1. Патерни проектування мобільних додатків

1.1 Поняття патерна проектування

Патерн (шаблон) проєктування — це іменований опис проблеми та її розв'язання, яке можна використовувати у розробці інших систем. Таке визначення дає у своїй книзі «Застосування UML і шаблонів проєктування» (Applying UML and Patterns) Крег Ларман — фахівець із гнучкої методології та ітеративної розробки.

Проста аналогія для пояснення: щоби пройти крізь стіну, ми ставимо двері, а якщо потрібно перетнути річку — будуємо міст, тобто для кожного завдання використовуємо відповідне розв'язання. Потрапити на інший берег річки можна і на дерев'яному плоті — це також допоможе впоратись із завданням. Але міст — професійніше, надійніше та довговічніше рішення. Саме такі використовують команди розробників успішних продуктів.

Сучасне програмування ближче до літератури, ніж до математики та інших точних наук. Код — це твір, написаний певною мовою з прийнятими в ній правилами граматики та орфографії. Його можна написати так, аби повідомлення зрозуміло широке коло читачів. А можна — заплутати не лише їх, а й самого себе.

Хоч би якою мовою не писав програміст — йому потрібні інструменти, щоби спрощувати складні конструкції. Патерни проєктування — один із таких інструментів.

1.2 Принципи вибору патернів проектування

Необов'язково використовувати патерн у кожному рядку коду — можна тільки там, де це необхідно для простоти й «читабельності».

У програмуванні є принцип: якщо дія виконується багато разів, її потрібно автоматизувати. Патерни необхідні саме в таких випадках, коли розв'язання використовується багаторазово. Якщо ж операція проста і поодинока (наприклад, потрібно протестувати, чи виконується функція), можна обмежитися перевірним брудним кодом або базовим скриптом.

Щоби писати чистий код, потрібно використовувати не лише патерни. Також програмісти застосовують:

```
class Order
{
```

```
    public void calculate(){ ... }
    public void addItem(Product product){ ... }
    public List<Product> getItems(){ ... }
    ...
    public void load(){ ... }
    public void save(){ ... }
    public void print(){ ... }
}
class Order
{
    public void calculate();
    public void addItem(Product product){ ... }
    public List<Product> getItems(){ ... }
}
class OrderRepository
{
    public Order load(int orderId){ ... }
    public void save(Order order){ ... }
}
class OrderPrinter
{
    public void print(Order order){ ... }
}
```

Навіщо застосовувати патерни

1. Патерни допомагають оптимізувати та підвищити ефективність роботи програмістів. Кодити без патернів — неначе забути, що хтось уже винайшов колесо.

2. Патерни спрощують взаємодію між програмістами. Ніхто не любить незрозумілий код — його доводиться переписувати. На автора такого коду дивляться скоса, як на марнотратника ресурсів команди.

Патерни — це універсальна мова розробників. Кожен патерн передбачає знання певної схеми. Але програмістам не потрібно описувати їх — достатньо сказати «Фабрика» чи «Компоновник». Як, наприклад, ми говоримо «Я зателефоную [телефоном]» замість «Я використовую засіб зв’язку за допомогою радіохвиль, передачі даних, дискретизації, фазового кодування та шифрування».

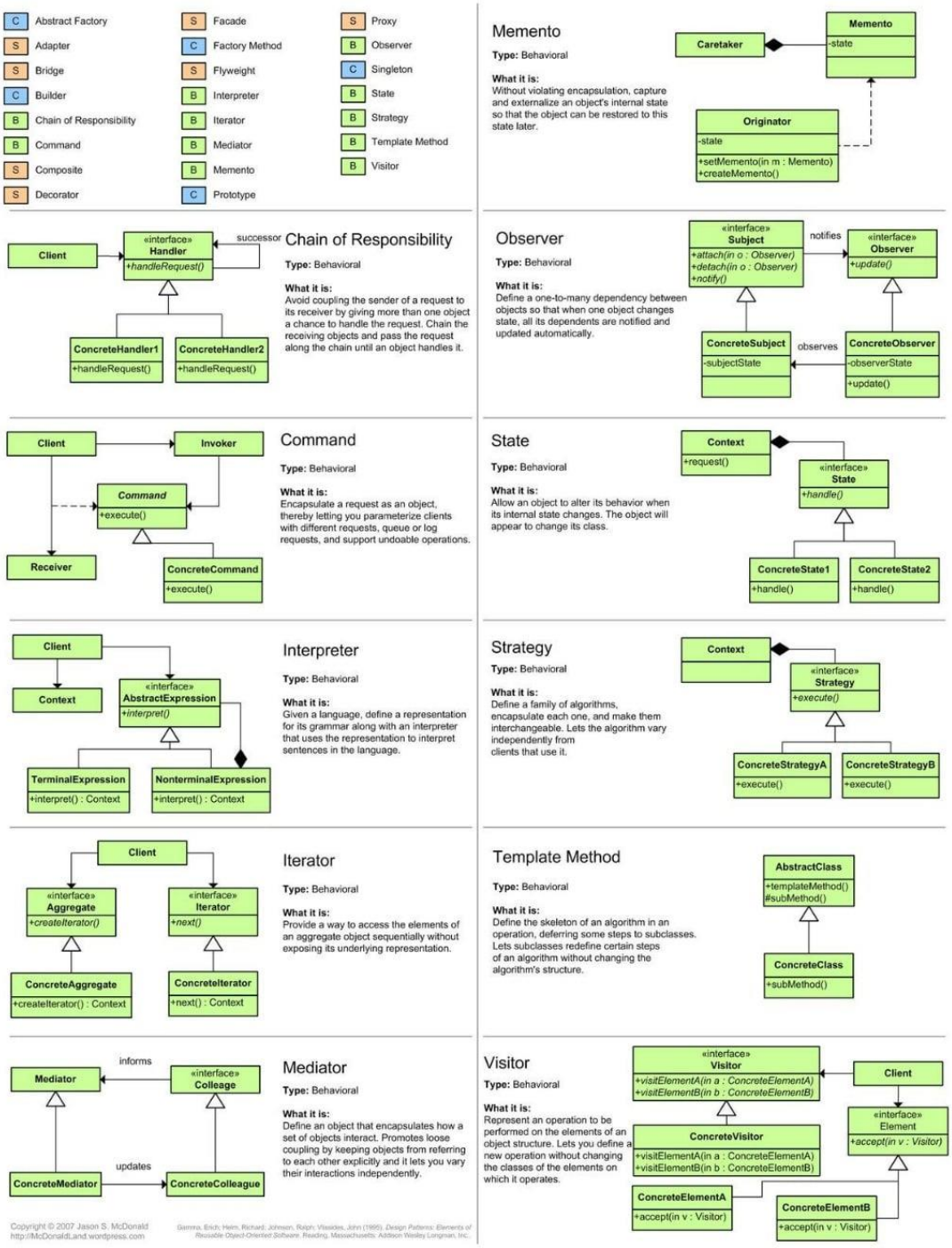
3. Патерни допомагають співпраці програмістів з іншими членами команди — від власника до профільних фахівців. Якісно написаний код прискорює процеси, зменшує кількість помилок та виправлень. Як зводити мости (писати код), потрібно знати тільки будівельникам (програмістам). Користувачам не цікава специфіка конструкцій опор або паль — для них достатньо швидко та безпечно перейти річку.

4. Управління продуктом зазвичай передбачає його поліпшення. Це досягається в тому числі за допомогою рефакторингу — перетвореннязміни вихідного коду, щоби він став простішим і зрозумілішим, але без зміни функціональності. У рефакторингу також не обійтися без патернів.

5. Знання патернів допомагає програмістам — як новачкам, так і досвідченим — у пошуку роботи. На технічній співбесіді менеджери з наймання часто дають завдання написати код — і звертають увагу на те, наскільки ефективно програміст його виконує. У тому числі — чи застосовує патерни там, де це є доцільним.

1.2.1 Які патерни використовують найчастіше

Немає більш чи менш потрібних патернів — кожен із них виконує свою вузьку функцію для розв’язання конкретного завдання.



Приклади графічних анотацій для деяких популярних патернів

Наприклад, у компанії SDL — у команді розробників відомої системи автоматизованого перекладу Trados. Складність завдання полягала в тому, що в оригінальних файлах часто є не тільки текст, а й таблиці, картинки, інші елементи. Плюс система підтримує понад 40 форматів — Word, Excel, PDF, html та десятки інших. У своїй роботі ми найчастіше використовували два патерни:

«Компоновник» (Composite) — об’єднує групи об’єктів у деревоподібну структуру, схожу на меню з пунктами, і дозволяє працювати як з окремими об’єктами, так і з групами.

«Відвідувач» (Visitor) — використовується, коли є багато об’єктів різних класів та інтерфейсів і потрібно виконувати дії над кожним із них.

Майже всі сучасні застосунки використовують під'єднання до серверів та баз даних. У цій операції застосовується патерн «Об’єктний пул» (Object Pool).

Патерн «Прототип» (Prototype) дозволяє створювати об’єкти на основі вже наявних (по суті, копіювати їх). Вбудований у мову програмування JavaScript та її похідні (такі як TypeScript).

Шаблонний метод (Template method) визначає основу алгоритму та допомагає спадкоємцям змінювати кроки без змін загальної структури. Використовується у мовах програмування, де є абстрактні класи (Java, C#, C++ тощо).

Патерн «Ітератор» (Iterator) покращує навігацію колекцією об’єктів.

Патерн «Об’єкт-Значення» (Value Object) використовується для зберігання простих величин (наприклад, таких як гроші або дати).

Сучасні системи часто використовують ліниву ініціалізацію (Lazy initialization) — підхід у програмуванні, коли складна дія виконується «на вимогу». Наприклад, «Фабричний метод» (Factory Method) передбачає, що базовий клас доручає створення об’єктів класам-спадкоємцям. Завдяки цьому не потрібно багаторазово описувати створення об’єкта та вносити зміни до різних підкласів (це робиться один раз).

З новими технологіями постійно з’являються нові патерни. Наприклад, популярність мікросервісів призвела до необхідності патернів Database per Service, External Configuration, Service Discovery та Circuit Breaker. Потреба розуміти стан розподілених систем — до необхідності патернів Log Aggregation, Distributed Tracing та Health Check. Акцент сучасних систем на маркетингу в тому числі сприяв популярності патерну Blue-Green Deployment.

Під час використання патернів є дві основні помилки: застосування там, де без нього можна обійтися, і відсутність там, де патерн відмінно підходить. Як наслідок — код складніше розуміти, підтримувати та дорожче супроводжувати.

1.3 Структурні патерни. Патерни поведінки

Об'єктно-орієнтоване програмування (ООП) - Парадигма програмування, заснована на представленні предметної області у вигляді взаємопов'язаних об'єктів, що належать різним класам.

У ООП вводиться поняття класу. Клас - користувальницький тип даних, що поєднує дані та методи їх обробки. Об'єктом називається екземпляр класу.

Приклади класу та об'єкта:

1) кішка – це клас, кішка Мурка із 29 квартири – це екземпляр класу «Кішка», тобто. об'єкт;

2) студент – це клас, студент Іванов із 332 групи ХАІ зростом 174 см – це екземпляр класу «Студент»;

3) вантажний літак – це клас, літак АН-70, що має серійний номер 1845838 та побудований у 2011 році, – це екземпляр класу «Вантажний літак».

Мовою С++ оголошення класу здійснюється у модулі з розширенням *.h, а створення об'єктів та операції з ними – у модулі *.cpp. Розглянемо у загальному вигляді оголошення класу:

class <ім'я класу> {

// поля класу

//Методи класу

};

Поля класу зберігають всю необхідну інформацію про об'єкт, формують стан, характеристики. Зміна стану об'єкта та його характеристик пов'язане із зміною значень його полів.

Клас може містити один або більше методів, що дають змогу маніпулювати даними (значеннями полів) об'єкта. Метод об'єкта – програмний код, виконаний як функції, реагує на передачу об'єкту певного повідомлення.

Конструктор класу – спеціальний блок інструкцій, що викликається під час створення об'єкта. Конструктор являє собою особливий метод, який відрізняється від інших тим, що не має певним чином певного типу даних, що повертаються, і його ім'я збігається з ім'ям класу, в якому він оголошується.

<Ім'я класу>(<Список вхідних аргументів>) {

// Операції конструктора

}

Деструктор – автоматично запускається щоразу, коли програма знищує об'єкт. Подібно до конструктора деструктор має таке ж ім'я, як і клас об'єкта.

Однак у випадку деструктора необхідно попереджати його ім'я символом тильди (~), як показано нижче:

~<Ім'я класу>() {

// Операції деструктора

}

Діаграма класів - Описує структуру системи, показуючи її класи, їх поля, методи, а також взаємозв'язки цих класів (рисунок 1.1). Клас на діаграмі зображується прямокутником, розділеним на три частини – ім'я класу, перелік полів та методів класу.

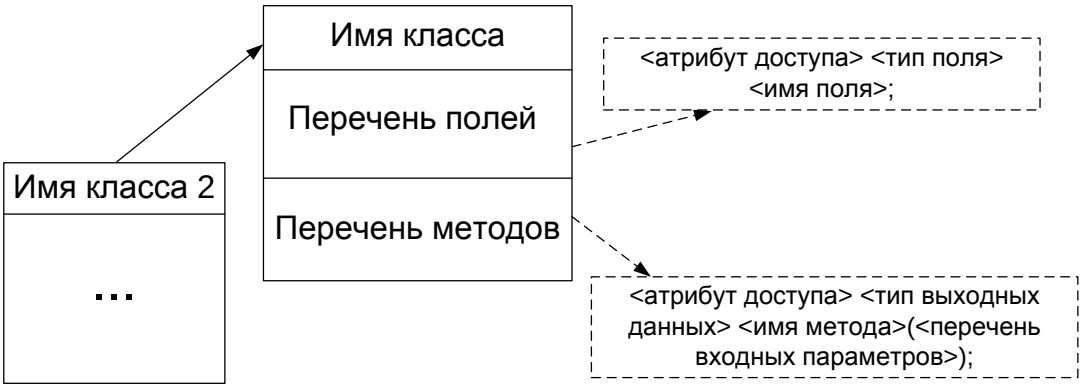


Рисунок 1.1 – Подання діаграми класів

Найважливіші принципи ООП:

1. Абстракція даних – об'єкти надають неповну інформацію про реальні сутності предметної області. Абстрагування дозволяє оперувати об'єктами на рівні, адекватному розв'язуваній задачі.

2. Інкапсуляція – здатність об'єкта приховувати внутрішній устрій своїх полів та методів. Згідно з цим принципом клас необхідно розглядати як чорну скриньку, зовнішній користувач не знає деталі реалізації об'єкта та працює через наданий об'єктом інтерфейс. Дотримання цього принципу може зменшити кількість зв'язків між класами та спростити їх реалізацію, модифікацію, тестування та використання.

Для розділення прав доступу до полів та методів класу використовують ключові слова (модифікатори доступу):

+ public – доступом до атрибутам можливий з частини коду, тобто. із самого класу, його спадкоємців та зовнішніх класів;

- private – доступ до атрибутів дозволено лише з методів поточного класу;

protected – доступ до атрибутів дозволено зсередини методів даного класу та всіх його нащадків.

3. Спадкування – полягає у породженні нових класів на основі вже існуючого батьківського (базового) класу. У клас-нащадок можна додати свої власні властивості та методи, користуватися методами та властивостями базового класу.

4. Поліморфізм – явище, у якому класи-нащадки можуть змінювати реалізацію методу класу-предка, зберігаючи його інтерфейс. Поліморфізм дозволяє обробляти об'єкти класів-нащадків як однотипні, як і раніше, що й методи можуть реалізовуватися по-різному.

Абстрагування застосовується лише на етапі проектування структури програмної системи, а інкапсуляція, спадкування та поліморфізм реалізуються у її програмному коді. Розглянемо приклад системи геометричних фігур з базовим класом Shape та спадкоємцями Point, Line та Rectangle (рисунок 1.2).

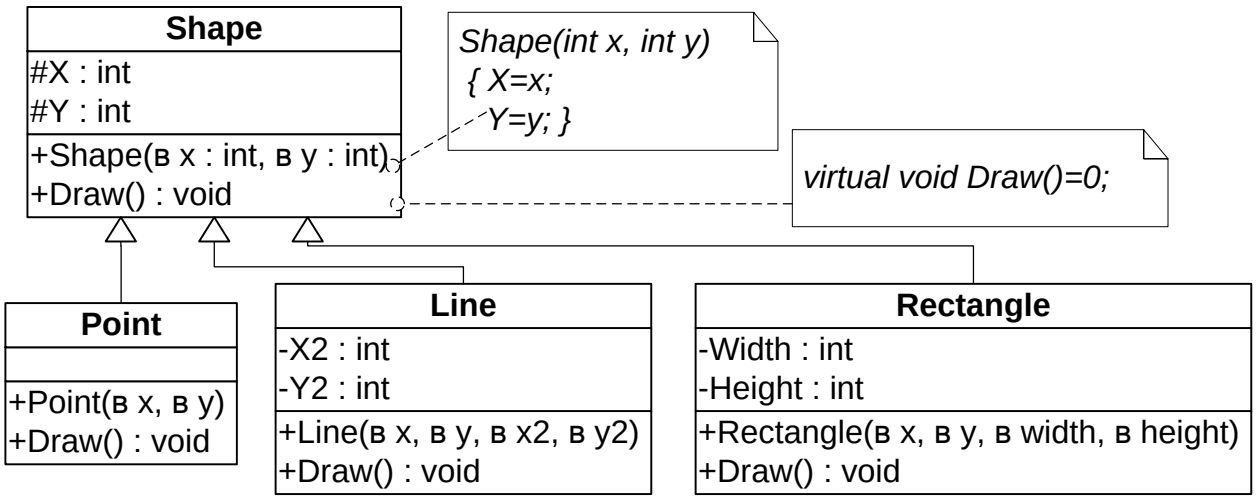


Рисунок 1.2 – Діаграма класів

Клас Shape – базовий абстрактний клас, містить конструктор Shape, а також суто віртуальну функцію Draw, яка не вимагає реалізації. Спадкоємці перевизначають суто віртуальну функцію Draw, пропонуючи власну реалізацію цієї функції для промальовування на екрані відповідного графічного примітиву (приклад реалізації поліморфізму). Клас Shape містить два параметри X і Y (координати положення примітиву на екрані) у protected-секції, які успадковуються класами-спадкоємцями (приклад реалізації інкапсуляції). Класи Line та Rectangle розширюють набір параметрів новими полями X2, Y2

(координати кінцевої точки лінії) та Width, Height (ширина та висота прямокутника) (приклад реалізації наслідування).

Приклад реалізації методів класу Rectangle у модулі *.cpp:

```
//Реалізація конструктора класу Rectangle, попередньо
//викликаємо конструктор Shape
Rectangle::Rectangle(int x, int y, int width, int height):Shape(x,y){
 Width = width;
 Height = height;
}
void Rectangle::Draw(){
 //Малювання прямокутника
}
```

Приклад створення кількох об'єктів графічних примітивів:

```
Point * pnt = new Point (10, 35);
Line * ln = new Line(40, 30, 120, 90);
Rectangle * rct = new Rectangle(70, 74, 100, 200);
//скористаємося перевагами поліморфізму
QList<Shape*> figures;
figures.Add(pnt);
figures.Add(ln);
figures.Add(rct);
for (int i=0;i<figures.count();i++) figures.Draw();
```

В результаті створено об'єкти трьох різних графічних примітивів, додано до списку figures і для кожного з об'єктів викликано функцію Draw, що має різну реалізацію залежно від типу об'єкта (одна команда – різні реакції).

Контрольні питання

1. У чому відмінність понять «клас» та «об'єкт»? Наведіть приклади класів та об'єктів.

2. Опишіть внутрішній пристрій класу.

3. Поясніть суть принципу абстракції даних ООП. Наведіть приклади використання принципу.

4. Поясніть суть принципу інкапсуляції в ООП. Наведіть приклади використання принципу.

5. Поясніть суть принципу успадкування в ООП. Наведіть приклади використання принципу.

6. Поясніть суть принципу поліморфізму ООП. Наведіть приклади використання принципу.

7. Опишіть особливості реалізації принципів ООП з прикладу програмного коду описи класів певної предметної області.

1.3.1 ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ «СПОСТЕРЕЖУВАЧ» (OBSERVER)

Якісні рішення об'єктно-орієнтованих програм мають бути спроектовані так, щоб їх можна було використати повторно. Структура моделі класів повинна відповідати розв'язуваній задачі та бути досить гнучкою для можливості її подальшої модифікації та поліпшення. Внаслідок того, що проектування розв'язання задачі з нуля – складна проблема, досвідчені програмісти повторно використовують ті рішення, які показали високий ступінь ефективності на практиці. Опис такого рішення називається шаблоном проектування.

Шаблон проектування - опис взаємодії об'єктів і класів, що представляє собою вирішення проблеми проектування в рамках деякого контексту, що часто виникає. Загалом шаблон складається з чотирьох основних елементів: ім'я (посилання на ім'я шаблону дозволяє відразу описати проблему проектування, її вирішення); завдання (опис контексту, котрим доцільно застосовувати шаблон); рішення (загальний опис структури класів та об'єктів вирішення проблеми); результати (наслідки застосування шаблону).

Розглянемо структуру поведінкового шаблону "Спостерігач", відомого також під назвою "видавець-передплатник".

Шаблон "Спостерігач" 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.

Доцільно застосовувати шаблон «Спостерігач» у ситуаціях, коли при модифікації одного об'єкта потрібно змінити кілька інших і об'єкти, що оповіщаються, повинні бути незалежними або слабко пов'язаними один з одним.

Ключові об'єкти шаблону – суб'єкт та спостерігач. Суб'єкт може мати скільки завгодно залежних від нього спостерігачів, які повідомляють про зміни стану суб'єкта. Отримавши повідомлення, спостерігач опитує суб'єкта, щоб синхронізувати свій стан. Діаграма класів описаного шаблону показано малюнку

2.1.

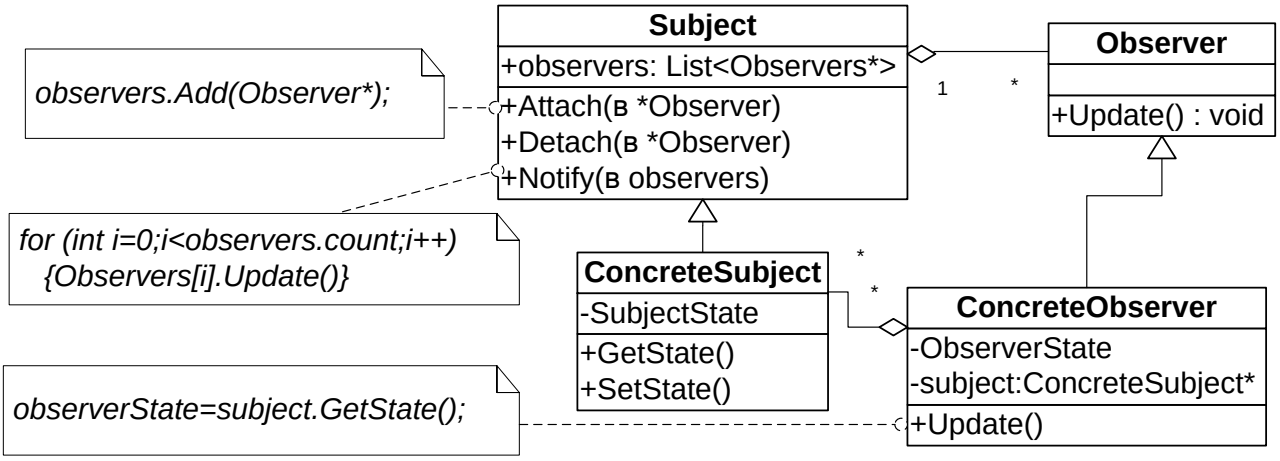


Рисунок 2.1 – Діаграма класів шаблону «Спостерігач»

Елементи діаграми класів:

1. Subject – базовий клас суб'єкта, що зберігає інформацію про список спостерігачів observers, а також містить методи для додавання нових спостерігачів (Attach), видалення доданих раніше (Detach) та розсилки повідомлень спостерігачам про зміну суб'єкта (NotifyObservers).

2. ConcreteSubject – клас конкретного суб'єкта, що зберігає інформацію про стан суб'єкта subjectState, а також містить методи для отримання даних про поточний стан суб'єкта (GetState) та зміну стану суб'єкта (SetState).

3. Observer – абстрактний клас спостерігачів, що містить суто віртуальну функцію Update.

4. ConcreteObserver – клас конкретного спостерігача, містить посилання суб'єкт subject, і навіть метод оновлення стану спостерігача Update.

При зміні стану суб'єкт розсилає повідомлення всім спостерігачам про свою зміну. Отримавши повідомлення, спостерігач запитує дані суб'єкта і, отримавши їх, оновлює свій стан.

Один із варіантів реалізації шаблону мовою C++:

Файл Observer.h:

```
class Observer {
 public:
 virtual void Update(Subject *achanged_subject)=0;
};
```

```
class Subject{
 private:
 QList<Observer*> observers;
 public:
 Subject();
 virtual void Attach (Observer * observer);
 virtual void Detach(Observer *observer);
 virtual void NotifyObservers();
 ~Subject();
};
class ConcreteSubject1:public Subject{
 public:
 ConcreteSubject1();
 int GetState();
 void SetState();
 ~ ConcreteSubject1 ();
};
class ConcreteObserver1: public Observer{
 private:
 Subject * subject;
 public:
 ConcreteObserver1(ConcreteSubject * cSubject);
 void Update(Subject * asubject);
 ~ ConcreteObserver1();
};
```

Приклади використання шаблону «Спостерігач»:

1. Поведінка живих організмів перед початком дощу. Різні живі організми по-різному реагують наближення грозового фронту. Якщо уявити небо як суб'єкт, то розсилкою повідомлень живим організмам, тобто. спостерігачам, логічно вважати зміну кольору неба на темніший, а також гуркіт грому. При цьому спостерігачі, які отримують повідомлення, часто не знають про існування один одного, проте виконують діяльність, кінцева мета якої – порятунок від дощу. У цьому прикладі: небо – суб'єкт; живі організми – спостерігачі; грім - розсилання повідомлень; спрямований до неба погляд живих організмів – запит до суб'єкта про його стан; підготовка до дощу організмів – відновлення стану спостерігачів.

2. Передплата журналу. Видавець розсилає повідомлення про вихід журналу всім передплатникам. Останні, отримавши повідомлення, прямують на пошту, щоб забрати журнал і поповнити свій багаж знань новими (прочитавши свіжий номер журналу).

3. Відображення інформації про швидкість літака у цифровому та аналоговому вигляді на приладовій дошці. В даному випадку датчик швидкості виступає в ролі суб'єкта, цифровий та аналоговий спідометри – спостерігачі.

Контрольні питання

1. Дайте визначення шаблону проектування.

2. Які переваги надає використання шаблонів проектування при створенні архітектури системи, що розробляється?

3. Поясніть призначення шаблону «Спостерігач».

4. Натисніть на діаграму класів шаблону «Спостерігач».

5. Поясніть призначення та функції кожного із класів створеної Вами програми.

6. Наведіть приклад використання ідеї шаблону «Спостерігач» у природній, соціальній чи технічній сферах.

1.3.2 ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ «АДАПТЕР» (ADAPTER)

Шаблон "Адаптер" відноситься до типу структурних шаблонів, які компонують об'єкти для отримання нової функціональності. Додаткова гнучкість у разі пов'язані з можливістю змінити композицію об'єктів під час виконання, що неприпустимо для статичної композиції класів.

Шаблон "Адаптер" служить для перетворення інтерфейсу одного класу в інтерфейс іншого, на який очікують клієнти. Таким чином, забезпечується спільна робота класів із несумісними інтерфейсами.

Рекомендується застосовувати шаблон у випадках, коли:

1) доцільно використовувати існуючий клас, та його інтерфейс відповідає поточним потребам;

2) необхідно створити повторно використовуваний клас, який повинен взаємодіяти із заздалегідь не відомими або не пов'язаними з ним класами, що мають несумісні інтерфейси;

2) 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.

Узагальнена структура шаблону «Адаптер» показано малюнку 3.1. Client використовує інтерфейс, визначений класом Target відповідно до особливостей предметної області. Клас Adaptee визначає існуючий інтерфейс, який потребує адаптації. Клас Adapter адаптує інтерфейс Adaptee до Target. Таким чином, клієнти викликають операції екземпляра адаптера Adapter. У свою чергу, адаптер викликає операції об'єкта, що адаптується, або класу Adaptee, який і виконує запит.

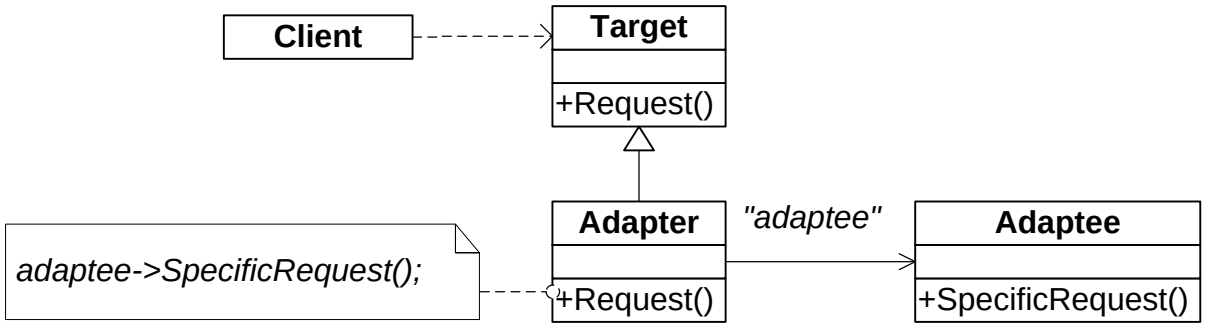


Рисунок 3.1 - Узагальнена структура шаблону "Адаптер"

Розглянемо деякі приклади використання шаблону «Адаптер»:

1. Дитинча кенгуру неспроможна відповідати відразу після народження тим вимогам, які пред'являє щодо нього тваринний світ виживання у ньому. Він не володіє достатньою силою та досвідом для того, щоб самостійно добувати їжу та захищатися від ворогів. Мати дитинчати кенгуру виступає в ролі адаптера, що забезпечує безпечне існування дитинчати, що адаптується в природі і поступове його пристосування до умов навколишнього середовища.

2. Персональний перекладач делегата організації на симпозіумі. Уявімо ситуацію, коли деяка організація відправила на симпозіум свого делегата для ведення важливих переговорів. Але делегат не володіє мовою, якою проводять переговори. Для подолання цієї складності організатори симпозіуму надають допомогу делегату перекладача. У цьому прикладі як адаптер виступає перекладач, як адаптований – делегат.

3. Веб-додаток, що працює з різними системами управління базами даних (СУБД). Нехай є веб-додаток, що взаємодіє з реляційною базою даних (БД). Можна визначити інтерфейс DB, який описує методи, що дозволяють підключитися до БД, надіслати запит БД та отримати результат виконання запиту у певному вигляді, а також інші необхідні дії. Але оскільки заздалегідь не відомо, з якою саме СУБД належить працювати веб-додатку (MySQL, PostgreSQL, Interbase), потрібно написати реалізацію інтерфейсу DB для популярних СУБД. Написання реалізації для популярних СУБД - трудомістка задача, тому розумно скористатися готовими реалізаціями і лише адаптувати їх під інтерфейс DB. На малюнку 3.2 зображено діаграму класів описаного рішення, що реалізує шаблон "Адаптер".

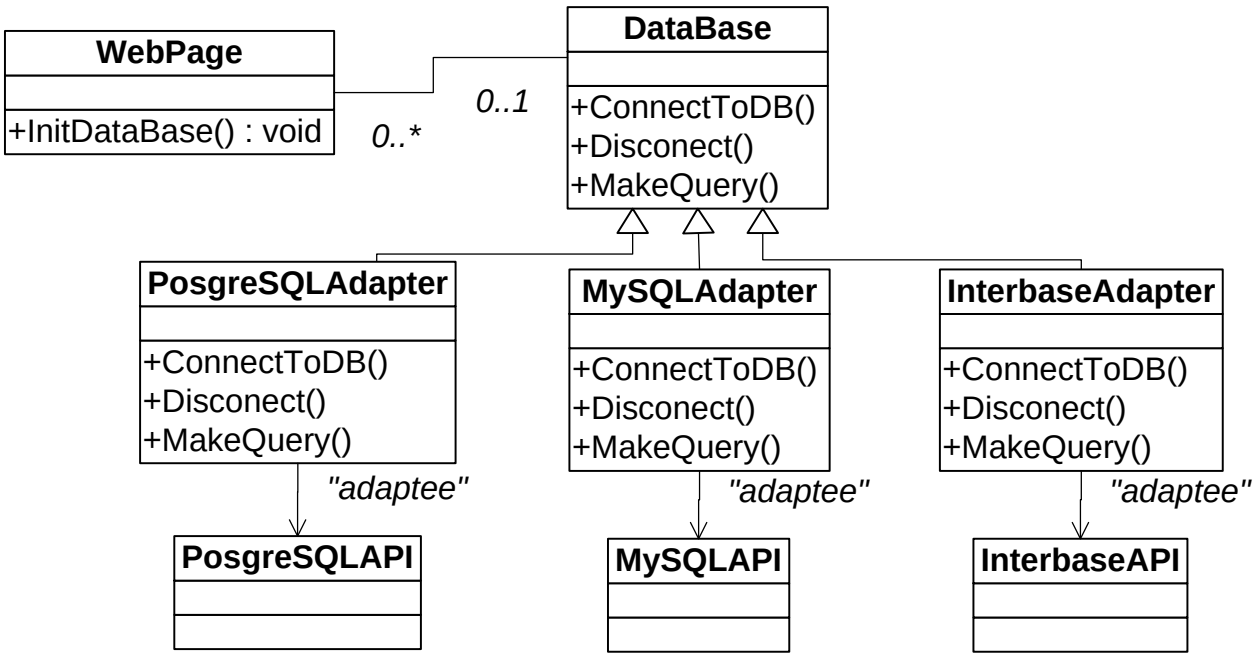


Рисунок 3.2 – Діаграма класів веб-програми

У цьому прикладі адаптери – це класи PosgreSQLAdapter, MySQLAdapter, InterbaseAdapter, а адаптовані – СУБД PosgreSQL, MySQL, Interbase, узагальнено представлені класами PosgreSQLAPI, MySQLAPI та InterbaseAPI.

Контрольні питання

1. Поясніть призначення шаблону "Адаптер".
2. Натисніть на діаграму класів шаблону «Адаптер».

3. Поясніть призначення та функції кожного з класів створеної Вами програми.

4. Наведіть приклад використання шаблону «Адаптер» у природній, соціальній чи технічній сферах.

1.3.3 ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ «КОМПОНОВЩИК» (COMPOSITE)

Шаблон «Компоновщик» відноситься до структурного типу. Він служить для компонування об'єктів у деревоподібні структури для представлення ієрархій частина-ціле, що спрямоване на забезпечення можливості клієнтів однаково трактувати індивідуальні та складові об'єкти. Шаблон «Компоновщик» застосовують для вирішення багатьох типів завдань, зокрема, при побудові графічного редактора або інтерфейсу програми, коли необхідно виконувати однотипні операції (наприклад, переміщення) з простими та складовими об'єктами, не замислюючись про відмінність у їх структурі та типах. Узагальнена структура класів шаблону «Компоновщик» показана малюнку 4.1.

Базовий клас ієрархії – абстрактний клас Component, що має два спадкоємці – Leaf (клас простих об'єктів ієрархії) та Composite (клас складових об'єктів, що містять список об'єктів components базового типу Component). При виклику команди Operation() простий об'єкт виконує цільові дії, а складовий – крім своїх функцій ініціює однойменні операції кожного з його складових. Таким чином, відбувається автоматичний спуск операцій Operation() від складових об'єктів до їх складових.

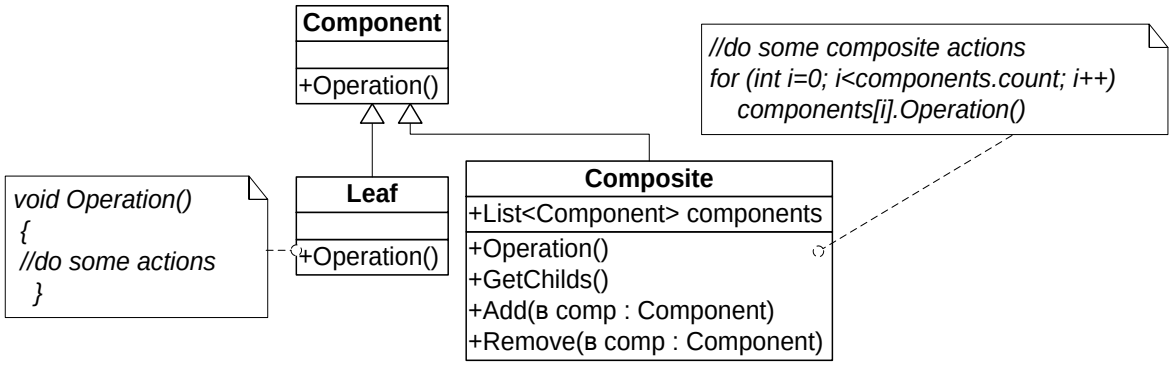


Рисунок 4.1 – Узагальнена структура шаблону «Компонувальник»

Рекомендується використовувати шаблон «Компонувальник»:

1) якщо необхідно уявити ієрархію об'єктів виду частина-ціле;

2) потрібно, щоб клієнти однаково трактували складові та індивідуальні об'єкти.

Розглянемо деякі приклади застосування шаблону «Компоновщик»:

1. Рух поживних речовин, у рослинах. Кожна рослина споживає поживні речовини і воду через корінь і транспортує їх по стовбуру до гілок, листя і в кінцевому підсумку до окремих клітин листя. Рослина – це ієрархічна структура, тому процес транспортування поживних речовин від кореня (ствола) до окремих клітин можна уявити за допомогою шаблону «Компоновщик». У цьому випадку стовбур, гілки, листя - це складові об'єкти, що споживають деяку кількість поживних речовин і направляють ті, що залишилися своїм складовим, а клітини - прості.

2. Збір гуманітарної допомоги працівниками підприємства. Ініціатива зі збирання благодійної допомоги (наприклад, у вигляді теплих речей) належить директору підприємства. Він поповнює фонд зібраної допомоги своїм вкладом та дає вказівку начальникам відділів здійснити збір допомоги у межах усіх підрозділів. Останні поповнюють фонд своїм вкладом, а також допомогою зібраної заступниками, які проводять збір серед своїх підлеглих. Зрештою директор підприємства може відправити фонд зібраної допомоги тим, хто її потребує, адресатам.

3. Аналізатор абстрактного синтаксичного дерева (АСД) програмного коду алгоритму розв'язання певної задачі. Відповідно до визначення АСД в інформатиці сприймається як кінцеве, позначене, орієнтоване дерево, у якому внутрішні вершини зіставлені з операторами мови програмування, а листя — з відповідними операндами. Нехай є фрагмент програмного коду такого виду:

double b = 4; a = b * sin (x + 0.5);

Один із варіантів побудови АСД для даного фрагмента програмного коду показаний на малюнку 4.2.

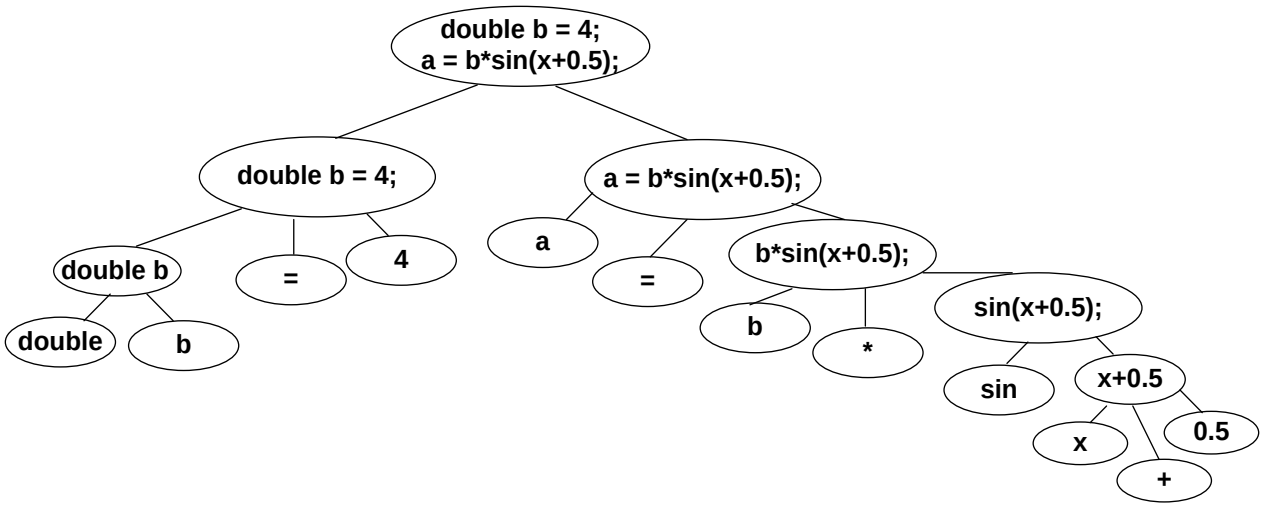


Рисунок 4.2 – АСД фрагмента програмного коду

Для аналізу програмного коду та представлення його АСД у вигляді деревоподібної структури необхідно створити спеціальну підпрограму, що реалізує шаблон «Компоновщик». Можливий вид діаграми класів аналізатора АСД показаний малюнку 4.3.

Базовий клас - абстрактний клас Node, однією з властивостей якого є список складових вузлів поточного вузла. Він має два спадкоємця – клас нетермінальних вузлів NotTerminalNode (що містять складові вузли) та термінальних TerminalNode (що не містять складові вузли). Функція GetNodes() класу NotTerminalNode визначає тип поточного вузла (наприклад, оголошення змінної, присвоєння значення змінної, виклик функції тощо), поділяє його на складові (наприклад, вузол типу Присвоєння значення змінної a = 7 можна розкласти на три термінальних вузла a, = та 7), поповнює список connections поточного вузла термінальними та нетермінальними вузлами та викликає функції GetNodes() для кожного складового вузла списку connections. Для вузла термінального типу функція GetNodes() не виконує жодних подальших дій щодо розкладання поточного вузла.

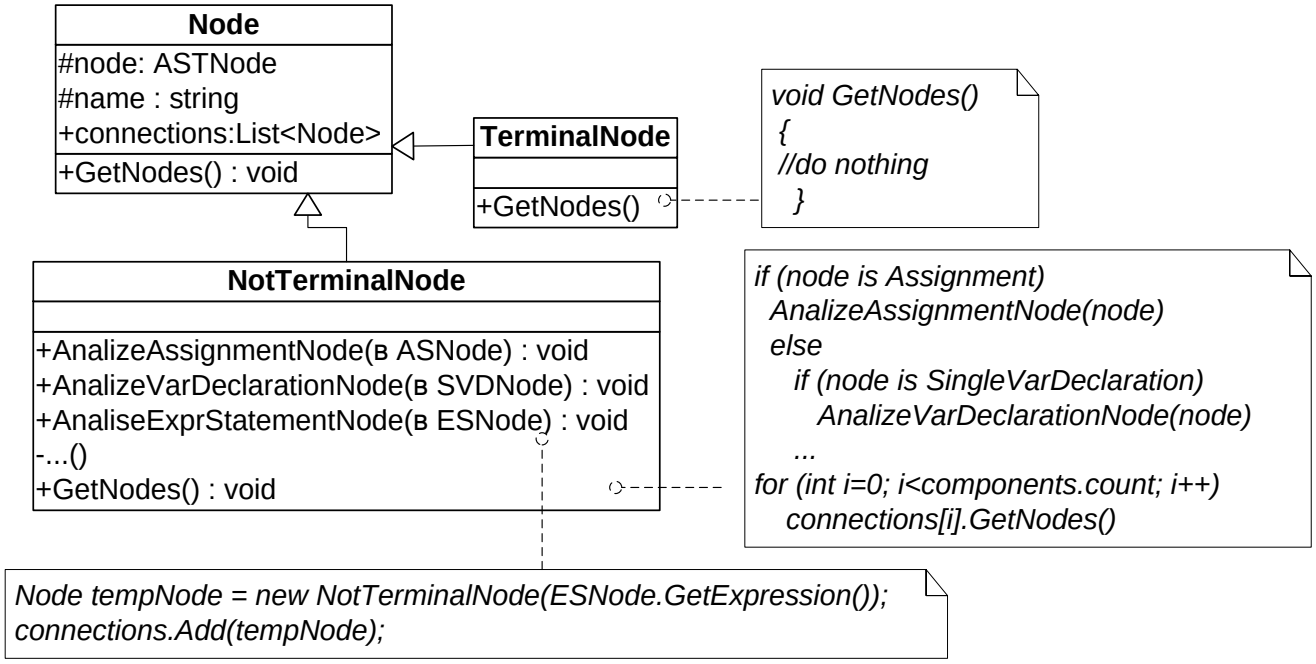


Рисунок 4.3 – Приклад діаграми класів аналізатора АСД

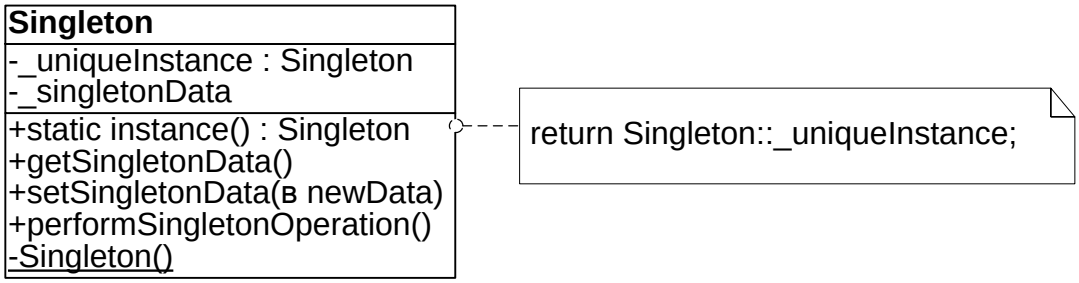
Таким чином, достатньо викликати функцію GetNodes() один раз для верхнього елемента АСД програмного коду, подальше розкладання його на складові програма виконає автоматично.

Контрольні питання

1. Зобразіть діаграму класів шаблону «Компонувальник».
2. Поясніть призначення та функції кожного з класів створеної Вами програми.
3. Поясніть призначення АСД та особливості застосування шаблону «Компоновщик».
4. Наведіть приклад використання шаблону «Компоновщик» у природній, соціальній чи технічній сферах.

1.3.4 ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ «ОДИНАК» (SINGLETON)

«Сінглтон» (або «одинак») – це шаблон проектування, який застосовують у тих випадках, коли необхідно контролювати існування об'єкта певного класу в єдиному екземплярі для всього програмного продукту. «Сінглтон» рекомендують використовувати замість статичних класів, щоб заощаджувати пам'ять і точно знати, коли виконається його конструктор.



Малюнок 5.1 – Структура шаблону «Сінглтон»

Структура шаблону показано малюнку 5.1. Слід звернути увагу, що конструктор класу Singleton – приватний, тобто. не може бути викликаний із зовнішнього коду. Таким чином, єдиний спосіб отримати доступ до екземпляра класу Singleton – викликати метод Singleton::Instance(), оголошений як статичний. Отже, він може бути викликаний з будь-якого місця в коді, де дозволено доступ до класу Singleton. Саме метод Singleton::Instance() повинен забезпечити можливість створення лише одного екземпляра Singleton для всього програмного продукту. Один із варіантів реалізації шаблону мовою C++ можна записати так:

Файл Singleton.h

```
class Singleton {
private:
 static Singleton* _uniqueInstance;
 char * _singletonData;
public:
 static Singleton* Instance();
 char* getSingletonData();
 void setSingletonData(char *newData);
 void performSingletonOperation();
private:
 Singleton(); };
```

Файл Singleton.cpp

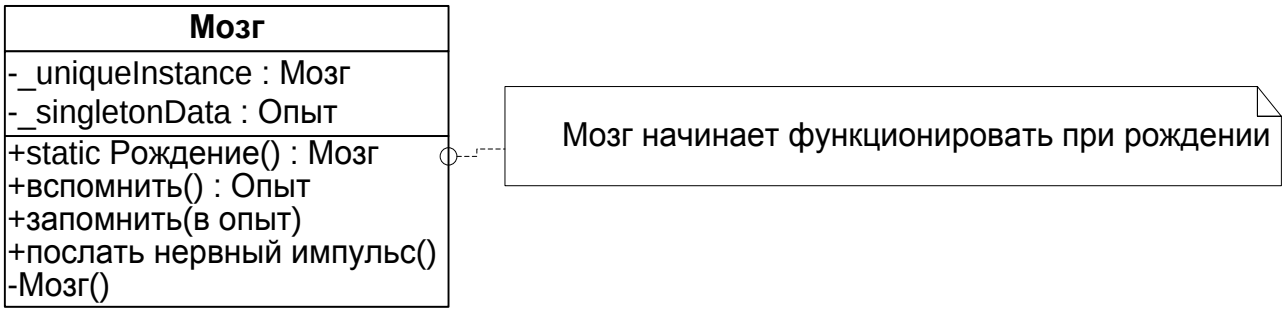
```
#include "singleton.h"
#include <iostream>;
Singleton* Singleton::_uniqueInstance = 0;
Singleton* Singleton::Instance() {
 if (Singleton::_uniqueInstance == 0)
 Singleton::_uniqueInstance = New Singleton();
 return Singleton::_uniqueInstance; }
char* Singleton::getSingletonData() {
 return this->_singletonData; }
void Singleton::setSingletonData(char *newData) {
 this->_singletonData = newData; }
void Singleton::performSingletonOperation() {
 std::cout << this->getSingletonData() << std::endl; }
Singleton::Singleton() { }
```

У наведеному прикладі клас Singleton служить для зберігання глобальної змінної _singletonData, значення якої можна отримати, викликавши метод getSingletonData(), а змінити цю змінну можливо за допомогою методу setSingletonData(), передавши як параметр нове значення. За допомогою Singleton можна також забезпечити доступ із будь-якої точки коду до певних операцій, оскільки сам Singleton, як правило, доступний скрізь. У цьому прикладі це здійснюється шляхом виклику методу performSingletonOperation(), який поміщає в стандартний потік виведення значення змінної _singletonData. Звичайно, Singleton може включати будь-яку кількість змінних, а також будь-яку кількість методів. Як правило, такі змінні зберігають глобальні налаштування проекту (наприклад, розмір і колір шрифту) або посилання на об'єкти (наприклад, виділений/активний елемент редактора), доступ до них повинен бути забезпечений для всього програмного продукту. Методи Singleton – це типові для всієї програми команди, такі як «вирізати», «копіювати», «вставити», або більш специфічні для конкретного програмного забезпечення (ПЗ). У цьому ПЗ прийнято прагнути до потокобезопасной реалізації Singleton, і навіть до реалізації з відкладеним («ледачим») інстанціюванням.

У MSDN (Microsoft Developer Network – бібліотека офіційної технічної документації для розробників під операційну систему Windows) наведено такі приклади використання "Одиночки", як глобальний журнал, де всі сервіси

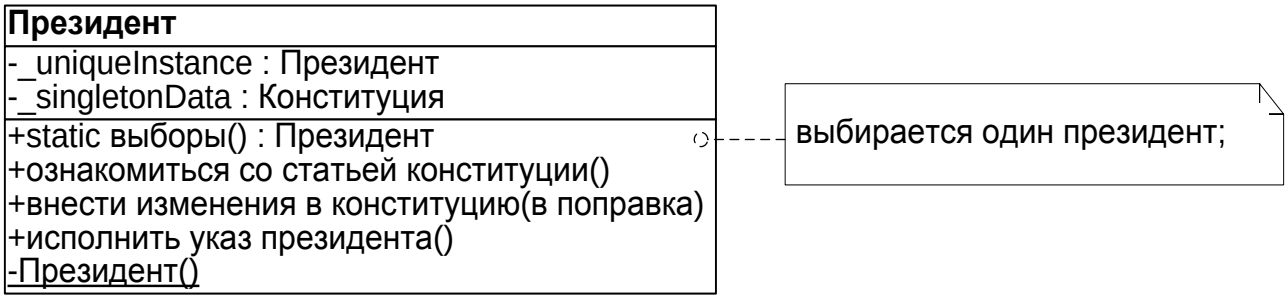
повинні виводити дані про свою роботу, єдиний комунікаційний порт або єдиний механічний двигун.

Якщо проводити аналогії між «Одиночкою» та природою, то одним із прикладів може бути людський мозок (рисунок 5.2). Цей орган в людини один, він здатний накопичувати інформацію, необхідну функціонування окремих органів протягом усього життя. Крім того, мозок – це орган, який віддає команди на основі накопиченої інформації. Ці команди, як правило, є реакцією на певні зовнішні обурення або просто на органи почуттів. Залежно від накопиченого мозком досвіду, ці дії можуть реалізовуватися по-різному.



Малюнок 5.2 – Мозок як аналог шаблону «Одиночка» у природі

У соціальній сфері як аналог «Одиночки» можна розглядати президента (рисунок 5.3). Зараз у країні може бути лише один повноважний президент. При цьому діє конституція, де описані основні закони держави, та існують певні укази президента, які він має право видавати на основі цієї конституції.



Малюнок 5.3 – Президент як соціальний аналог шаблону «Одиночка»

У технічних системах управління центральний процесор можна як «Одиночку» (рисунок 5.4). Центральний процесор – монолітний чи складовий у багатопроцесорних системах, але у обох випадках це центральний орган, який

збирає інформацію від елементів і віддає команди управління. Центральний процесор починає працювати у момент подачі живлення, а закінчує – у момент відключення живлення. Таким чином, він функціонує протягом усього робочого циклу управління.

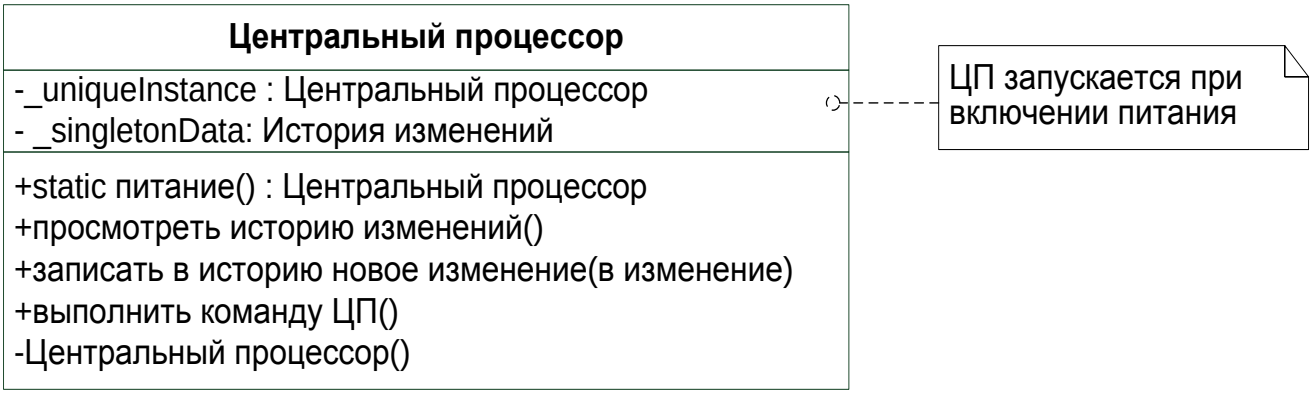


Рисунок 5.4 – Центральний процесор як технічний аналог «Одиночки»

Контрольні питання

Перерахуйте основні вимоги до класу, який є базовим для створення «Одиночки».

Наведіть приклади аналога «Одиночки» у природній, соціальній та технічній сферах.

Скільки екземплярів класу Singleton може бути створено в проміжку між запуском та зупинкою програмного продукту? Чим це зумовлено?

Які вимоги мають бути дотримані в процесі проектування класу Singleton в однопотоковому та багатопотоковому додатках?

Яка мінімальна кількість статичних полів та методів повинен мати базовий клас для створення «Одиночки»?

Чи потрібний деструктор для класу, що реалізує «Одиночку»? Поясніть, чому.

1.3.5 ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ «АБСТРАКТНА ФАБРИКА» (ABSTRACT FABRIC)

Абстрактна фабрика– це шаблон проектування, який застосовують у тих випадках, коли необхідно за допомогою єдиного інтерфейсу інстанцувати деякий

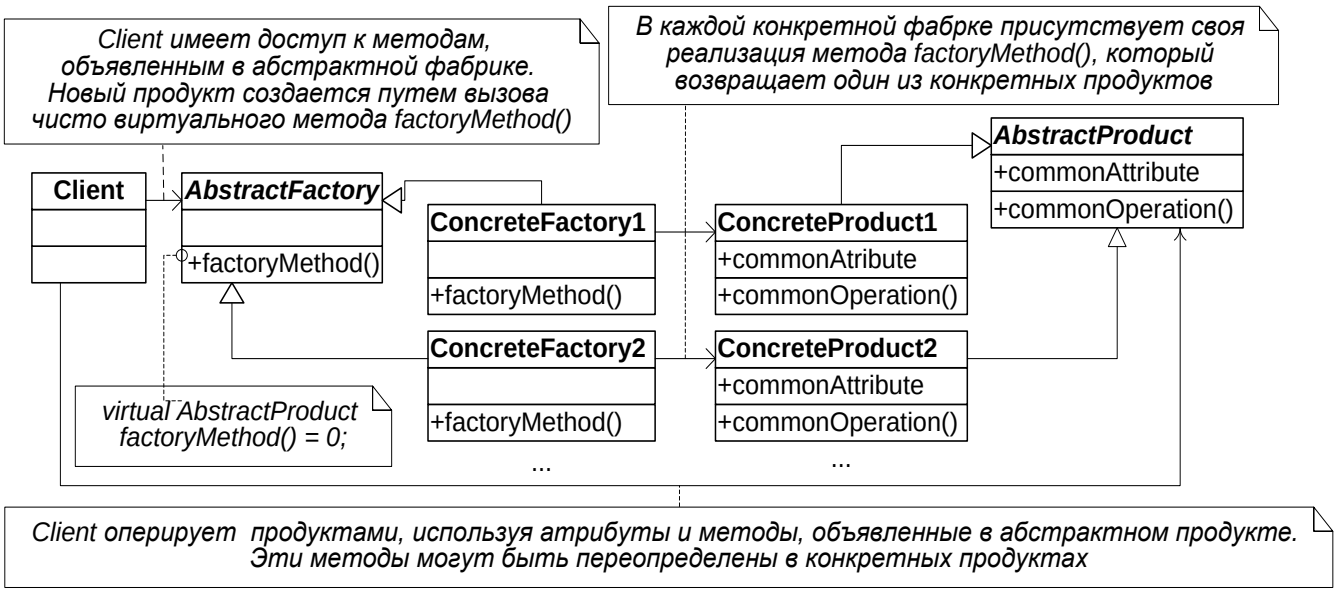
набір об'єктів, які мають однаковий інтерфейс. Розглянемо особливості цього шаблону на прикладі.

Припустимо, потрібно вирішити задачу моделювання системи управління (СУ), що включає безліч різнорідних датчиків. Незважаючи на те, що кожен із датчиків використовує свій специфічний спосіб виміру, а також вимірює різні величини, всі вони повинні реалізовувати одну загальну функцію - вимірювання цільового параметра. Для кожного датчика повинна бути передбачена операція зчитування останньої виміряної величини. Таким чином, всі датчики є набором об'єктів з однаковим інтерфейсом, де ключові операції - це вимірювання цільової величини і читання значення останнього вимірювання.

У різних СУ використовують різні зміни набору датчиків. Наприклад, СУ електричним двигуном може бути оснащена датчиками напруги, кутової швидкості та кута, а система кондиціювання – датчиками температури, вологості та тиску. Незважаючи на різну конфігурацію, у першому та другому випадках для реалізації датчиків необхідний той самий інтерфейс (команди «провести вимір» і «рахувати останній вимір» для кожного з датчиків). Система конфігурації набору датчиків також повинна мати той самий інтерфейс. Це дозволяє створювати системи управління з різною конфігурацією датчиків, використовуючи ту саму команду.

Отже, до створення конкретної СУ необхідно направити команду із зазначенням типу створюваної СУ системі конфігурації, остання, своєю чергою, сама вибере типи датчиків. Після цього для взаємодії з датчиками можна використовувати єдиний інтерфейс, тоді написання алгоритмів роботи з кожним датчиком окремо не потрібно.

Загальна структура шаблону показана малюнку 6.1. У цьому прикладі конкретна фабрика (Concrete Factory) – це певна конфігурація СУ, а конкретні продукти (Concrete Product) – датчики напруги, кутової швидкості, кута, вологості, температури і тиску. Сама конфігурація датчиків СУ може бути сформована шляхом виклику методів factoryMethod() зі своєю конкретною реалізацією в ConcreteFactory1 або ConcreteFactory2.

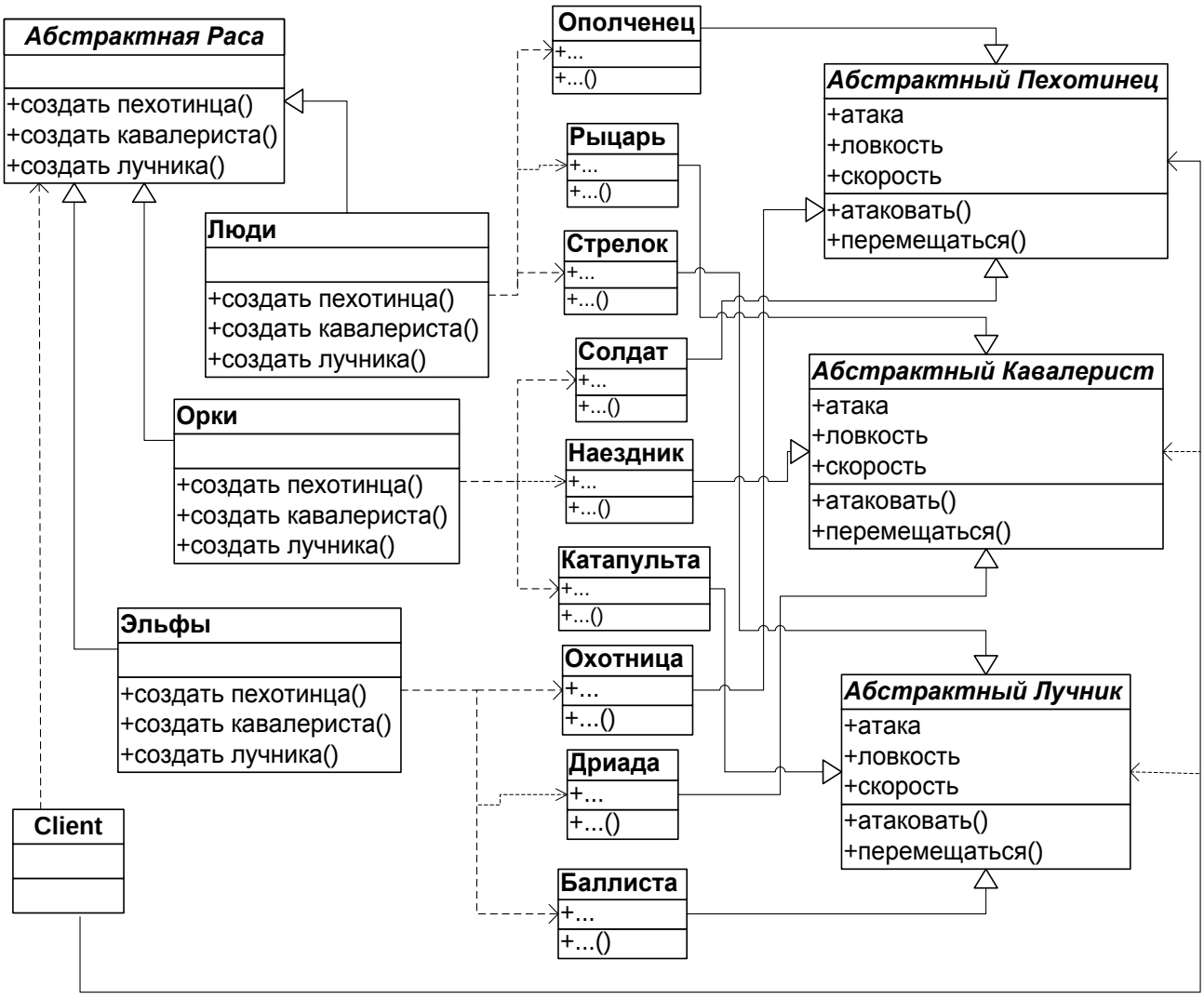


Малюнок 6.1 – Структура шаблону «Абстрактна фабрика»

Для реалізації каркасу шаблону слід створити два класи – абстрактну фабрику та абстрактний продукт. Для його реального застосування необхідно наповнити систему набором конкретних фабрик та конкретних продуктів, які створюватимуться цими фабриками при виклику фабричних методів.

У природі можна знайти чимало аналогів шаблону «Абстрактна фабрика». Наприклад, це може бути бджолиний вулик, де абстрактний продукт – бджола, а абстрактна фабрика – бджолина матка. У кожному вулику є своя матка (конкретна фабрика). Вона формує рій своїм набором конкретних бджіл - бджіл-стражників, бджіл-робітників або бджіл, що годують.

Ще один приклад абстрактної фабрики можна виявити в комп'ютерній грі War Craft, де присутні три раси – люди, орки та ельфи. При цьому в кожній расі зустрічаються типові бойові одиниці - піхотинець, кавалерія і лучник, що мають різні параметри сили атаки, дальність ураження і швидкість переміщення. У такій структурі аналогом фабрики є раса, а конкретними продуктами – певні види бойових одиниць. Докладніше структура показана малюнку 6.2. Кожна з рас – конкретна фабрика (саме відповідає за створення певних бойових одиниць у грі), а бойова одиниця конкретної раси – конкретний продукт.



Малюнок 6.2 – «Абстрактна фабрика» у комп'ютерній грі War Craft

Введення в гру нової раси передбачає написання нового класу конкретної фабрики та набору конкретних класів піхотинця, кавалериста та лучника. Наприклад, клас раси «Нежити» і три класи конкретних продуктів – скелет (успадковується від піхотинця), павук (успадковується від кавалериста) та некромант (успадковується від лучника).

Контрольні питання

Перерахуйте класи, що входять до структури шаблону «Абстрактна фабрика».

Наведіть приклад аналога шаблону «Абстрактна фабрика» у природній, соціальній та технічній сферах.

З якою метою створюють абстрактні класи фабрик та продуктів? Чому не обмежуються створенням класів конкретних фабрик та продуктів?

Яка перевага дає використання інтерфейсів абстрактних класів при створенні та роботі з великою кількістю екземплярів їх конкретних нащадків?

Яка вимога висувається до класів конкретних продуктів та конкретних фабрик шаблону «Абстрактна фабрика»?