(); }
};

class User: public Subject
{
  ObserverStorage observers;
  std::string email;
public:
  User(){}

  void attach(Observer *observer)
  {
        observers.attach(observer);
  }

  void detach(Observer *observer)
  {
        observers.detach(observer);
  }

  void changeEmail(std::string e)
  {
        email = e;
        notify();
  }

  void notify()
  {
        ObserverStorage::Iterator it = observers;
        for (; !it; ++it) {
```

```
                Observer * ob = it;
                ob->update(this);
            }
    }

    std::string getEmail() {
            return email;
    }
};

class UserObserver: public Observer
{
public:
    void update(Subject *subject)
    {
            std::cout << "Email changed: " << ((User *)subject)->getEmail() << std::endl;
    }

};

int main()
{
    UserObserver * observer = new UserObserver();

    User *user = new User();
    user->attach(observer);

    user->changeEmail("foo@bar.com");

    user->detach(observer);
    delete user;
    delete observer;
    return 0;
}
```

4.9. Патерн State.

Призначення. Дозволяє об'єкту варіювати свою поведінку залежно від внутрішнього стану. Ззовні складається враження, що змінився клас об'єкта.

Мотивація.

Розглянемо клас TCPConnection, з допомогою якого представлено мережне з'єднання. Об'єкт цього класу може бути в одному з кількох станів: Established (встановлено), Listening (прослуховування), Closed (закрито). Коли об'єкт TCPConnection отримує запити з інших об'єктів, залежно від поточного стану він

відповідає по-різному. Наприклад, відповідь на запит Open (відкрити) залежить від того, чи знаходиться з'єднання в стані Closed або Established. Паттерн стан описує, як об'єкт TCPConnection може поводитися по-різному, перебуваючи у різних станах.

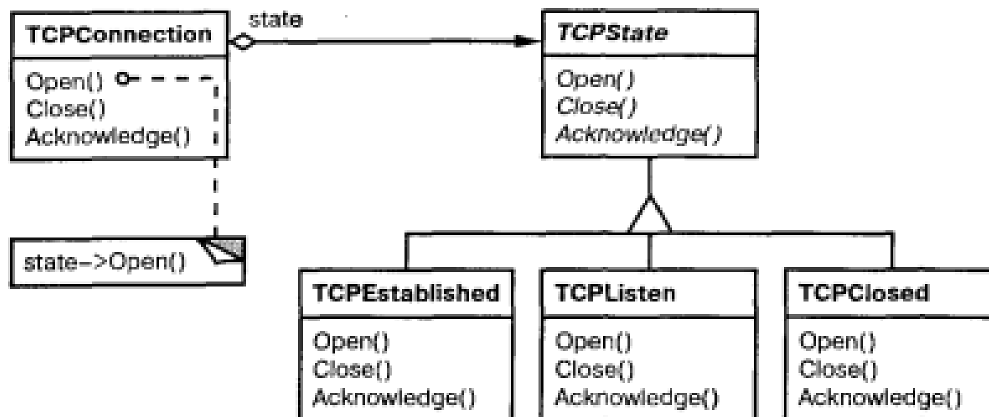

Рис. 4.13. Приклад взаємодії класів в патерні State

Основна ідея цього патерну полягає в тому, щоб запровадити абстрактний клас TCPState для представлення різних станів з'єднання. Цей клас оголошує інтерфейс, загальний для всіх класів, які описують різні робочі стани. У підкласах TCPState реалізовано поведінку, специфічну для конкретного стану. Наприклад, у класах TCPEstablished і TCPClosed реалізовано поведінку, характерну для станів Established і Closed відповідно.

Клас TCPConnection зберігає в себе об'єкт стану (примірник деякого підкласу TCPConnection), що представляє поточний стан з'єднання, і делегує всі залежні від стану запити цьому об'єкту. TCPConnection використовує свій екземпляр підкласу TCPState для виконання операцій, властивих лише цьому стану з'єднання.

При кожній зміні стану з'єднання TCPConnection змінює свій об'єкт-стан. Наприклад, коли встановлене з'єднання закривається, TCPConnection замінює екземпляр класу TCPEstablished на TCPClosed.

Застосування. Використовуйте патерн стан у таких випадках:

- коли поведінка об'єкта залежить від його стану та має змінюватися під час виконання;
- коли в коді операцій зустрічаються умовні оператори, що складаються з багатьох гілок, у яких вибір гілки залежить від стану. Зазвичай стан представлено перелічуваними константами. Часто та сама структура умовного оператора повторюється у кількох операціях. Паттерн стан пропонує помістити кожну гілку в окремий клас. Це дозволяє трактувати стан об'єкта як самостійний об'єкт, який може змінюватись незалежно від інших.

Структура.

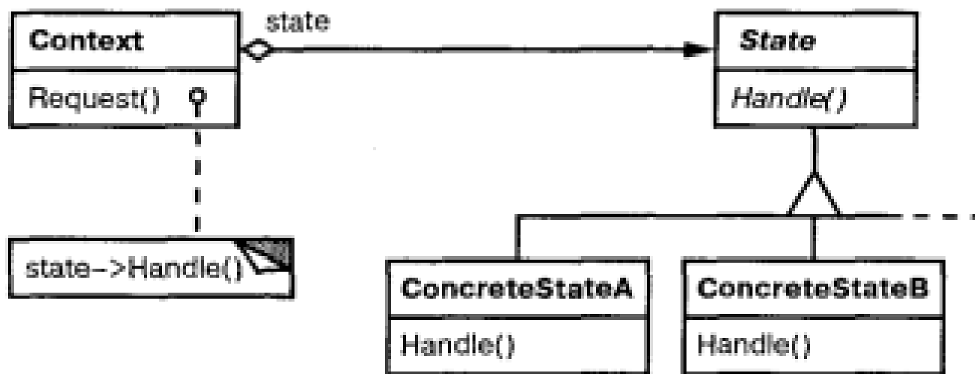

Рис. 4.14.Структура патерну State

Учасники:

- **Context** (TCPConnection) - контекст: визначає інтерфейс, що представляє інтерес для клієнтів;зберігає екземпляр підкласу ConcreteState, яким визначається поточний стан;
- **State** (TCPState) - стан: визначає інтерфейс для інкапсуляції поведінки, асоційованої з конкретним станом контексту Context;
- **підкласи ConcreteState** (TCPEstablished, TCPListen, TCPClosed) - конкретний стан: кожен підклас реалізує поведінку, асоційовану з деяким станом контексту Context.

Відносини:

- клас Context делегує залежні від стану запити поточному об'єкту ConcreteState;
- контекст може передати себе як аргумент об'єкту State, який оброблятиме запит. Це дає можливість об'єкту-стану за необхідності отримати доступ до контексту;
- Context – це основний інтерфейс для клієнтів. Клієнти можуть конфігурувати контекст об'єктами стану State. Одного разу конфігурувавши контекст, клієнти не повинні безпосередньо зв'язуватися з об'єктами стану;
- або Context або підкласи ConcreteState можуть вирішити, за яких умов та в якому порядку відбувається зміна станів.

Результати. Результати використання патерного стану:

- локалізує поведінку, що залежить від стану, і ділить її на частини, що відповідають станам. Паттерн стан поміщає всю поведінку, асоційовану з конкретним станом, в окремий об'єкт. Оскільки залежний від стану код цілком знаходиться в одному з підкласів класу State, то додавати нові стани та переходи можна просто шляхом створення нових підкласів.

Натомість можна було б використовувати дані-члени для визначення внутрішніх станів, тоді операції об'єкта Context перевіряли б ці дані. Але в такому разі схожі умовні оператори або оператори розгалуження були б розкидані по всьому коду класу Context. При цьому додавання нового стану потребувало б зміни кількох операцій, що ускладнило б супровід.

Паттерн стан дозволяє вирішити цю проблему, але водночас породжує іншу,

оскільки поведінка для різних станів виявляється розподіленою між кількома підкласами State. Це збільшує кількість класів. Звичайно, один клас компактніший, але якщо станів багато, то такий розподіл ефективніший, тому що в іншому випадку довелося б мати справу з громіздкими умовними операторами.

Наявність громіздких умовних операторів небажана, як і наявність довгих процедур. Вони занадто монолітні, тому модифікація і розширення коду стає проблемою. Паттерн стан пропонує найбільш зручний метод структурування залежить від стану коду. Логіка, що описує переходи між станами, більше не укладена в монолітні оператори if або switch, а розподілена між підкласами State. При інкапсуляції кожного переходу та дії у клас стан стає повноцінним об'єктом. Це покращує структуру коду та прояснює його призначення; '

- робить явними переходи між станами. Якщо об'єкт визначає свій поточний стан виключно в термінах внутрішніх даних, то переходи між станами немає явного уявлення; вони виявляються лише як присвоєння деяким змінним. Введення окремих об'єктів різних станів робить переходи більш явними. Крім того, об'єкти State можуть захистити контекст Context від неузгодженості внутрішніх змінних, оскільки переходи з погляду контексту – це атомарні дії. Для здійснення переходу треба змінити значення лише однієї змінної (об'єктної змінної State у класі Context), а не кількох;
- об'єкти стану можна розділяти. Якщо в об'єкті стану State відсутні змінні екземпляра, тобто стан, який він представляє, кодується виключно самим типом, то різні контексти можуть розділяти один і той же об'єкт State. Коли стани поділяються таким чином, вони є, по суті, пристосуванцями, у яких немає внутрішнього стану, а є тільки поведінка.

Реалізація. З патерном стан пов'язаний цілий ряд питань реалізації:

- що визначає переходи між станами. Патерн стан нічого не повідомляє про те, який учасник визначає критерій переходу між станами. Якщо критерії зафіксовані, їх можна реалізувати у класі Context. Однак у загальному випадку більш гнучкий і правильний підхід полягає в тому, щоб дозволити самим підклас класу State визначати наступний стан і момент переходу. Для цього до класу Context треба додати інтерфейс, що дозволяє об'єктам State встановити стан контексту.

Таку децентралізовану логіку переходів простіше модифікувати та розширювати – потрібно лише визначити нові підкласи State. Недолік децентралізації в тому, що кожен підклас State повинен «знати» ще хоча б про один підклас, що вносить реалізаційні залежності між підкласами;

таблична альтернатива. Можна використати таблицю для відображення вхідних даних на переходи між станами. З її допомогою можна визначити, який стан потрібно перейти при надходженні деяких вхідних даних. Фактично, цим замінюємо умовний код (чи віртуальні функції, якщо йдеться про патерн стан) пошуком у таблиці. Основна перевага таблиць - їх регулярності: зміни критеріїв переходу досить модифікувати лише дані, а не код. Але є й недоліки:

- ✓ пошук у таблиці часто менш ефективний, ніж виклик функції (віртуальної);
- ✓ подання логіки переходів в однорідному табличному форматі

робить критерії менш явними і, отже, більш складними для розуміння;

- ✓ зазвичай важко додати дії, якими супроводжуються переходи між станами. Табличний метод враховує стан і переходи між ними, але його необхідно доповнити, щоб при кожній зміні стану можна було виконувати довільні обчислення.

Головна різниця між кінцевими автоматами на базі таблиць і патерну стан можна сформулювати так: патерн стан моделює поведінку, що залежить від стану, а табличний метод акцентує увагу на визначенні переходів між станами;

- створення та знищення об'єктів стану. У процесі розробки зазвичай доводиться вибирати між:

- ✓ створенням об'єктів стану, коли в них виникає необхідність, і знищенням відразу після використання;
- ✓ створенням їх заздалегідь і назавжди.

Перший варіант кращий, коли заздалегідь невідомо, у які стани потраплятиме система, і контекст змінює стан порівняно рідко. При цьому ми не створюємо об'єктів, які ніколи не будуть використані, що є істотним, якщо в об'єктах стану зберігається багато інформації. Коли зміни стану відбуваються часто, тому не хотілося б знищувати об'єкти, що їх представляють (бо вони можуть дуже скоро знадобитися знову), слід скористатися другим підходом. Час створення об'єктів витрачається лише один раз - на початку, але на знищення - не витрачається зовсім. Щоправда, цей підхід може виявитися незручним, оскільки в контексті повинні зберігатися посилання на всі стани, які система теоретично може потрапити;

- використання динамічного успадкування. Варіювати поведінку на запит можна, змінюючи клас об'єкта під час виконання, але у більшості об'єктно-орієнтованих мов це не підтримується. Зі зміною цільового об'єкта делегування під час виконання по суті змінюється і структура графа успадкування. Такий механізм дозволяє об'єктам варіювати поведінку шляхом зміни свого класу.

```
#include <iostream>

class TCPOctetStream {

};

class TCPState;

class TCPConnection {
public:
  TCPConnection();
  ~TCPConnection();
  void ActiveOpen();
  void PassiveOpen();
  void Close();
  void Send();
```

```
  void Acknowledge();
  void Synchronize();
  void ProcessOctet(TCPOctetStream*) {
          std::cout << "process octet" << std::endl;
  }
private:
  friend class TCPState;
  void ChangeState(TCPState *);
  TCPState *_state;
};

class TCPState {
public:
  virtual void Transmit(TCPConnection *, TCPOctetStream*) {}
  virtual void ActiveOpen(TCPConnection *) {}
  virtual void PassiveOpen(TCPConnection *) {}
  virtual void Close(TCPConnection *) {}
  virtual void Send(TCPConnection *) {}
  virtual void Acknowledge(TCPConnection *) {}
  virtual void Synchronize(TCPConnection *) {}
protected:
  void ChangeState(TCPConnection *t, TCPState *s) { t->ChangeState(s); }
};

class TCPEstablished : public TCPState {
public:
  static TCPState *Instance() { return new TCPEstablished(); }
  virtual void Transmit(TCPConnection *, TCPOctetStream*);
  virtual void Close(TCPConnection *);
};

class TCPListen : public TCPState {
public:
  static TCPState *Instance() { return new TCPListen(); }
  virtual void Send(TCPConnection *);
};


class TCPClosed: public TCPState{
public:
  static TCPState *Instance() { return new TCPClosed(); }
  virtual void ActiveOpen(TCPConnection *t);
  virtual void PassiveOpen(TCPConnection *);
};
```

```
void TCPEstablished::Transmit(TCPConnection *t, TCPOctetStream*o) {
  std::cout << "Established Transmit" << std::endl;
  t->ProcessOctet(o);
}

void TCPEstablished::Close(TCPConnection * t) {
  std::cout << "Established Close" << std::endl;
  ChangeState(t, TCPListen::Instance());
}


void TCPClosed::ActiveOpen(TCPConnection *t) {
  std::cout << "Closed ActiveOpen" << std::endl;
  ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen(TCPConnection *t) {
  std::cout << "Closed PassiveOpen" << std::endl;
  ChangeState(t, TCPListen::Instance());
}

void TCPListen::Send(TCPConnection *t) {
  std::cout << "Listen Send" <<std::endl;
  ChangeState(t, TCPEstablished::Instance());
}

TCPConnection::TCPConnection() {
  _state = TCPClosed::Instance();
}

TCPConnection::~TCPConnection() {
  delete _state;
}

void TCPConnection::ChangeState(TCPState *s) {
  delete _state;
  _state = s;
}

void TCPConnection::ActiveOpen() {
  _state->ActiveOpen(this);
}
void TCPConnection::PassiveOpen() {
  _state->PassiveOpen(this);
}
```

```
void TCPConnection::Close() {
  _state->Close(this);
}
void TCPConnection::Send() {
  _state->Send(this);
}
void TCPConnection::Acknowledge() {
  _state->Acknowledge(this);
}

void TCPConnection::Synchronize() {
  _state->Synchronize(this);
}

int main()
{
  TCPConnection connection;
  connection.PassiveOpen();
  connection.Send();
  connection.Close();
  return 0;
}
```

4.10. Патерн Strategy.

Призначення. Визначає сімейство алгоритмів, інкапсулює кожен із них і робить їх взаємозамінними. Стратегія дозволяє змінювати алгоритми незалежно від клієнтів, які користуються ними.

Мотивація. Існує багато алгоритмів для розбиття тексту на рядки. Жорстко «зашивати» всі подібні алгоритми до класів, які їх потребують, небажано з кількох причин:

- клієнт, якому потрібний алгоритм розбиття на рядки, ускладнюється при включенні до нього відповідного коду. Таким чином, клієнти стають громіздкішими, а супроводжувати їх важче, особливо якщо потрібно підтримати відразу кілька алгоритмів;
- залежно від обставин варто застосовувати той чи інший алгоритм. Не хотілося б підтримувати кілька алгоритмів розбиття на рядки, якщо ми не користуватимемося ними;
- якщо розбиття на рядки – невід'ємна частина клієнта, то завдання додавання нових та модифікації існуючих алгоритмів ускладнюється.

Всіх цих проблем можна уникнути, якщо визначити класи, що інкапсулюють різні алгоритми розбиття на рядки. Інкапсульований таким чином алгоритм називається стратегією.

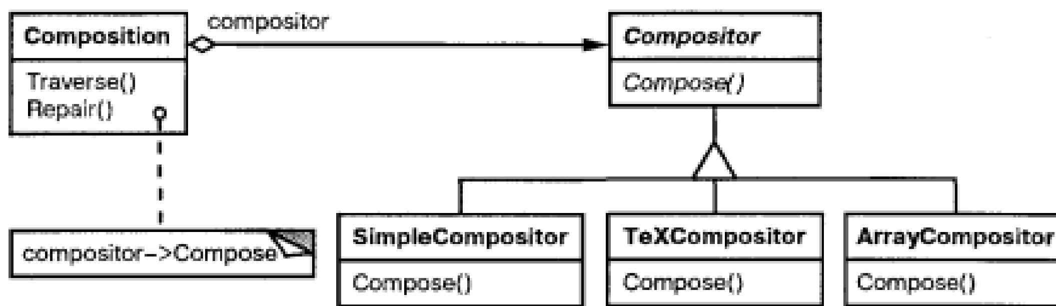

Рис. 4.15. Приклад взаємодії класів у патерні Strategy

Припустимо, що клас Composition відповідає за розбиття на рядки тексту, що відображається у вікні програми перегляду, та його своєчасне оновлення. Стратегії розбиття на рядки визначаються не в класі Composition, а в підкласах абстрактного класу Compositor. Це можуть бути, наприклад, такі стратегії:

- SimpleCompositor реалізує просту стратегію, що виділяє по одному рядку за раз;
- TeXCompositor реалізує алгоритм пошуку точок розбиття на рядки, прийнятий у редакторі TeX. Ця стратегія намагається виконати глобальну оптимізацію розбиття на рядки, розглядаючи одразу цілий параграф;
- ArrayCompositor реалізує стратегію розміщення переходів на новий рядок таким чином, що в кожному рядку виявляється те саме число елементів. Це корисно, наприклад, при строковому відображенні набору піктограм.

Об'єкт Composition зберігає посилання на об'єкт с. Щоразу, коли об'єкту Composition потрібно переформатувати текст, він делегує цей обов'язок своєму об'єкту Compositor. Клієнт вказує, який об'єкт Compositor слід використовувати, параметризуючи їм об'єкт Composition.

Застосування. Використовуйте патерн стратегія, коли:

- є багато споріднених класів, які відрізняються лише поведінкою. Стратегія дозволяє налаштувати клас, задавши одне з можливих поведінок;
- Вам потрібно мати декілька різних варіантів алгоритму. Наприклад, можна визначити два варіанти алгоритму, один із яких вимагає більше часу, а інший – більше пам'яті. Стратегії дозволяється застосовувати, коли варіанти алгоритмів реалізовані як ієрархії класів;
- алгоритм містить дані, про які клієнт не повинен «знати». Використовуйте патерн стратегію патерн, щоб не розкривати складні, специфічні для алгоритму структури даних;
- у класі визначено багато поведінки, що представлено розгалуженими умовними операторами. У цьому випадку простіше перенести код із гілок в окремі класи стратегій.

Учасники:

- **Strategy** (Compositor) - стратегія: оголошує загальний всім підтримуваних алгоритмів інтерфейс. Клас Context користується цим інтерфейсом для виклику конкретного алгоритму, визначеного в класі ConcreteStrategy;
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor) -

конкретна стратегія: реалізує алгоритм, який використовує інтерфейс, оголошений у класі Strategy;

- **Context** (Composition) - контекст: конфігурується об'єктом класу ConcreteStrategy; зберігає посилання на об'єкт класу Strategy; може визначати інтерфейс, який дозволяє об'єкту ConcreteStrategy отримати доступ до даних контексту.

Структура

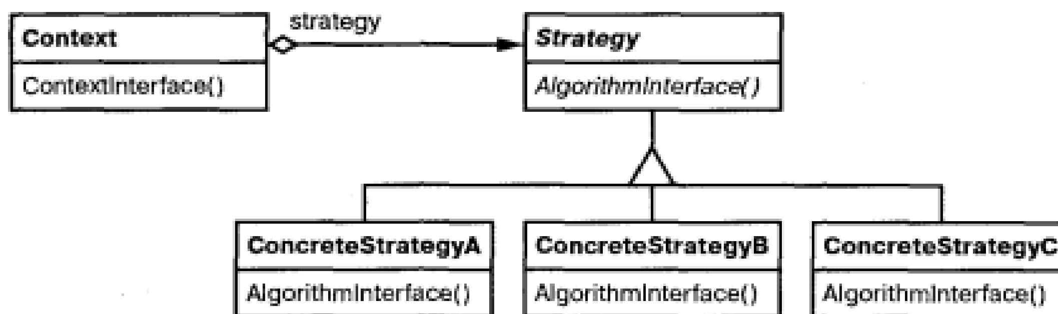

Рис. 4.16. Структура патерну Strategy

Відносини:

- класи Strategy та Context взаємодіють для реалізації обраного алгоритму. Контекст може передати стратегії всі необхідні алгоритму дані на момент його виклику. Натомість контекст може дозволити звертатися до своїх операцій у потрібні моменти, передавши посилання на себе операціям класу Strategy;
- контекст переадресує запити своїх клієнтів об'єкту-стратегії. Зазвичай клієнт створює об'єкт ConcreteStrategy і передає його контексту, після чого клієнт «спілкується» виключно з контекстом. Часто у розпорядженні клієнта знаходиться кілька класів ConcreteStrategy, які він може вибирати.

Результати. У патерну стратегія є такі переваги та недоліки:

- сімейства родинних алгоритмів. Ієрархія класів Strategy визначає сімейство алгоритмів чи поведінки, які можна повторно використовувати у різних контекстах. Спадкування дозволяє вичленувати загальну для всіх алгоритмів функціональність;
- альтернатива породженню підкласів. Спадкування підтримує різномаїття алгоритмів чи поведінки. Можна безпосередньо породити від Context підкласи з різними поведінками. Але при цьому поведінка жорстко зашивається в клас Context. Ось чому реалізації алгоритму та контексту поєднуються, що ускладнює розуміння, супровід та розширення контексту. Крім того, замінити алгоритм динамічно вже не вдасться. В результаті ви отримаєте безліч споріднених класів, що відрізняються лише алгоритмом чи поведінкою. Інкапсуляції алгоритму в окремий клас Strategy дозволяють змінювати його незалежно від контексту;
- за допомогою стратегій можна позбутися умовних операторів. Завдяки патерну стратегія вдається відмовитися від умовних операторів під час вибору потрібної поведінки. Коли різні поведінки містяться в одному класі, важко вибрати потрібне без застосування умовних операторів. Інкапсуляція кожної

поведінки в окремий клас Strategy вирішує цю проблему;

- вибір реалізації. Стратегії можуть пропонувати різні реалізації однієї й тієї ж поведінки. Клієнт має право вибирати відповідну стратегію залежно від своїх вимог до швидкодії та пам'яті;

- клієнти повинні «знати» про різні стратегії. Потенційний недолік цього патерну в тому, що для вибору стратегії клієнт повинен розуміти, чим відрізняються різні стратегії. Тому, напевно, доведеться розкрити клієнту деякі особливості реалізації. Звідси випливає, що стратегію варто застосовувати лише тоді, коли відмінності в поведінці мають значення для клієнта;

- обмін інформацією між стратегією та контекстом. Інтерфейс класу Strategy поділяється всіма підкласами ConcreteStrategy - неважливо, складна або тривіальна їх реалізація. Тому цілком імовірно, що деякі стратегії не будуть користуватися всією інформацією, що передається, особливо прості. Це означає, що в окремих випадках контекст створить і проініціалізує параметри, які нікому не потрібні. Якщо виникне проблема, між класами Strategy і Context доведеться встановити тісніший зв'язок;

- збільшення кількості об'єктів. Застосування стратегій збільшує кількість об'єктів у додатку. Іноді ці витрати можна скоротити, якщо реалізувати стратегії як об'єкти без стану, які можуть розділятися кількома контекстами. Залишковий стан зберігається у самому контексті та передається при кожному зверненні до об'єкта-стратегії. Стратегії, що розділяються, не повинні зберігати стан між викликами.

Реалізація. Розглянемо такі питання реалізації:

- визначення інтерфейсів класів Strategy і Context. Інтерфейси класів Strategy і Context можуть забезпечити об'єкту класу ConcreteStrategy ефективний доступ до будь-яких даних контексту, і навпаки.

Наприклад, Context передає дані як параметрів операціям класу Strategy. Це розриває тісний зв'язок між контекстом та стратегією. При цьому не виключено, що контекст передаватиме дані, які стратегії не потрібні.

Інший метод - передати контекст як аргумент, у такому разі стратегія запитуватиме у нього дані, або, наприклад, зберегти посилання на свій контекст, так що передавати взагалі нічого не доведеться. І в тому, і в інших випадках стратегія може вимагати тільки ту інформацію, яка реально потрібна. Але тоді в контексті має бути визначений більш розвинений інтерфейс до своїх даних, що дещо посилює зв'язаність класів Strategy і Context.

Який підхід кращий, залежить від конкретного алгоритму та вимог, які він пред'являє до даних;

- стратегії як параметри шаблону. У C++ конфігурування класу стратегією можна використовувати шаблони. Цей спосіб хороший, тільки якщо стратегія визначається на етапі компіляції та її не потрібно міняти під час виконання. Тоді клас, що конфігурується (наприклад, Context) визначається у вигляді шаблону, для якого клас Strategy є параметром:

При використанні шаблонів відпадає потреба в абстрактному класі для визначення інтерфейсу Strategy. Крім того, передача стратегії у вигляді параметра шаблону дозволяє статично пов'язати стратегію з контекстом,

внаслідок чого підвищується ефективність програми;

- об'єкти-стратегії можна не задавати. Клас Context дозволяється спростити, якщо йому відсутність будь-якої стратегії є нормою. Перш ніж звертатися до об'єкта Strategy, об'єкт Context перевіряє наявність стратегії. Якщо так, то робота продовжується як завжди, інакше контекст реалізує певну поведінку за умовчанням. Перевага такого підходу в тому, що клієнтам взагалі не потрібно мати справу зі стратегіями, якщо їх влаштовує стандартна поведінка.

Розглянемо стратегію на прикладі сортування масиву.

```
#include <iostream>
#include <vector>

class ComparatorStrategy
{
public:
  virtual int compare(double a, double b) = 0;
  virtual ~ComparatorStrategy() { }
};

class LesserCompratаor : public ComparatorStrategy
{
public:
  int compare(double a, double b) { return  (a > b); }
};
class GreaterCompratаor : public ComparatorStrategy
{
public:
  int compare(double a, double b) { return (a < b);  }
};


class SortingAlgo
{
  ComparatorStrategy * m_pComparator;
  void swap(double &x, double &y)
  {
          double tmp = x;
          x = y;
          y = tmp;
  }
public:
  SortingAlgo()
  {
          m_pComparator = new LesserCompratаor();
  }
```

```
	void sort(std::vector<double> & arr, ComparatorStrategy * pComparator = NULL)
	{
		if (pComparator == NULL)
			pComparator = m_pComparator;
		bool isSwapped = true;
		int x = 0;
		while (isSwapped)
		{
			isSwapped = false;
			x++;
			for (size_t i = 0; i < arr.size() - x; i++)
			{
				if (pComparator->compare(arr[i], arr[i + 1]))
				{
					swap(arr[i], arr[i + 1]);
					isSwapped = true;
				}
			}
		}
	}
};


int main()
{
	std::vector<double> arr = { 1.0,5.0,2.0,4.0,3.0 };
	SortingAlgo obj;
	ComparatorStrategy * pComp = new LesserComprataor();
	obj.sort(arr, pComp);
	for (int var = 0; var < 5; ++var) {
		std::cout << arr[var] << " ";
	}
	std::cout << std::endl;
	delete pComp;
	pComp = NULL;
	pComp = new GreaterComprataor();
	obj.sort(arr, pComp);
	for (int var = 0; var < 5; ++var) {
		std::cout << arr[var] << " ";
	}
	std::cout << std::endl;
	delete pComp;
	pComp = NULL;
```

```
    delete pComp;
    pComp = NULL;
    obj.sort(arr);
    for (int var = 0; var < 5; ++var) {
        std::cout << arr[var] << " ";
    }
    std::cout << std::endl;
    delete pComp;
    pComp = NULL;
    return 0;
}
```

4.11. Патерн Template Method.

Призначення. Шаблонний метод визначає основу алгоритму та дозволяє підкласам перевизначити деякі кроки алгоритму, не змінюючи його структуру загалом.

Мотивація.

Розглянемо каркас програми, в якій є класи Application та Document. Клас Application відповідає за відкриття існуючих документів, що зберігаються у зовнішньому форматі, наприклад у вигляді файлу. Об'єкт класу Document надає інформацію документа після його читання з файлу.

Програми, побудовані з урахуванням цього каркасу, можуть породжувати підкласи від класів Application і Document, відповідальні конкретним потребам. Наприклад, графічний редактор визначить підкласи DrawApplication та DrawDocument, а електронна таблиця - підкласи SpreadsheetApplication та SpreadsheetDocument.

В абстрактному класі Application визначено алгоритм відкриття та зчитування документа в операції OpenDocument. Операція OpenDocument визначає всі кроки відкриття документа. Вона перевіряє, чи можна відкрити документ, створює об'єкт класу Document, додає його до набору документів та зчитує документ із файлу.

Операцію виду OpenDocument ми називатимемо шаблонним методом, що описує алгоритм у термінах абстрактних операцій, які заміщені в підкласах для отримання потрібної поведінки. Підкласи класу Application додатково виконують перевірку можливості відкриття (CanOpenDocument) і створення документа (DoCreateDocument). Підкласи класу Document зчитують документ (DoRead). Шаблонний метод визначає також операцію, яка дозволяє підкласам Application отримати інформацію про те, що документ ось-ось буде відкритий (AboutToOpenDocument). Визначаючи деякі кроки алгоритму за допомогою абстрактних операцій, шаблонний метод фіксує їхню послідовність, але дозволяє реалізувати їх у підкласах класів Application та Document.

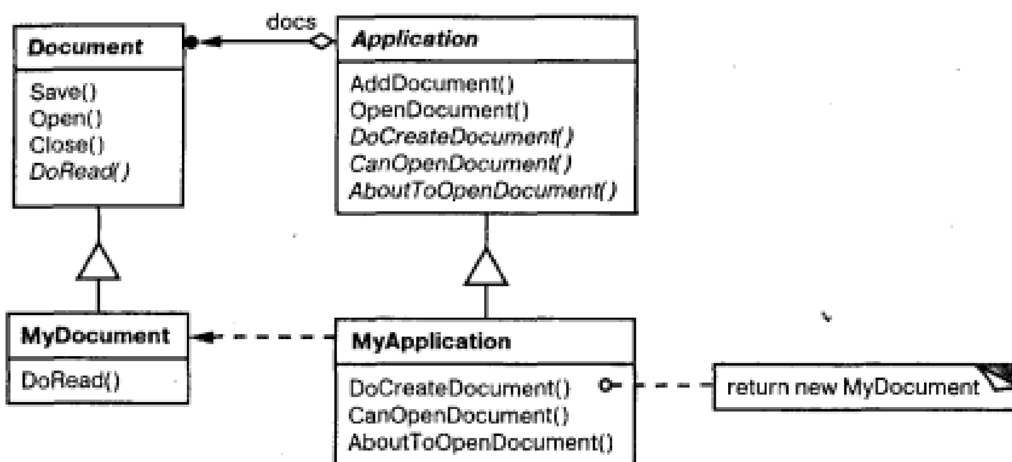

Рис. 4.17. Приклад взаємодії класів в патерні Template Method

Застосування. Паттерн шаблонний метод слід використовувати:

- щоб одноразово використовувати інваріантні частини алгоритму, залишаючи реалізацію змінної поведінки на розсуд підкласів;
- коли потрібно вичленувати та локалізувати в одному класі поведінку, спільну для всіх підкласів, щоб уникнути дублювання коду. Це гарний приклад техніки «винесення за дужки з метою узагальнення». Спочатку ідентифікуються відмінності у існуючому коді, а потім вони виносяться в окремі операції. У кінцевому результаті фрагменти коду, що розрізняються, замінюються шаблонним методом, з якого викликаються нові операції;
- для керування розширеннями підкласів. Можна визначити шаблонний метод так, що він викликатиме операції-зачіпки (hooks) у певних точках, дозволивши цим розширення тільки в цих точках.

Структура.

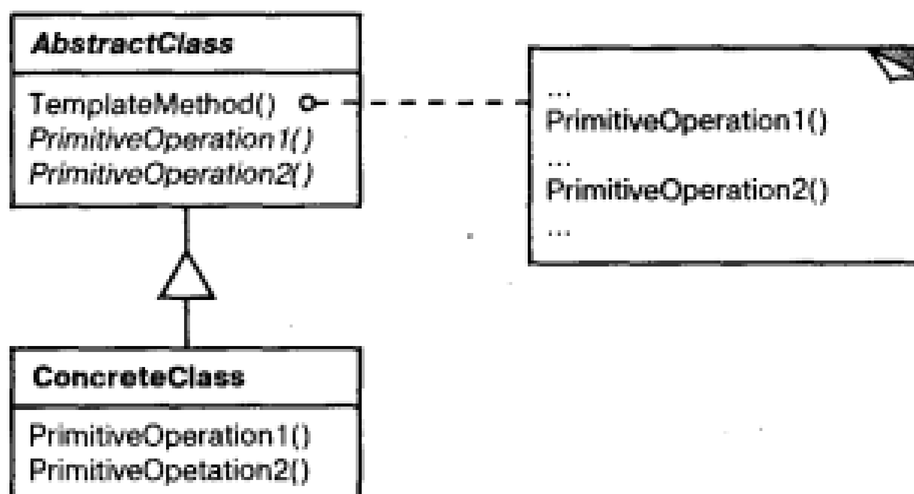

Рис. 4.18. Структура патерну Template Method

Учасники:

- **AbstractClass** (Application) - абстрактний клас: визначає абстрактні примітивні операції, які заміщаються у конкретних підкласах для реалізації

кроків алгоритму; реалізує шаблонний метод, що визначає скелет алгоритму. Шаблонний метод викликає примітивні операції, а також операції, визначені в класі AbstractClass або в інших об'єктах;

- **ConcreteClass** (MyApplication) - конкретний клас: реалізує примітивні операції, що виконують кроки алгоритму у спосіб, що залежить від підкласу.

Відносини: ConcreteClass передбачає, що інваріантні кроки алгоритму будуть виконані в AbstractClass.

Результати.

Шаблонні методи - один із фундаментальних прийомів повторного використання коду. Вони особливо важливі у бібліотеках класів, оскільки дають можливість винести загальну поведінку до бібліотечних класів.

Шаблонні методи призводять до інвертованої структури коду. Це означає, що батьківський клас викликає операції підкласу, а не навпаки.

Шаблонні методи викликають операції наступних видів:

- конкретні операції (або з класу ConcreteClass, або з класів клієнта);
- конкретні операції з класу AbstractClass (тобто операції, корисні всім підкласам);
- примітивні операції (тобто абстрактні операції);
- фабричні методи;
- операції-зачіпки, що реалізують стандартну поведінку, яка може бути розширена в підкласах. Часто така операція за умовчанням нічого не робить.

Важливо, щоб у шаблонному методі чітко розрізнялися операції-зачіпки (які можна замінювати) та абстрактні операції (які потрібно замінювати). Щоб повторно використовувати абстрактний клас із максимальною ефективністю, автори підкласів повинні розуміти, які операції призначені для заміщення.

На жаль, дуже легко забути необхідність викликати успадковану операцію. У нас є можливість трансформувати таку операцію на шаблонний метод з метою надати батькові контроль над тим, як підкласи розширюють його. Ідея в тому, щоб викликати операцію-зачіпку із шаблонного методу в батьківському класі. Тоді підкласи зможуть перевизначити саме цю операцію:

Реалізаця. Варто розповісти про три аспекти, що стосуються реалізації:

- використання контролю доступу до C++. Примітивні операції, які викликає шаблонний метод, можна оголосити захищеними членами. Тоді гарантується, що викликати їх зможе лише сам шаблонний метод. Примітивні операції, які обов'язково потрібно заміщати, оголошуються як суто віртуальні функції. Сам шаблонний метод заміщати не треба, тому його можна зробити невіртуальною функцією-членом;
- скорочення числа примітивних операцій. Важливою метою при проектуванні шаблонних методів є усіляке скорочення числа примітивних операцій, які мають бути заміщені у підкласах. Чим більше операцій потрібно заміщати, тим стомливішим стає програмування клієнта;
- угоду про імена. Виділити операції, які необхідно замістити, можна шляхом додавання до імен певного префікса.

Наступний приклад днмонструє реалізацію цього патерну на прикладі сортування масиву за різними критеріями.

```
#include <iostream>
#include <ctime>
class AbstractSort
{
    // Shell sort
public:
    void sort(int v[], int n)
    {
        for (int g = n / 2; g > 0; g /= 2)
                for (int i = g; i < n; i++)
                        for (int j = i - g; j >= 0; j -= g)
                                if (needSwap(v[j], v[j + g]))
                                        doSwap(v[j], v[j + g]);
    }
private:
    virtual int needSwap(int, int) = 0;
    void doSwap(int &a, int &b)
    {
        int t = a;
        a = b;
        b = t;
    }
};

class SortUp : public AbstractSort
{
    /* virtual */
    int needSwap(int a, int b)
    {
        return (a > b);
    }
};
class SortDown : public AbstractSort
{
    /* virtual */
    int needSwap(int a, int b)
    {
        return (a < b);
    }
};

int main()
{
    const int NUM = 10;
    int array[NUM];
```

```
    srand((unsigned)time(0));
    for (int i = 0; i < NUM; i++)
    {
        array[i] = rand() % 10 + 1;
        std::cout << array[i] << ' ';
    }
    std::cout << std::endl;

    AbstractSort *sortObjects[] =
    {
     new SortUp, new SortDown
    };
    sortObjects[0]->sort(array, NUM);
    for (int u = 0; u < NUM; u++)
        std::cout << array[u] << ' ';
    std::cout << std::endl;

    sortObjects[1]->sort(array, NUM);
    for (int d = 0; d < NUM; d++)
        std::cout << array[d] << ' ';
    std::cout << std::endl;
    return 0;
}
```

4.12. Патерн Visitor.

Призначення. Описує операцію, що виконується з кожним об'єктом із певної структури. Паттерн відвідувач дозволяє визначити нову операцію, не змінюючи класи цих об'єктів.

Мотивація.

Розглянемо компілятор, який представляє програму у вигляді абстрактного синтаксичного дерева. Над такими деревами він повинен виконувати операції статичного семантичного аналізу, наприклад перевіряти, що всі змінні визначені. Ще йому потрібно генерувати код. Аналогічно можна було б визначити операції контролю типів, оптимізації коду, аналізу потоку виконання, перевірки того, що кожній змінній було надано конкретне значення перед першим використанням, і т.д. Більш того, абстрактні синтаксичні дерева могли б служити для гарного друку програми, реструктурування коду та обчислення різних метрик програми.

У більшості таких операцій вузли дерева, що представляють оператори присвоєння, слід розглядати інакше, ніж вузли, які мають змінні та арифметичні вирази. Тому один клас буде створено для операторів присвоювання, інший – для доступу до змінних, третій – для арифметичних виразів тощо. Набір класів вузлів, звичайно, залежить від мови, що компілюється, але не дуже сильно.

Проблема тут у тому, що якщо розкидати всі операції за класами різних вузлів, то вийде система, яку важко зрозуміти, супроводжувати та змінювати.

Навряд чи хтось розбереться у програмі, якщо код, відповідальний за перевірку типів, буде перемішаний з кодом, що реалізує гарний друк чи аналіз потоку виконання. Крім того, додавання будь-якої нової операції вимагатиме перекомпіляції всіх класів. Оптимальний варіант - наявність можливості додавати операції окремо і відсутність залежності класів вузлів від операцій, що застосовуються до них.

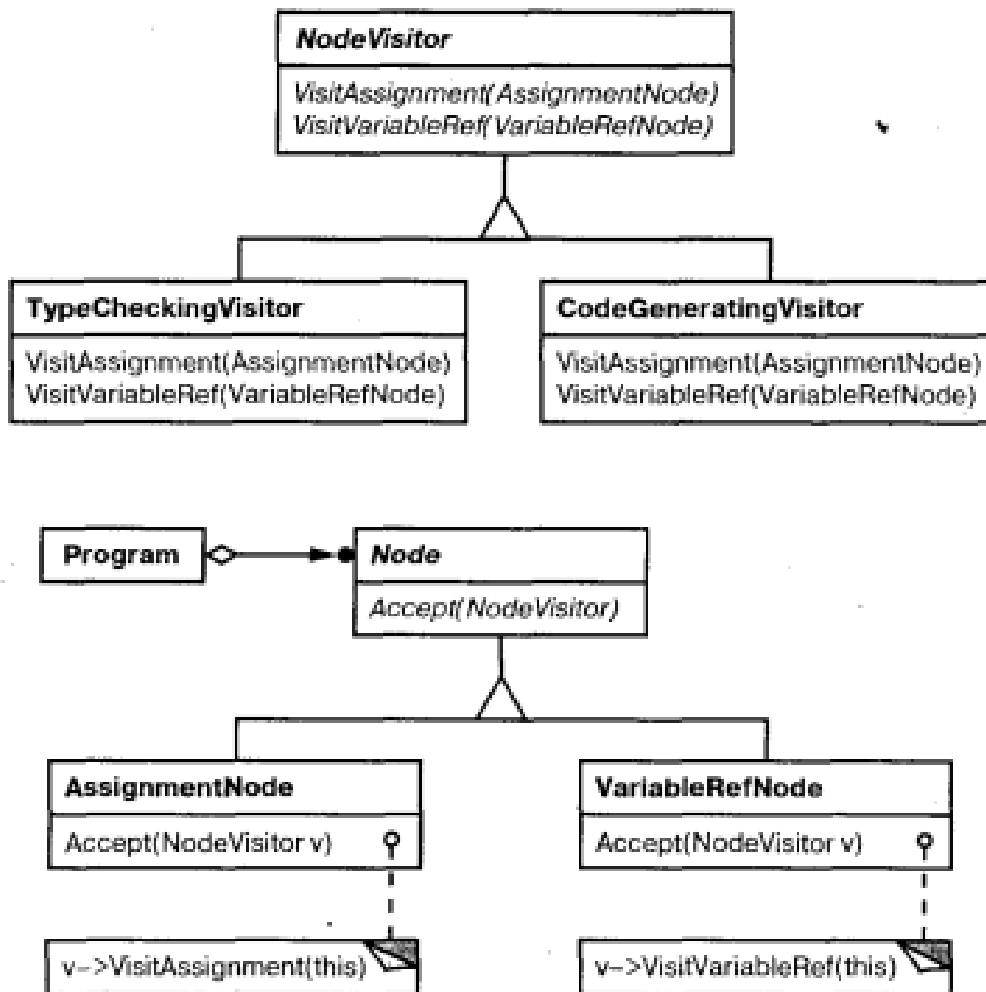

Рис. 4.19. Приклад взаємодії класів в патерні Visitor

І того, й іншого можна домогтися, якщо помістити взаємопов'язані операції з кожного класу в окремий об'єкт, який називають відвідувачем, і передавати його елементам абстрактного синтаксичного дерева в міру обходу. «Приймаючи» відвідувача, елемент надсилає йому запит, у якому міститься, зокрема, клас елемента. Крім того, у запиті є у вигляді аргументу і сам елемент. Відвідувачу в цій ситуації належить виконати операцію над елементом, ту саму, яка, напевно, перебувала б у класі елемента.

Наприклад, компілятор, який не використовує відвідувачів, міг би перевірити тип процедури, викликавши операцію TypeCheck для її абстрактного синтаксичного дерева. Кожен вузол дерева мав реалізувати операцію TypeCheck шляхом рекурсивного виклику її для своїх компонентів. Якщо ж компілятор перевіряє тип процедури за допомогою відвідувачів, йому достатньо створити

об'єкт класу TypeCheckingVisitor і викликати для дерева операцію Accept, передавши їй цей об'єкт як аргумент. Кожен вузол повинен був реалізувати Accept шляхом звернення до відвідувача: вузол, що відповідає оператору присвоєння, викликає операцію відвідувача VisitAssigment, а вузол, що посилається на змінну, - операцію VisitVariableReference. Те, що раніше було операцією TypeCheck в класі AssigmentNode, стало операцією VisitAssigment у класі TypeCheckingVisitor.

Щоб відвідувачі могли займатися не лише перевіркою типів, нам потрібен абстрактний клас NodeVisitor, який є батьком для всіх відвідувачів синтаксичного дерева. Додаток, якому потрібно обчислювати метрики програми, визначив би нові підкласи NodeVisitor, так що нам не довелося б додавати код, що залежить від програми, до класів вузлів. Паттерн відвідувач інкапсулює операції, що виконуються на кожній фазі компіляції, у класі Visitor, асоційованому з цією фазою.

Застосовуючи патерн відвідувач, ви визначаєте дві ієрархії класів: одну для елементів, над якими виконується операція (ієрархія Node), а іншу - для відвідувачів, що описують операції, що виконуються над елементами (ієрархія NodeVisitor). Нова операція створюється шляхом додавання підкласу до ієрархії класів відвідувачів. Доки граматика мови залишається постійною (тобто не додаються нові підкласи Node), нову функціональність можна отримати шляхом визначення нових підкласів NodeVisitor.

Застосування. Використовуйте патерн відвідувач, коли:

- у структурі присутні об'єкти багатьох класів з різними інтерфейсами, і ви хочете виконувати над ними операції, що залежать від конкретних класів;
- над об'єктами, що входять до складу структури, треба виконувати різноманітні, не пов'язані між собою операції, і ви не хочете засмічувати класи такими операціями. Відвідувач дозволяє об'єднати споріднені операції, помістивши в один клас. Якщо структура об'єктів є спільною для кількох додатків, то патерн відвідувач дозволить у кожну програму включити операції, що тільки відносяться до нього;
- класи, які встановлюють структуру об'єктів, змінюються рідко, але нові операції над цією структурою часто додаються. При зміні класів, представлених у структурі, потрібно буде перевизначити інтерфейси всіх відвідувачів, а це може спричинити труднощі. Тому якщо класи змінюються досить часто, то, ймовірно, краще визначити операції у них.

Учасники:

– **Visitor** (NodeVisitor) - відвідувач: оголошує операцію Visit; для кожного класу ConcreteElement у структурі об'єктів. Ім'я та сигнатура цієї операції ідентифікують клас, який посилає відвідувачеві запит Visit;. Це дозволяє відвідувачеві визначити, елемент якого конкретного класу він відвідує. Володіючи такою інформацією, відвідувач може звертатися до елемента безпосередньо через його інтерфейс;

– **ConcreteVisitor** (TypeCheckingVisitor г) - конкретний відвідувач: реалізує всі операції, оголошені у класі Visitor. Кожна операція реалізує фрагмент алгоритму, визначеного для відповідного класуоб'єкта у структурі.

Клас ConcreteVisitor надає контекст для цього алгоритму та зберігає його локальний стан. Часто у цьому стані акумулюються результати, отримані у процесі обходу структури;

– **Element** (Node) - елемент: визначає операцію Accept, яка приймає відвідувача як аргумент;

– **ConcreteElement** (AssigmentNode, VariableRefNode) - конкретний елемент: реалізує операцію Accept, яка приймає відвідувача як аргумент;

– **ObjectStructure** (Program) - структура об'єктів: може перерахувати свої елементи; може надати відвідувачу високорівневий інтерфейс відвідування своїх елементів; може бути як складовим об'єктом, так і колекцією, наприклад, списком або множиною.

Структура.

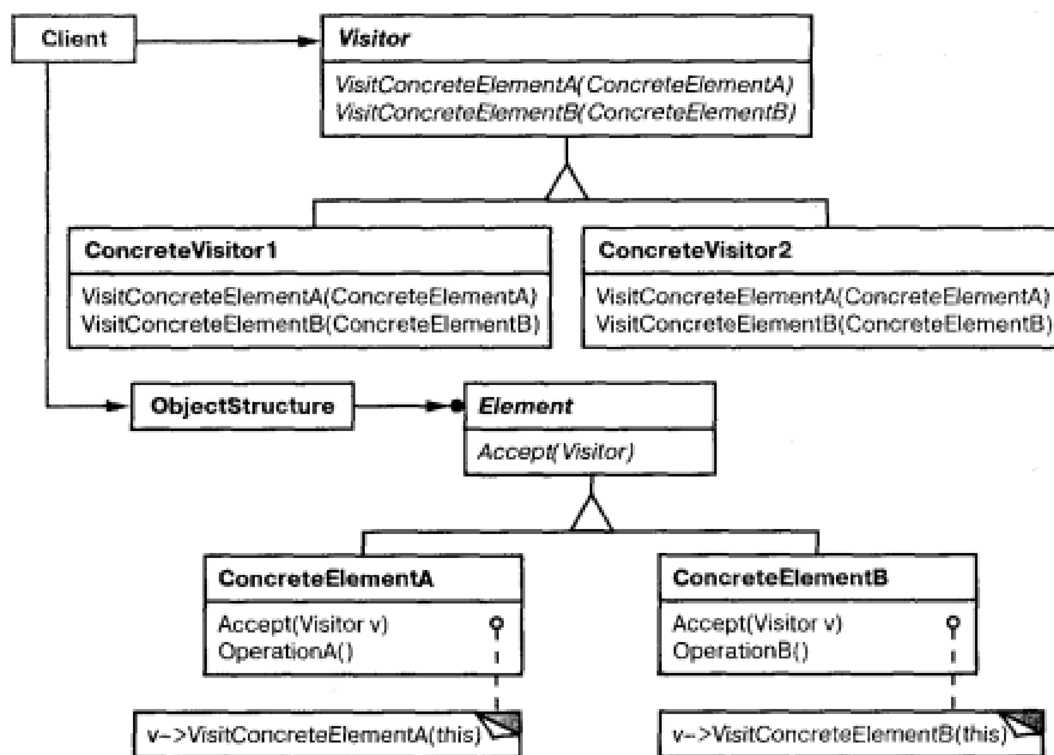

Рис. 4.20. Структура патерну Visitor

Відносини:

- клієнт, який використовує патерн відвідувач, повинен створити об'єкт класу ConcreteVisitor, а потім обійти всю структуру, відвідавши кожен її елемент.
- при відвідуванні елемента останній викликає операцію відвідувача, яка відповідає своєму класу. Елемент передає в цю операцію себе як аргумент, щоб відвідувач міг за необхідності отримати доступ до його стану.

Результати. Деякі переваги та недоліки патерну відвідувач:

- спрощує додавання нових операцій. За допомогою відвідувачів легко додавати операції, що залежать від компонентів складних об'єктів. Для визначення нової операції над структурою об'єктів досить просто запровадити нового відвідувача. Навпаки, якщо функціональність розподілена за кількома

класами, то для визначення нової операції доведеться змінити кожен клас;

- об'єднує споріднені операції та відсікає ті, що не мають до них відношення. Споріднена поведінка не розноситься по всіх класах, присутніх у структурі об'єктів, вона локалізована у відвідувачі. Не пов'язані одна з одною функції розподіляються за окремими підкласами класу Visitor. Це сприяє спрощенню як класів, які визначають елементи, так і алгоритмів, інкапсульованих у відвідувачах. Усі структури даних, що належать до алгоритму, можна приховати у відвідувачі;
- додавання нових класів ConcreteElement ускладнено. Паттерн відвідувач ускладнює додавання нових підкласів класу Element. Кожен новий конкретний елемент вимагає оголошення нової абстрактної операції у класі Visitor, яку потрібно реалізувати у кожному з існуючих класів ConcreteVisitor. Іноді більшість конкретних відвідувачів можуть успадкувати за замовчуванням операцію, що надається класом Visitor, що швидше виняток, ніж правило.

Тому при вирішенні питання про те, чи варто використовувати патерн відвідувач, потрібно перш за все подивитися, що змінюватиметься частіше: алгоритм, що застосовується до об'єктів структури, або класи об'єктів, що становлять цю структуру. Цілком імовірно, що супроводжувати ієрархію класів Visitor буде нелегко, якщо нові класи ConcreteElement додаються часто. У разі простіше визначити операції у класах, представлених у структурі. Якщо ж ієрархія класів Element: стабільна, але постійно розширюється набір операцій чи модифікуються алгоритми, то патерн відвідувач допоможе краще керувати такими змінами;

- відвідування різних ієрархій класів. Ітератор може відвідувати об'єкти структури в міру її обходу, викликаючи операції об'єктів. Але ітератор неспроможний працювати зі структурами, які складаються з об'єктів різних типів. У відвідувача таких обмежень немає. Йому дозволено відвідувати об'єкти, які не мають загального батьківського класу. До інтерфейсу класу Visitor можна додати операції для об'єктів будь-якого типу;
- акумулювання стану. Відвідувачі можуть акумулювати інформацію про стан відвідування об'єктів структури. Якщо не використовувати цей патерн, стан доведеться передавати як додаткові аргументи операцій, що виконують обхід, або зберігати в глобальних змінних;
- порушення інкапсуляції. Застосування відвідувачів має на увазі, що у класу ConcreteElement досить розвинений інтерфейс для того, щоб відвідувачі могли впоратися зі своєю роботою. Тому при використанні цього патерну доводиться надавати відкриті операції для доступу до внутрішнього стану елементів, що ставить під загрозу інкапсуляцію.

Реалізація.

При вирішенні питання про застосування патерну відвідувач часто виникає два спірні моменти:

- подвійна диспетчеризація. За своєю суттю патерн відвідувач дозволяє, не змінюючи класи, додавати нові операції. Досягає він цього за допомогою прийому, що називається подвійною диспетчеризацією.

Для визначення того, яка операція виконуватиме запит, у мовах з одинарною

диспетчеризацією необхідні ім'я запиту та тип одержувача. Таким чином, виконувана операція визначається одночасно видом запиту та типом одержувача. Поняття «подвійна диспетчеризація» означає, що операція, що виконується, залежить від виду запиту і типів двох одержувачів. Accept - це операція з подвійною диспетчеризацією. Її семантика залежить від типів двох об'єктів: Visitor та Element. Подвійна диспетчеризація дозволяє відвідувачеві вимагати різні операції для кожного класу елемента. Тому виникає необхідність у патерні відвідувач: операція, що виконується, залежить і від типу відвідувача, і від типу відвідуваного елемента. Замість статичної прив'язки операцій до інтерфейсу класу Element ми можемо консолідувати ці операції в класі Visitor і використовувати Accept для прив'язки їх під час виконання. Розширення інтерфейсу класу Element зводиться до визначення нового підкласу Visitor, а не модифікації багатьох підкласів Element;

- який учасник відповідає за обхід структури. Відвідувач має обійти кожен елемент структури об'єктів. Питання у тому, як туди потрапити. Відповідальність за обхід можна покласти на структуру об'єктів, на відвідувача або на окремий об'єкт-ітератор. Найчастіше структура об'єктів відповідає за обхід. Колекція просто оминає всі свої елементи, викликаючи для кожного операцію Accept. Складовий об'єкт зазвичай обходить себе, «примушуючи» операцію Accept відвідати нащадків поточного елемента і рекурсивно викликати Accept для кожного з них.

Інше рішення – скористатися ітератором для відвідування елементів. У C++ можна застосувати внутрішній чи зовнішній ітератор, залежно від цього доступно й ефективніше. Оскільки внутрішні ітератори реалізуються самої структурою об'єктів, робота з ними багато в чому нагадує попереднє рішення, коли за обхід відповідає структура. Основна відмінність полягає в тому, що внутрішній ітератор не призводить до подвійної диспетчеризації: він викликає операцію відвідувача з елементом як аргумент, а не операцію елемента з відвідувачем як аргумент. Проте використовувати паттерн відвідувач із внутрішнім ітератором легко у разі, коли операція відвідувача викликає операцію елемента без рекурсії.

Можна навіть помістити алгоритм обходу у відвідувач, хоча закінчиться це дублюванням коду обходу в кожному класі ConcreteVisitor для кожного агрегату ConcreteElement. Основна причина такого рішення – необхідність реалізувати особливо складну стратегію обходу, яка залежить від результатів операцій над об'єктами структури.

```
#include <iostream>
#include <string>
class User;
class Group;
class RoleVisitor
{
public:
    virtual void visitUser(User *) = 0;
    virtual void visitGroup(Group *) = 0;
```

```
};

class Role
{
public:
    virtual void accept(RoleVisitor *visitor) = 0;
};


class User:public Role
{
    std::string name;
public:
    User(std::string n): name(n) { }

    std::string getName() { return name; }

    void accept(RoleVisitor *visitor) {
        visitor->visitUser(this);
    }
};

class Group :public Role
{
    std::string name;
public:
    Group(std::string n) : name(n) {  }

    std::string getName() { return name; }

    void accept(RoleVisitor *visitor){
        visitor->visitGroup(this);
    }
};

class RecordingVisitor: public RoleVisitor
{
    /**
     * @var Role[]
     */

    void visitGroup(Group *role){
        std::cout << "Role: " << role->getName() << std::endl;
    }
```

```
    void visitUser(User * user) {
        std::cout << "User: " << user->getName() << std::endl;
    }

};

int main()
{
    RecordingVisitor visitor;
    User user("Dominik");
    Group group("Admin");
    user.accept(&visitor);
    group.accept(&visitor);
    return 0;
}
```

Контрольні питання

1. Які переваги та недоліки патерну Chain of Repository?
2. Які переваги та недоліки патерну Command?
3. Які переваги та недоліки патерну Interpreter?
4. Які переваги та недоліки патерну Iterator?
5. Які переваги та недоліки патерну Mediator?
6. Які переваги та недоліки патерну Memento?
7. Які переваги та недоліки патерну Observer?
8. Які переваги та недоліки патерну State?
9. Які переваги та недоліки патерну Strategy?
10. Які переваги та недоліки патерну Template Method?
11. Які переваги та недоліки патерну Visitor?

РОЗДІЛ 5
ЛАБОРАТОРНІ РОБОТИ

5.1. Лабораторна робота №1. Дослідження архітектурних моделей та методологій розробки ПЗ для обраної предметної області.

Тема: Дослідження архітектурних моделей та методологій розробки програмного забезпечення для обраної предметної області.

Мета роботи:

- дослідити базові моделі архітектур програмного забезпечення;
- дослідити методології та підходи до розробки ПЗ;
- ознайомитися з принципом ООП - SOLID;
- обрати базову для реалізації архітектуру обраної предметної області. Описати підхід (з точки методології розробки ПЗ) до розробки програмного продукту згідно обраної предметної області. Визначити переваги та недоліки застосування моделі архітектури;
- ознайомитися з середовищем проектування IBM Rational Software Architect.
- проаналізувати обрану предметну область. Визначити вимоги.

Завдання роботи:

- Опрацюйте теоретичний матеріал.
- Ознайомтесь з основними архітектурами реалізації ПЗ.
- Ознайомтесь з методологіями розробки ПЗ.
- Розгляньте теоретичний матеріал щодо принципів побудови Хорошої архітектури SOLID.
- Оберіть тематику власної розробки. Основні принципи моделі архітектури. Виділіть основні функції проектованої інформаційної системи та зовнішні сутності, з якими система взаємодіє.

Теоретичні відомості:

1. 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.

Архітектура - це принцип організації компонентів усередині системи: їх кількість, якість, інтерфейси і протоколи взаємодії. Від архітектури залежать ціна та витрати на підтримку і розробку нових можливостей, 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. Вибір або створення

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

2. Вибір архітектури ПЗ — це етап проектування, що виконується після етапу аналізу і формулювання вимог. Задача такого проектування — перетворення вимог до системи у вимоги до ПЗ і побудова на їх основі архітектури системи. Побудова архітектури системи здійснюється шляхом визначення цілей системи, її вхідних і вихідних даних, декомпозиції системи на підсистеми, компоненти або модулі та розроблення її загальної структури. Проектування архітектури системи може проводитися різними методами (стандартизованим, об'єктно-орієнтованим, компонентним і ін.), кожний з яких пропонує свій шлях побудови архітектури, а саме, визначення концептуальної, об'єктної й інших моделей за допомогою відповідних конструктивних елементів (блок-схем, графів, структурних діаграм тощо).

3. Методологія - це система принципів, а також сукупність ідей, понять, методів, способів і засобів, які визначають стиль розробки програмного забезпечення. Це - реалізація стандарту. Самі стандарти лише вказують на те, що повинно бути, залишаючи свободу вибору і адаптації. Методології являють собою ядро теорії управління розробкою програмного забезпечення. До існуючої класифікації, залежно від використовуваної моделі життєвого циклу (каскадні і ітераційні методології) додалася більш загальна класифікація на прогнозовані і адаптивні методології. Базовими підходами вважають:

- Rational Unified Process (RUP);
- Microsoft Solutions Framework (MSF);
- Методологія аналізу вимог і гнучкий процес розробки ПЗ ICONIX, що ґрунтується на варіантах використання (пропонований фірмою ICONIX Software Engineering, Inc.);
- Швидка розробка додатків — Rapid Application Development (RAD);
- Метод розробки динамічних систем — Dynamic Systems Development Method (DSDM).

4. Принципи "хорошої" архітектури SOLID:

- Принцип Єдиної Відповідальності (Single Responsibility Principle) Клас повинен мати тільки одну причину для зміни
- Принцип відкриття-закриття (Open Close Principle або OCP) Програмні сутності такі як класи, модулі та функції повинні бути відкриті для розширення, але закриті для змін.
- Принцип Заміщення Ліскоу (Liskov's Substitution Principle) Похідні типи повинні бути здатні повністю заміщатися їх базовими типами.
- Принцип Відділення Інтерфейсу (Interface Segregation Principle) Клієнти не повинні бути залежними від інтерфейсів, які вони не використовують. 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.

Зразки предметних областей для проектування та розробки:

1. Розробка системи управління товарами.
2. Розробка системи управління реєстрацією покупців.
3. Розробка системи управління замовленнями.
4. Розробка системи споживацького кошика.
5. Розробка системи каталогу товарів.
6. Розробка системи анкетування клієнтів.
7. Розробка системи поштової служби.
8. Розробка системи диспетчера списків розсилання.
9. Розробка системи підтримки Web-форумів.
10. Розробка системи управління доставкою товарів.
11. Розробка системи управління надходженням комерційних пропозицій постачальників.
12. Розробка системи управління рухом та запасами продукції на складах.
13. Розробка системи інтерактивного банківського обслуговування.
14. Розробка системи підтримки електронних платежів.
15. Розробка інформаційної системи для автоматизації маркетингових досліджень у діяльності комерційного підприємства.
16. Розробка інформаційної системи для вибору каналів збуту продукції.
17. Розробка інформаційної системи для вибору постачальників ресурсів.
18. Розробка інформаційної системи для дослідження попиту на готову продукцію.
19. Розробка інформаційної системи для оцінки конкурентного стану ринку.
20. Розробка інформаційної системи для планування потреби в матеріальних ресурсах.
21. Розробка інформаційної системи для формування портфелю замовлень збуту готової продукції.
22. Розробка інформаційної системи для аналізу ефективності реклами.
23. Розробка інформаційної системи управління рекламною діяльністю.
24. Розробка інформаційної системи управління ціноутворенням.
25. Розробка інформаційної системи управління рухом та запасами продукції на складах.
26. Розробка інформаційної системи для обліку відвантаження та реалізації продукції.
27. Розробка інформаційної системи з обліку розрахунків з постачальниками.
28. Розробка інформаційної системи з обліку та аналізу виконання договорів.
29. Інформаційна система для організації та проведення державних закупівель.
30. Система моніторингу міського транспорту
31. Поштовий клієнт (для синхронізації поштових акаунтів).

32. Диспетчерський центр оперативного вирішення скарг мешканців району.

33. Служба обліку спожитої електроенергії в навчальних корпусах університету.

34. Інформаційно довідкова система туристичних пам’яток України.

35. Автоматизована інформаційна система реєстрації та обліку замовлень торгівельного підприємства.

36. Система розподілу та контролю завдаль на підприємстві.

37. Інтернет-аукціон.

38. Електронна платіжна система.

39. Ресурс для бронювання та продажу квитків на азалізничні перевезення.

40. Система документообігу підприємства з продаж будівельних матеріалів.

41. Ресурс для замовлення послуг таксі.

42. Центр прийому замовлень на телерекламу міського телебачення.

43. Служба реєстрації викликів швидкої допомоги.

44. Служба реєстрації хворих поліклініки.

45. Інформаційна система електронного документообігу на підприємстві

46. Інформаційна система обліку продукції на складі.

47. Автоматизована система обліку нарахування заробітної плати.

48. Система автоматизації бронювання та продажу квитків на проїзд автобусним транспортом.

49. Система автоматизації бронювання та оренди місць на автопарковці.

50. Автоматизована система формування та оцінювання тестових завдань для навчального закладу.

51. Система автоматизації роботи страхової агенції.

52. Автоматизована система ресторану.

53. Система планування та контролю робіт на підприємстві.

54. Логістична система планування маршруту.

5.2. Лабораторна робота №2. Розробка програмного забезпечення з використанням патерну проектування “Abstract Factory” та “Factory Method”.

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування “Abstract Factory” та “Factory Method”.

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - “Abstract Factory” та “Factory Method”.

Теоретичні відомості:

Абстрактна фабрика - гнучка і динамічна архітектурна модель проектування моделі предметної області системи (МПО), що включає взаємозв’язані ієрархії на основі інтерфейсів (абстрактних класів) для створення множини взаємозв’язаних об’єктів, не специфікуючи їх конкретних класів. Спочатку проектуються абстрактні класи (базові класи ієрархій), що описують інтерфейси для створення компонентів системи, а потім створюються похідні конкретні класи, що поліморфно реалізують ці інтерфейси.

Переваги:

1. Внесення швидких змін у місця коду ієрархій, строго визначені архітектором (підсистеми/ класи, їх зміна, розширення їх переліку), не порушуючи цілісності всієї системи.

2. Суттєве зменшення кількості класів та їх описів, що веде до оптимізації коду, підвищення рівня читабельності щодо доопрацювання (відхід від важкої спадщини попередників).

3. Забезпечення формалізованих уніфікованих запитів користувача до системи ч/з інтерфейси (відокремлення описів класів від реалізацій, інкапсуляція і захист від несанкціонованого доступу).

4. Гнучкість і незалежність розроблюваних компонент ПЗ, їх взаємозамінність, повторне використання коду.

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Абстрактна фабрика (Abstract Factory) та фабричний метод (Factory Method).

2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.

3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах Абстрактна фабрика (Abstract Factory)) та фабричний метод (Factory Method). Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.

4. Підготувати розробку до захисту.

5.3. Лабораторна робота №3. Розробка програмного забезпечення з використанням патерну проектування “Decorator” та “Flyweight”.

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування “Decorator” та “Flyweight”.

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - “Decorator” та “Flyweight”.

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Декоратор (Decorator) та Пристосуванець (Flyweight).
2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.
3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах Декоратор (Decorator) та Пристосуванець (Flyweight). Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.
4. Підготувати розробку до захисту.

Теоретичні відомості:

Структурні патерни проектування відповідають за побудову зручних в підтримці ієрархій класів та визначають взаємозалежності між класами і об'єктами, дозволяючи їм працювати спільно. При цьому можуть використовуватися такі механізми компонування системи на основі класів і об'єктів:

Наслідування, коли клас визначає інтерфейс, а підкласи - реалізацію. Структури на основі наслідування виходять статичними.

Композиція, коли структури будуються шляхом об'єднання об'єктів деяких класів. Композиція дозволяє отримувати структури, які можна змінювати під час виконання.

Декоратор — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки». Спадкування — це перше, що приходить в голову багатьом програмістам, коли потрібно розширити якунебудь чинну поведінку. Проте механізм спадкування має кілька прикрих проблем.

Він статичний. Ви не можете змінити поведінку об’єкта, який вже існує. Для цього необхідно створити новий об’єкт, вибравши інший підклас.

Він не дозволяє наслідувати поведінку декількох класів одночасно. Тому доведеться створювати безліч підкласів-комбінацій, щоб досягти поєднання поведінки.

Одним зі способів, що дозволяє обійти ці проблеми — є заміна спадкування агрегацією або композицією . Це той випадок, коли один об’єкт утримує інший і делегує йому роботу, замість того, щоб самому успадкувати його поведінку. Саме на цьому принципі побудовано патерн Декоратор. Декоратор має альтернативну назву — обгортка. Вона більш вдало описує суть патерна: ви

розміщуєте цільовий об'єкт у іншому об'єкті-обгортці, який запускає базову поведінку об'єкта, а потім додає до результату щось своє.

Обидва об'єкти мають загальний інтерфейс, тому для користувача немає жодної різниці, з чим працювати — з чистим чи загорнутим об'єктом. Ви можете використовувати кілька різних обгорток одночасно — результат буде мати об'єднану поведінку всіх обгорток. Застосовуючи Декоратор, ви не змінюєте початковий клас і не створюєте дочірніх класів.

Застосування:

Якщо вам потрібно додавати об'єктам нові обов'язки «на льоту», непомітно для коду, який їх використовує. Об'єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об'єкти мають однаковий інтерфейс, тому клієнтам не важливо, з чим працювати — зі звичайним об'єктом чи з загорнутим.

Якщо не можна розширити обов'язки об'єкта за допомогою спадкування. У багатьох мовах програмування є ключове слово final, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

Легковаговик — це структурний патерн проектування, що дає змогу вмістити більшу кількість об'єктів у відведеній оперативній пам'яті. Легковаговик заощаджує пам'ять, розподіляючи спільний стан об'єктів між собою, замість зберігання однакових даних у кожному об'єкті. Патерн Легковаговик пропонує не зберігати зовнішній стан у класі, а передавати його до тих чи інших методів через параметри. Таким чином, одні і ті самі об'єкти можна буде повторно використовувати в різних контекстах. Головна ж перевага в тому, що тепер знадобиться набагато менше об'єктів, адже вони тепер відрізнятимуться тільки внутрішнім станом, а він не має так багато варіацій. Патерн Легковаговик слід застосовувати при дотриманні всіх наступних умов:

Коли додаток використовує велику кількість одноманітних об'єктів, через що відбувається виділення великої кількості пам'яті.

Коли частина стану об'єкта, яке є змінним, можна винести за. Винесення зовнішнього стану дозволяє замінити безліч об'єктів невеликою групою загальних поділюваних об'єктів.

Застосування:

Якщо не вистачає оперативної пам'яті для підтримки всіх потрібних об'єктів. Ефективність патерна Легковаговик багато в чому залежить від того, як і де він використовується. Застосовуйте цей патерн у випадках, коли виконано всі перераховані умови: у програмі використовується велика кількість об'єктів; через це високі витрати оперативної пам'яті; більшу частину стану об'єктів можна винести за межі їхніх класів; великі групи об'єктів можна замінити невеликою кількістю об'єктів, що розділяються, оскільки зовнішній стан винесено.

5.4. Лабораторна робота №4. Розробка програмного забезпечення з використанням патерну проектування “Proxy” та “Facade”.

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування “Proxy” та “Facade”.

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - “Proxy” та “Facade”.

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Замісник (Proxy) та Фасад (Facade).

2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.

3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах “Proxy” та “Facade”. Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.

4. Підготувати розробку до захисту.

Теоретичні відомості:

Замісник — це структурний патерн проектування, що дає змогу підставляти замість реальних об’єктів спеціальні об’єкти-замінники. Ці об’єкти перехоплюють виклики до оригінального об’єкта, дозволяючи зробити щось до чи після передачі виклику оригіналові. Це - об’єкт, який виступає прошарком між клієнтом та реальним сервісним об’єктом. Замісник отримує виклики від клієнта, виконує свою функцію (контроль доступу, кешування, зміна запиту та інше), а потім передає виклик сервісному об’єктові.

Патерн Замісник пропонує створити новий клас-дублер, який має той самий інтерфейс, що й оригінальний службовий об’єкт. При отриманні запиту від клієнта об’єктзамісник сам би створював примірник службового об’єкта та переадресовував би йому всю реальну роботу.

Замісник має той самий інтерфейс, що і реальний об’єкт, тому для клієнта немає різниці — працювати з реальним об’єктом безпосередньо, чи за допомогою замісника.

Застосування:

Лінива ініціалізація (віртуальний проксі). Коли у вас є важкий об’єкт, який завантажує дані з файлової системи або бази даних. Замість того, щоб завантажувати дані відразу після старту програми, можна заощадити ресурси й створити об’єкт тоді, коли він дійсно знадобиться.

Захист доступу (захищаючий проксі). Коли в програмі є різні типи користувачів, і вам хочеться захистити об’єкт від неавторизованого доступу. Наприклад, якщо ваші об’єкти — це важлива частина операційної системи, а користувачі — сторонні програми (корисні чи шкідливі). Проксі може перевіряти доступ під час кожного виклику та передавати виконання службовому об’єкту, якщо доступ дозволено.

Локальний запуск сервісу (віддалений проксі). Коли справжній сервісний

об'єкт знаходиться на віддаленому сервері. У цьому випадку замісник транслює запити клієнта у виклики через мережу по протоколу, який є зрозумілим віддаленому сервісу.

Логування запитів (логуючий проксі). Коли потрібно зберігати історію звернень до сервісного об'єкта. Замісник може зберігати історію звернення клієнта до сервісного об'єкта.

Кешування об'єктів («розумне» посилання). Коли потрібно кешувати результати запитів клієнтів і керувати їхнім життєвим циклом. Замісник може підраховувати кількість посилань на сервісний об'єкт, які були віддані клієнту та залишаються активними. Коли всі посилання звільняться, можна буде звільнити і сам сервісний об'єкт (наприклад, закрити підключення до бази даних). Крім того, Замісник може відстежувати, чи клієнт не змінював сервісний об'єкт. Це дозволить повторно використовувати об'єкти й суттєво заощаджувати ресурси, особливо якщо мова йде про великі «ненажерливі» сервіси.

Фасад — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті можливості, які потрібні клієнтові, і приховує все інше.

Фасад корисний у тому випадку, якщо ви використовуєте якусь складну бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина.

Вашому коду доводиться працювати з великою кількістю об'єктів певної складної бібліотеки чи фреймворка. Ви повинні самостійно ініціалізувати ці об'єкти, стежити за правильним порядком залежностей тощо. В результаті бізнес-логіка ваших класів тісно переплітається з деталями реалізації сторонніх класів. Такий код досить складно розуміти та підтримувати.

Застосування:

Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми. Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Таку підсистему простіше використовувати повторно, налаштовуючи 23 її кожен раз під конкретні потреби, але, разом з тим, використовувати таку підсистему без налаштовування важче. Фасад пропонує певний вид системи за замовчуванням, який влаштовує більшість клієнтів.

Якщо ви хочете розкласти підсистему на окремі рівні. Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади. Наприклад, візьмемо ту ж саму складну систему конвертації відео. Ви хочете розбити її на окремі шари для роботи з аудіо й відео. Можна спробувати створити фасад для кожної з цих частин і примусити класи аудіо та відео обробки спілкуватися один з одним через ці фасади, а не безпосередньо.

5.5. Лабораторна робота №5. Розробка програмного забезпечення з використанням патерну проектування “Observer” та “Visitor”.

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування “Observer” та “Visitor”.

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - “Observer” та “Visitor”.

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Спостерігач (Observer) та Відвідувач (Visitor).

2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.

3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах “Observer” та “Visitor”. Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.

4. Підготувати розробку до захисту.

Теоретичні відомості:

Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах. Список підписників складається динамічно, об’єкти можуть як підписуватися на певні події, так і відписуватися від них прямо під час виконання програми.

Для додавання до програми нових підписників не потрібно змінювати класи видавців, допоки вони працюють із підписниками через загальний інтерфейс.

Застосування:

Якщо після зміни стану одного об’єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об’єкти мають відреагувати. Патерн Спостерігач надає змогу будь-якому об’єкту з інтерфейсом підписника зареєструватися для отримання сповіщень про події, що трапляються в об’єктах-видавцях.

Якщо одні об’єкти мають спостерігати за іншими, але тільки у визначених випадках. Видавці ведуть динамічні списки. Усі спостерігачі можуть підписуватися або відписуватися від отримання сповіщень безпосередньо під час виконання програми

Відвідувач — це поведінковий патерн проектування, що дає змогу додавати до програми нові операції, не змінюючи класи об’єктів, над якими ці операції можуть виконуватися. Патерн Відвідувач пропонує розмістити нову поведінку в окремому класі, замість того, щоб множити її відразу в декількох класах. Об’єкти, з якими повинна бути пов’язана поведінка, не виконуватимуть її самостійно. Замість цього ви будете передавати ці об’єкти до методів відвідувача.

Застосування:

Якщо вам потрібно виконати якусь операцію над усіма елементами складної

структури об’єктів, наприклад, деревом. Відвідувач дозволяє застосовувати одну і ту саму операцію до об’єктів різних класів.

Якщо над об’єктами складної структури об’єктів потрібно виконувати деякі не пов’язані між собою операції, але ви не хочете «засмічувати» класи такими операціями. Відвідувач дозволяє витягти споріднені операції з класів, що складають структуру об’єктів, помістивши їх до одного класу-відвідувача. Якщо структура об’єктів використовується в декількох програмах, то патерн дозволить кожній програмі мати тільки потрібні в ній операції.

Якщо нова поведінка має сенс тільки для деяких класів з існуючої ієрархії. Відвідувач дозволяє визначити поведінку тільки для цих класів, залишивши її порожньою для всіх інших.

5.6. Лабораторна робота №6. Розробка програмного забезпечення з використанням патерну проектування "Strategy" та "Chain of Responsibility".

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування "Strategy " та " Chain of Responsibility ".

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - "Strategy " та " Chain of Responsibility ".

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Стратегія (Strategy) та Ланцюг відповідальності (Chain of Responsibility).
2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.
3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах "Strategy " та " Chain of Responsibility ". Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.
4. Підготувати розробку до захисту.

Теоретичні відомості:

Стратегія — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми. Патерн Стратегія пропонує визначити сімейство схожих алгоритмів, які часто змінюються або розширюються, й винести їх до власних класів, які називають стратегіями.

Важливо, щоб всі стратегії мали єдиний інтерфейс. Використовуючи цей інтерфейс, контекст буде незалежним від конкретних класів стратегій. З іншого боку, ви зможете змінювати та додавати нові види алгоритмів, не чіпаючи код контексту

Застосування:

Якщо вам потрібно використовувати різні варіації якого-небудь алгоритму всередині одного об'єкта. Стратегія дозволяє варіювати поведінку об'єкта під час виконання програми, підставляючи до нього різні об'єкти-поведінки (наприклад, що відрізняються балансом швидкості та споживання ресурсів).

Якщо у вас є безліч схожих класів, які відрізняються лише деякою поведінкою. Стратегія дозволяє відокремити поведінку, що відрізняється, у власну ієрархію класів, а потім звести початкові класи до одного, налаштовуючи його поведінку стратегіями.

Якщо ви не хочете оголювати деталі реалізації алгоритмів для інших класів. Стратегія дозволяє ізолювати код, дані й залежності алгоритмів від інших об'єктів, приховавши ці деталі всередині класів-стратегій.

Якщо різні варіації алгоритмів реалізовано у вигляді розлогого умовного оператора. Кожна гілка такого оператора є варіацією алгоритму. Стратегія

розміщує кожну лапу такого оператора до окремого класу-стратегії. Потім контекст отримує певний об'єктстратегію від клієнта й делегує йому роботу. Якщо раптом знадобиться змінити алгоритм, до контексту можна подати іншу стратегію.

Ланцюг відповідальності (Chain of Responsibility) — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен 29 наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком. Як і багато інших поведінкових патернів, ланцюжок обов'язків базується на тому, щоб перетворити окремі поведінки на об'єкти. У нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

Застосування:

Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться. За допомогою Ланцюжка обов'язків ви можете зв'язати потенційних обробників в один ланцюг і по отриманню запита по черзі питати кожного з них, чи не хоче він обробити даний запит.

Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку. Ланцюжок обов'язків дозволяє запускати обробників один за одним у тій послідовності, в якій вони стоять в ланцюзі.

Якщо набір об'єктів, здатних обробити запит, повинен задаватися динамічно. У будь-який момент ви можете втрутитися в існуючий ланцюжок і перевизначити зв'язки так, щоби прибрати або додати нову ланку.

5.7. Лабораторна робота №7. Розробка програмного забезпечення з використанням патерну проектування “State” та “Composite”.

Тема: Побудова програмного забезпечення на Visual C++ з використанням архітектурного шаблону проектування “State” та “Composite”.

Мета роботи: Реалізувати на Visual C++ архітектурну модель (АМ) проектування ПЗ - “State” та “Composite”.

Завдання роботи:

1. Опрацюйте теоретичний матеріал. Ознайомтесь з специфікою архітектурного шаблону проектування Стан (State) та Компонувальник (Composite).
2. Визначити вимоги до ПЗ згідно обраної предметної області, проаналізувати прецеденти, способи застосування та архітектурну модель майбутньої системи.
3. Реалізувати у середовищі Visual Studio ієрархію класів, яка б базувалася на шаблонах “State” та “Composite”. Описати та прокоментувати використання ієрархії класів, інтерфейси, розмежування атрибутів та методів конкретних класів.
4. Підготувати розробку до захисту.

Теоретичні відомості:

Стан — це поведінковий патерн проектування, що дає змогу об’єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об’єкта. Патерн Стан неможливо розглядати у відриві від концепції машини станів, також відомої як стейт-машина або кінцевий автомат. Основна ідея в тому, що програма може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Набір цих станів, а також переходів між ними, визначений наперед та кінцевий. Перебуваючи в різних станах, програма може по-різному реагувати на одні і ті самі події, що відбуваються з нею.

Стани найчастіше реалізують за допомогою множини умовних операторів, if або switch, які перевіряють поточний стан об’єкта та виконують відповідну поведінку. Патерн Стан пропонує створити окремі класи для кожного стану, в якому може перебувати контекстний об’єкт, а потім винести туди поведінки, що відповідають цим станам.

Застосування:

Якщо у вас є об’єкт, поведінка якого кардинально змінюється в залежності від внутрішнього стану, причому типів станів багато, а їхній код часто змінюється. Патерн пропонує виділити в окремі класи всі поля й методи, пов’язані з визначеним станом. Початковий об’єкт буде постійно посилатися на один з об’єктів-станів, делегуючи йому частину своєї роботи. Для зміни стану до контексту достатньо буде підставляти інший об’єкт-стан.

Якщо код класу містить безліч великих, схожих один на одного умовних операторів, які вибирають поведінки в залежності від поточних значень полів класу. Патерн пропонує перемістити кожну гілку такого умовного оператора до власного класу. Сюди ж можна поселити й усі поля, пов’язані з цим станом.

Якщо ви свідомо використовуєте табличну машину станів, побудовану на умовних операторах, але змушені миритися з дублюванням коду для схожих станів та переходів. Патерн Стан дозволяє реалізувати ієрархічну машину станів, що базується на наслідуванні. Ви можете успадкувати схожі стани від одного батьківського класу та винести туди весь дублюючий код.

Компонувальник (Composite) — це структурний патерн проектування, що дає змогу згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об’єкт. Патерн Компонувальник має сенс тільки в тих випадках, коли основна модель вашої програми може бути структурована у вигляді дерева.

Застосування:

Якщо вам потрібно представити деревоподібну структуру об’єктів. Патерн Компонувальник пропонує зберігати в складових об’єктах посилання на інші прості або складові об’єкти. Вони, у свою чергу, теж можуть зберігати свої вкладені об’єкти і так далі. У підсумку, ви можете будувати складну деревоподібну структуру даних, використовуючи всього два основних різновида об’єктів.

Якщо клієнти повинні однаково трактувати прості та складові об’єкти. Завдяки тому, що прості та складові об’єкти реалізують спільний інтерфейс, клієнту байдуже, з яким саме об’єктом він працюватиме.

ВИСНОВКИ

Технологія конструювання програмного забезпечення — система інженерних принципів для створення економічного програмного забезпечення, яке надійне і ефективно працює в реальних комп'ютерах. Методи технології забезпечують вирішення наступних завдань: планування і оцінка проекту; аналіз системних і програмних вимог; проектування алгоритмів, структур даних і програмних структур; кодування; тестування; супровід. Застосування парадигм технологій конструювання програмного забезпечення гарантує систематичний, впорядкований підхід до промислової розробки, використання і супроводу програмного забезпечення.

Патерни проектування спрощують повторне використання вдалих проектних і архітектурних рішень. За допомогою патернів можна поліпшити якість документації і супроводу існуючих систем, дозволяючи явно описати взаємодії класів і об'єктів, а також причини, за якими система була побудована так, а не інакше. Простіше кажучи, патерни проектування дають розробнику можливість швидше знайти «правильний» шлях.

Породжуючи патерни проектування абстрагують процес інстанціонування. Вони допоможуть зробити систему незалежною від способу створення, композиції та представлення об'єктів. Патерн, який породжує класи, використовує спадкування, щоб варіювати інстанційований клас, а патерн, який породжує об'єкти, делегує інстанціюваня іншому об'єкту. Ці патерни виявляються важливі, коли система більше залежить від композиції об'єктів, ніж від успадкування класів.

У структурних патернах розглядається питання про те, як із класів та об'єктів утворюються більші структури. Структурні патерни рівня класу використовують успадкування для складання композицій з інтерфейсів та реалізацій.

Паттерни поведінки пов'язані з алгоритмами та розподілом обов'язків між об'єктами. Річ в них йде не лише про самі об'єкти та класи, а й про типові способи взаємодії. Паттерни поведінки характеризують помилковий потік управління, який важко простежити під час виконання програми. Увага акцентована не на потоці управління як такому, а на зв'язках між об'єктами.

СПИСОК ЛІТЕРАТУРИ

1. Бек, Кент. Шаблоны реализации корпоративных приложений.: Пер. с англ.-М.: ООО “И.Д.Вильямс”, 2008.-176с.
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования.-СПб: Питер, 2010.-368 с.
3. Зандстра М. PHP: объекты, шаблоны и методики программирования. 2-е издание.: Пер. с англ. – СПб.: Издательский дом «Вильямс», 2010. – 478 с.
4. Кериевски Д. Рефакторинг с использованием шаблонов.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2006. – 400 с.
5. Ларман К. Применение UML и шаблонов проектирования. 2-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 624 с.
6. Нильсон Д. Применение DDD и шаблонов проектирования: проблемно-ориентированное проектированиеприложений с примерами на C# и .NET.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 560 с.
7. Тепляков С. Паттерны проектирования на платформе .NET. — СПб.: Питер, 2015. — 320 с.
8. Хоп Г., Вульф Б. Шаблоны интеграции корпоративных приложений.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 672 с.