Design Patterns GoF

Behavioral Patterns

-Template Method Defer the exact steps of an algorithm to a subclass

-Interpreter A way to include language elements in a program

-Chain of Resp. A way of passing a request between a chain of objects

-Command Encapsulate a command request as an object

-Iterator Sequentially access the elements of a collection

-Mediator Defines simplified communication between classes

-Memento Capture and restore an object's internal state

-Observer A way of notifying change to a number of classes

-State Alter an object's behavior when its state changes

-Strategy Encapsulates an algorithm inside a class

Visitor Defines a new operation to a class without change

Означення

Патерн проектування:

1) опис *взаємодії об'єктів і класів*,
2) розроблених для *вирішення загальної задачі* проектування
3) в конкретному *контексті*

??? Behavioral Patterns ???

Патерн Template Method

- Назва та класифікація

  Template Method (Шаблонний метод) – патерн поведінки **класів**

- Призначення

  Визначає основу алгоритму (послідовність кроків) і надає похідним класам можливість перевизначати окремі кроки алгоритму, залишаючи незмінною його структуру в цілому

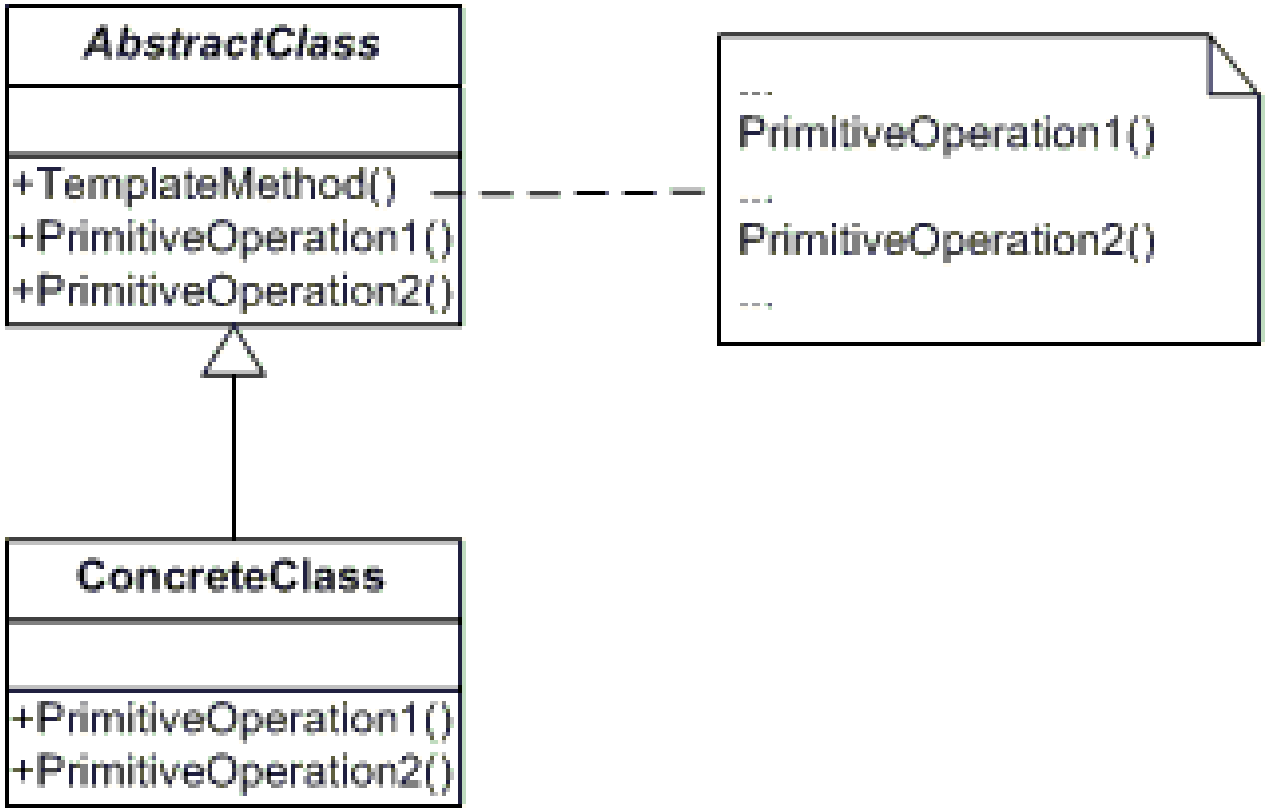
- Мотивація

  Приклад: ***Document*** має методи відкриття і зчитування документу з файла, ***Application::OpenDocument*** може викликати їх, якщо попередні кроки щодо перевірки дозволу на відкриття, зчитування файла, утворення об'єкта документу були успішними.

Template Method

- Застосування
  - **Інваріантна поведінка** для усієї ієрархії класів виокремлюється та задається у методах базового класу (уникаємо дублювання коду)
  - **Варіації поведінки** на окремих кроках алгоритму реалізуть окремі методи (можуть бути закритими+віртуальними), які перевизначають у похідних типах
  - **Керування розширенням** похідних класів за рахунок зачіпок(**hooks**): шаблонний метод викликатиме зачіпки лише у фіксованих точках, лише в них дозволяючи розширення

Template Method



Template Method

- **Учасники**:
  - **AbstractClass**
    - — defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
    - — implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
  - **ConcreteClass**
    - — implements the primitive operations ot carry out subclass-specific steps of the algorithm
- **Відношення**

Передбачається, що інваріантна дія виконується методами базового абстрактного класу, а відмінна – методами похідних типів, які реалізують відповідні примітивні операції

TemplateMethodClassic
TemplateMethodCPP

Template Method Результати

Шаблонні методи можуть викликати такі операції:

- конкретні операції (методи **ConcreteClass** або класів клієнта)
- конкретні операції-методи **AbstractClass** (інваріанти для всіх класів)
- примітивні (абстрактні, віртуальні) операції
- фабричні методи
- операції-зачіпки(hooks), які реалізують поведінку за замовчуванням, що може розширюватися у похідних класах

Template Method Реалізація

- використання контролю доступу(в C++): примітивні операції, які викликає шаблонний метод, оголошують закритими або захищеними (виклик можливий лише з шаблонного методу)
- примітивні операції, які обов’язково перевизначають, оголошують як чисто віртуальні; сам шаблонний метод може бути невіртуальним
- при проектуванні важливою ціллю є скорочення кількості примітивних операцій, які необхідно перевизначати в похідних типах
- погодження назв: ідентифікаторам примітивних операцій, які необхідно перевизначати в похідних типах, додавати деякий префікс, напр.,
  void doPh2();

Патерн Inerpreter

- Назва та класифікація

  Inerpreter(Інтерпретатор) – патерн поведінки **класів**

- Призначення

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

- Мотивація

  Якщо якась задача виникає досить часто, є сенс подати її конкретні прояви у вигляді речень простою мовою. Потім можна буде створити інтерпретатор, котрий розв’язує задачу, аналізуючи речення цієї мови. Кожне правило граматики подають окремим класом, символи у правій частині правила – об’єкти таких класів. Наприклад, пошук рядків за зразком – досить розповсюджена задача, тому регулярні вирази – це стандартна мова для задання зразків пошуку.

Inerpreter

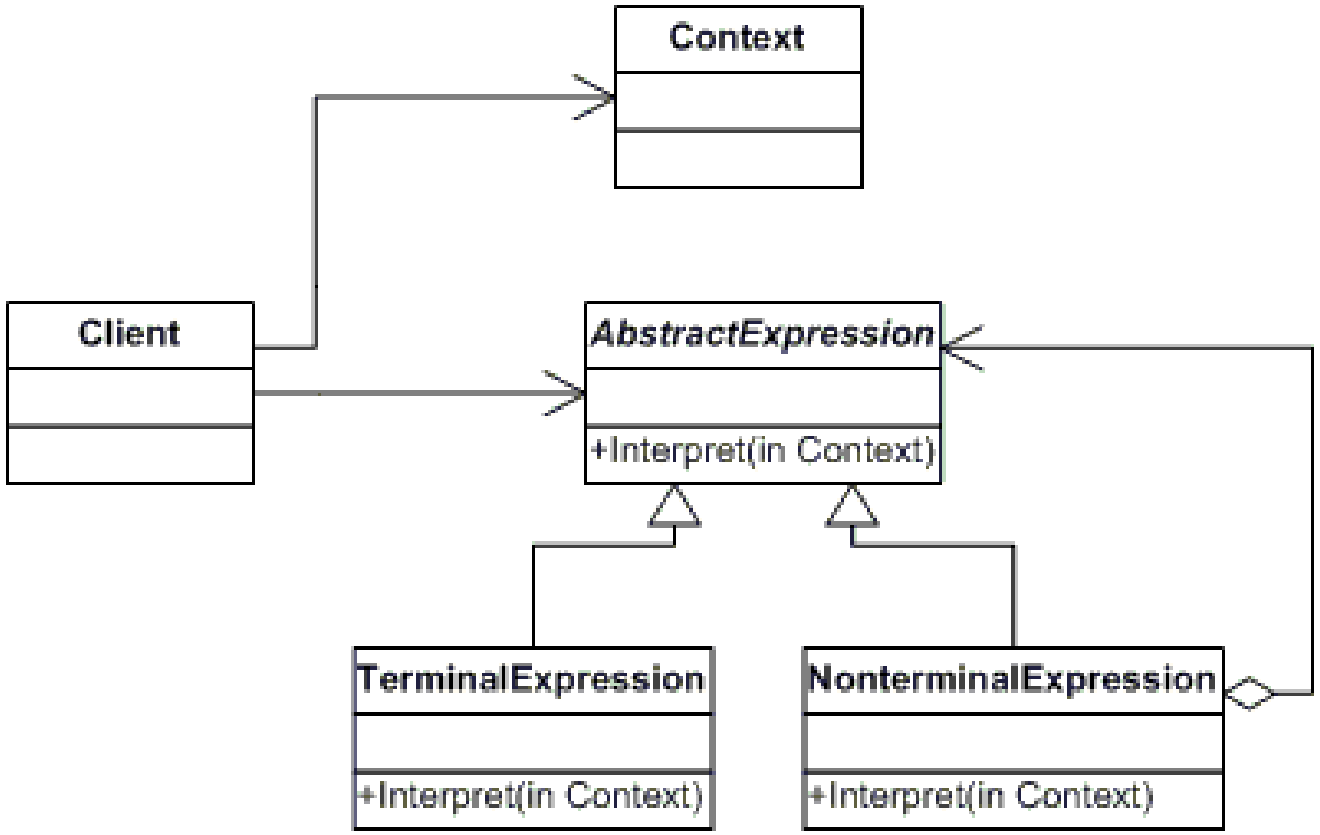
- Застосування

  для інтерпретації сформованих деякою мовою речень, які можна подати у вигляді абстрактних синтаксичних дерев, коли:

  – граматика проста

  – ефективність – не головний критерій: найбільш ефективні інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму, напр., регулярний вираз перетворюють на скінченний автомат

Inerpreter Структура



InterpreterClassic
InterpreterRomanNum

Inerpreter Учасники

- **AbstractExpression**
  - оголошує абстрактну операцію **Interpret**, загальну для усіх вузлів у абстрактному синтаксичному дереві;
- **TerminalExpression**
  - реалізує операцію **Interpret** для термінальних символів граматики;
  - для кожного термінального символу у реченні необхідний окремий екземпляр;
- **NonterminalExpression**
  - для *кожного* граматичного правила *по одному* такому класу;
  - зберігає змінні екземпляру типу **AbstractExpression** для кожного символу;
  - реалізує операцію **Interpret** для нетермінальних символів граматики, яка рекурсивно викликає себе для усіх компонентів
- **Context**
  - містить інформацію, глобальну по відношенню до інтерпретатора
- **Client**
  - будує (або отримує готове) абстрактне синтаксичне дерево для речення
  - викликає операцію **Interpret**

Inerpreter Відношення

- клієнт будує (або отримує у готовому вигляді) речення у вигляді абстрактного синтаксичного дерева, вузли якого містять об'єкти класів **NonterminalExpression** та **TerminalExpression**. Далі клієнт ініціалізує контекст та викликає операцію **Interpret**;
- у кожному вузлі виду **NonterminalExpression** через операцію **Interpret** визначають операцію **Interpret** для кожного підвиразу. Для класу **TerminalExpression** операція **Interpret** визначає базу рекурсії;
- операції **Interpret** у кожному вузлі використовують контекст для зберігання та доступу до стану інтерпретатора.

Патерн Mediator

- Назва та класифікація

Mediator (Посередник) – патерн поведінки **об'єктів**

- Призначення

Визначає об’єкт, який інкапсулює спосіб взаємодії множини інших об'єктів, забезпечуючи при цьому їх слабку зв’язність

- Мотивація

Якщо розподіл деякої поведінки системи об’єктів приводить до великої кількості зв’язків між ними, то система стає монолітом: окремі класи важко повторно використовувати і складно змінювати алгоритм їхньої взаємодії. Цих проблем можна уникнути, якщо інкапсулювати колективну поведінку в окремому об’єкті-посереднику:

- відповідає за координацію взаємодії об’єктів
- позбавляє об’єкти необхідності посилань один на одного

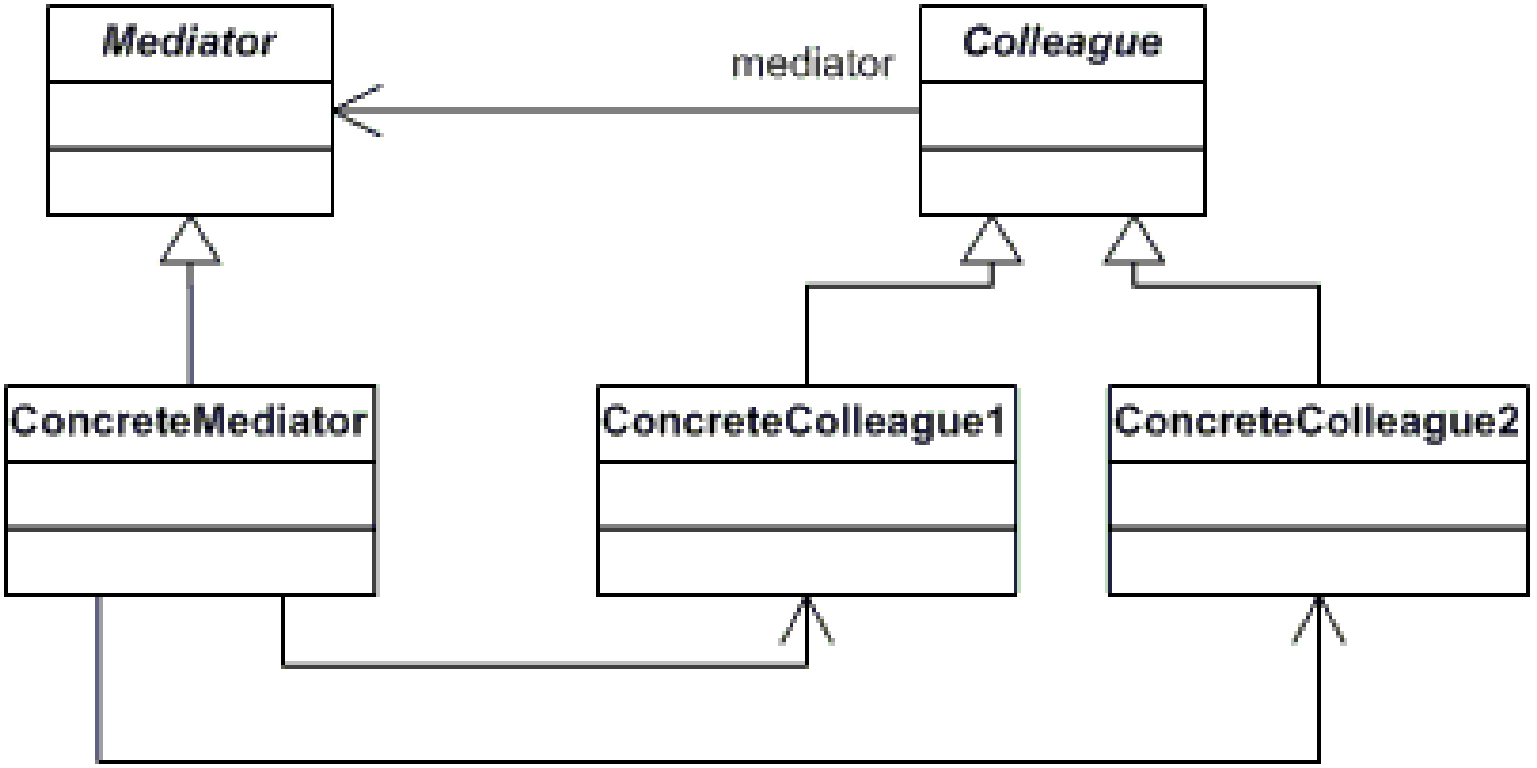
Mediator Застосування

- Є об’єкти, зв’язки між якими чітко визначені, але складні, не структуровані і важко зрозумілі
- Неможливо повторно використовувати об’єкт, оскільки він обмінюється даними з багатьма іншими об’єктами
- Треба налаштовувати розподілену між класами поведінку без породження похідних класів (підкласів)

Приклад: взаємодія в діалоговому вікні для вибору файлів розміщених у ньому віджетів: список папок, список файлів, редактор фільтру(розширення назви), редактор назви файла вікно про зміну свого стану. Кожен віджет повідомляє вікно про зміну свого стану, не знаючи про існування інших віджетів.

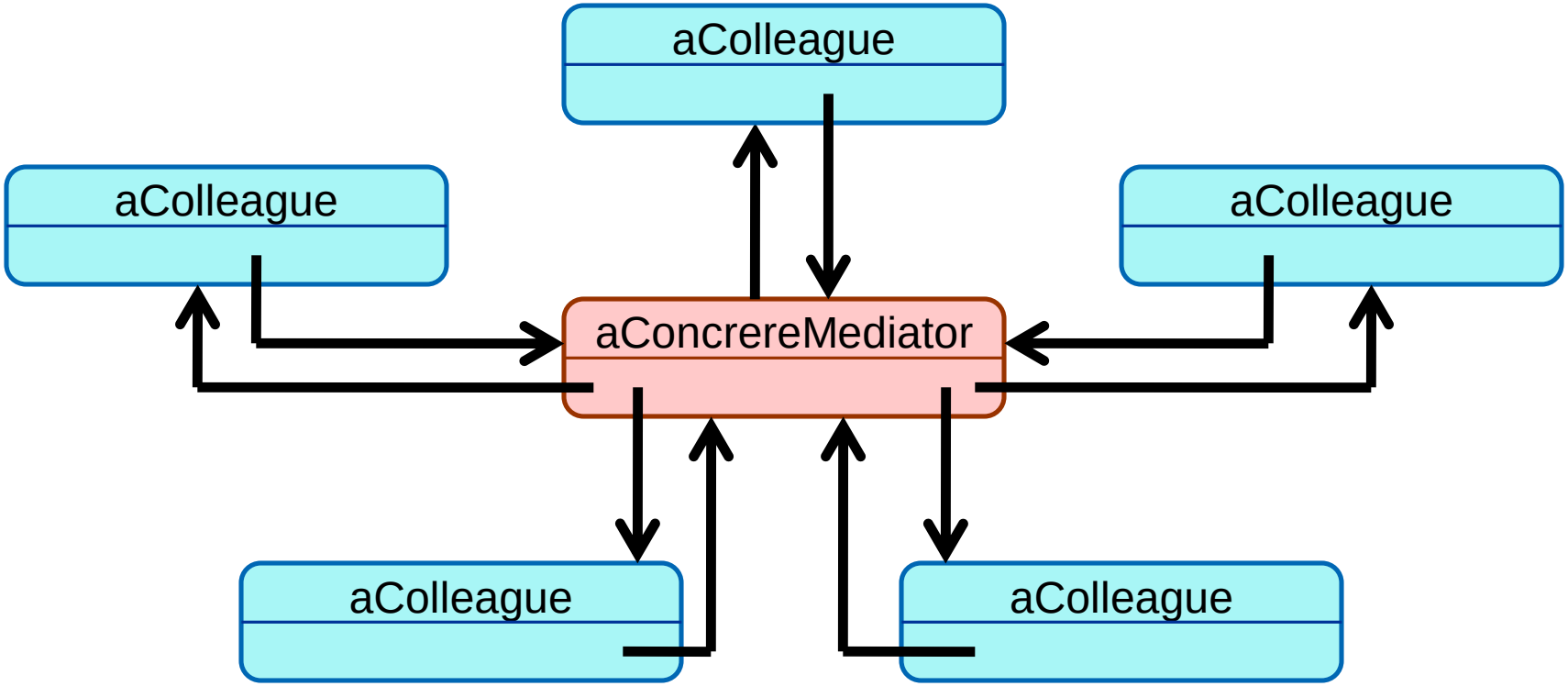
Mediator

Mediator



MediatorClassic
MediatorFileSelection

Mediator


aColleague
aColleague
aColleague
aConcrereMediator
aColleague
aColleague

Mediator

- **Учасники**:
  - **Mediator** (**IChatroom**)
    - визначає інтерфейс для комунікації з об’єктами **Colleague**
  - **ConcreteMediator** (**Chatroom**)
    - реалізує кооперативну поведінку, координуючи об’єкти **Colleague**
    - знає і підтримує взаємодію своїх колег
  - **Colleague classes** (**Participant**)
    - кожен об’єкт класу **Colleague** знає об’єкт класу **Mediator**
    - об’єкти **Colleague** взаємодіють між собою через посередника
- **Відношення** Колеги посилають запити посередникові та отримують запити від нього. Посередник реалізує кооперативну поведінку шляхом переадресації кожного запиту відповідному колезі (або декільком з них).

Mediator Результати

- Зменшує кількість похідних класів: для заміни поведінки достатньо утворити похідний тип від посередника, а класи колег залишаються без змін.
- Позбавляє колег зв'язності – посередник забезпечує між ними слабку зв'язність
- Спрощується протокол взаємодії колег: посередник замінює складне відношення "усі-до-всіх" на простіше "один-до-всіх"
- Абстрагує спосіб кооперації об'єктів: інкапсуляція виділеної взаємодії в одному об'єкті робить наголос на взаємодії, а не на індивідуальній поведінці окремих об'єктів; у результаті проясняється взаємодія
- Централізує управління: оскільки посередник інкапсулює протоколи взаємодії, то він може бути складнішим об'єктів-колег і перетворитися на моноліт, непридатний для супроводу і повторного використання

Mediator Реалізація

- Якщо об’єкти-колеги працюють лише з одним посередником, то не треба оголошувати абстрактний **Mediator**
- Обмін інформацією можна задавати різними способами. Один з найефективніших – використання патерну Observer:
    - колеги індивідуально надсилають посереднику повідомлення про зміну їхнього стану
    - посередник повідомляє про це потрібних колег згідно алгоритму взаємодії

Патерн Observer

- Назва та класифікація

Observer (Спостерігач) – патерн поведінки об'єктів

- Призначення

Визначає залежність “один-до-одного” між об’єктами, при якій у випадку зміни стану одного об’єкта залежні від нього об'єкти повідомляються про це і автоматично оновлюються

- Мотивація

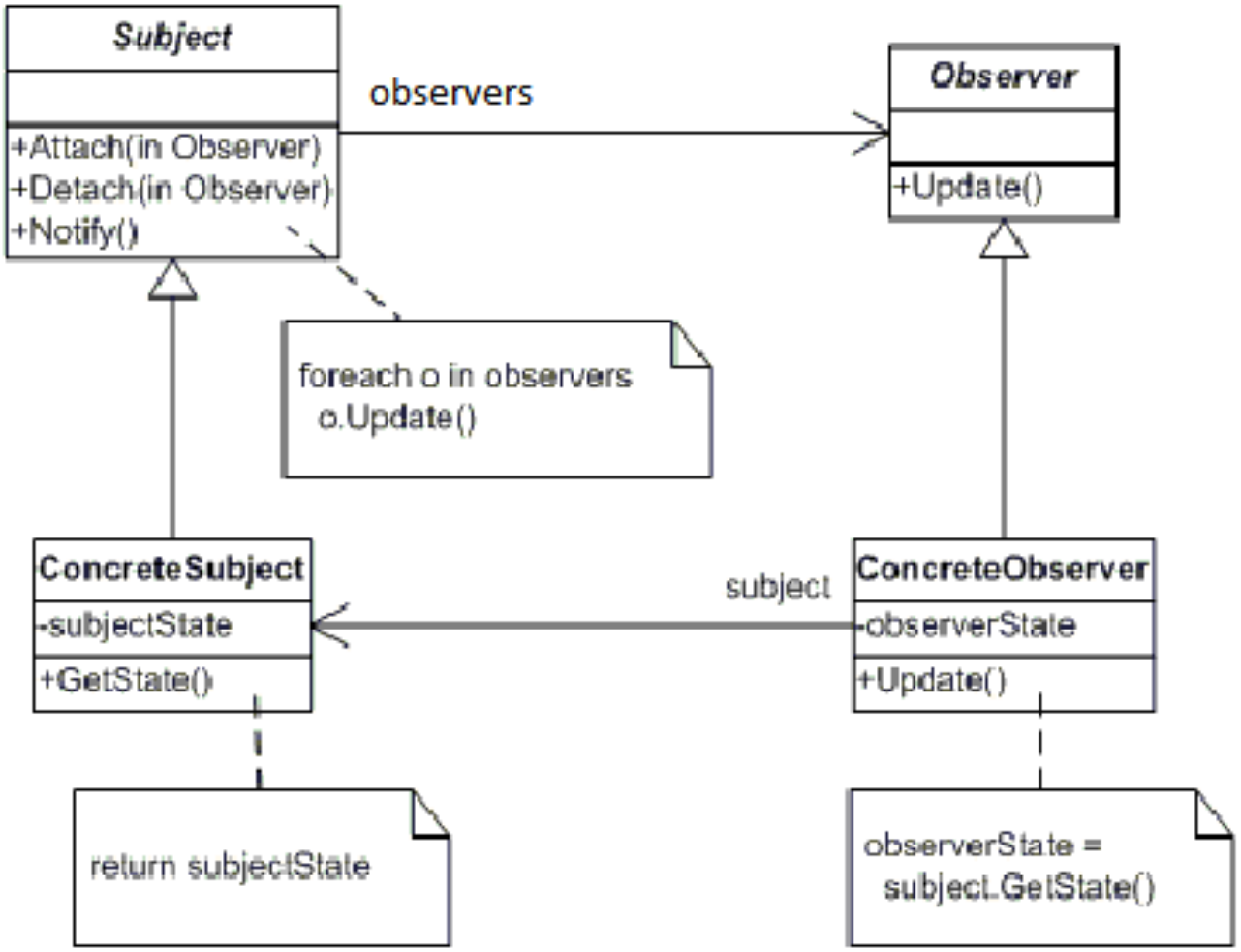
Якщо розподіл деякої поведінки системи об’єктів приводить до великої кількості зв’язків між ними, то система стає монолітом: окремі класи важко повторно використовувати і складно змінювати алгоритм їхньої взаємодії. Цих проблем можна уникнути, якщо інкапсулювати колективну поведінку в окремому об’єкті-посереднику:

- відповідає за координацію взаємодії об’єктів
- позбавляє об’єкти необхідності посилань один на одного

Observer Застосування

- Абстракція стосується двох аспектів, один з яких залежить від іншого; інкапсуляція їх у два різні об’єкти дає змогу незалежно їх змінювати і повторно використовувати
- Модифікація одного об’єкта вимагає зміни інших об’єктів, але невідомо, скільки їх є
- Один об’єкт має сповістити інші об’єкти без жодних припущень щодо них, тобто без потреби бути тісно пов’язаними

Observer



Observer_MVC
StockObserverEventDr

Observer Учасники

- **Subject**
  - має в розпорядженні інформацію про своїх спостерігачів, кількість яких може динамічно змінюватися
  - надає інтерфейс для приєднання і від'єднання спостерігачів
- **Observer**
  - визначає інтерфейс оновлення для об'єктів, які отримують повідомлення про зміни суб'єкта
- **ConcreteSubject**
  - зберігає стан, параметрами якого цікавляться спостерігачі
  - надає інформацію про параметри стану спостерігачам, коли відбувається зміна стану
- **ConcreteObserver**
  - зберігає посилання на конкретного суб'єкта
  - зберігає дані, які повинні узгоджуватися з суб'єктом
  - реалізує інтерфейс **Observer** для оновлення узгоджених із суб'єктом даних

Observer Результати

- **Абстрактна з’язність суб’єкта і спостерігача:**
  - суб’єктові відомо, що в нього є спостерігачі з інтерфейсом абстрактного класу і невідомі конкретні типи спостерігачів
  - можуть перебувати на різних рівнях абстракції системи: суб’єкт нижчого рівня може повідомляти спостерігачів верхніх рівнів, не порушуючи ієрархії системи
- **Підтримка широкої смуги комунікації:**
  - повідомлення автоматично надсилається усім підписаним на нього спостерігачам, кількість яких в довільний момент може незалежно змінюватися
  - спостерігач сам вирішує, чи повинен він обробляти отримане повідомлення, чи може його ігнорувати
- **Неочікувані оновлення:**
  - оскільки спостерігачі не мають інформації один про одного, то не відомо, в що сумарно обходиться оновлення суб’єкта – операція над ним може спричинити цілий ряд оновлень спостерігачів та залежних від них об’єктів
  - нечітко визначені критерії залежності можуть спричинити неочікувані оновлення, які складно відслідкувати

Observer
Реалізація механізму залежностей – I

- **Відображення суб’єктів на спостерігачів:**
  - суб’єкт зберігає явні посилання на спостерігачів, яким він надсилає повідомлення
  - зростання накладних затрат при великій кількості суб’єктів і кількох спостерігачах
- **Спостереження за кількома суб’єктами:** інтерфейс **Update()** розширюють, напр., додаючи як аргумент посилання на суб’єкт
- **Висячі посилання на знищені суб’єкти:** перед знищенням суб’єкт має посилати повідомлення про це спостерігачам

Observer

Реалізація механізму залежностей – II

- **Хто ініціює оновлення** (2 варіанти):
  - операції **Subject**, які змінили стан, викликають **Notify()**: перевага – спостерігачам не треба пам’ятати про її виклик, недолік – повідомлення при виконанні кожної з послідовних операцій може привести до неефективного виконання програми
  - відповідальність за своєчасний виклик **Notify()** на клієнті: перевага – клієнт може відкласти ініціалізацію оновлення до завершення певної серії змін суб’єкта, недолік – додаткове навантаження на клієнта

Observer
Реалізація механізму залежностей – III

- **Гарантія несуперечливості стану суб'єкта перед надсиланням повідомлення:**
  - може порушуватися за рахунок виклику операцій базових класів
  - вирішення – патерн **Шаблонний метод** у абстрактному класі **Subject**, коли примітивна операція перевизначається у похідному класі
- **Залежність протоколу оновлення:**
  - ***push model***: **Subject** надсилає спостерігачам детальну інформацію про зміни незалежно від їхніх потреб; акцент на інформованості типу **Subject** про тип спостерігачів **Observer**, імовірність повторного використання **Subject** зменшується
  - ***pull model***: **Subject** надсилає об'єктам **Observer** мінімальну інформацію про факт зміни, а вони звертаються за потрібною інформацією до **Subject**; акцент на витяганні спостерігачами додаткової інформації від суб'єкта про його новий стан

Observer

Реалізація механізму залежностей – IV

- **Явна специфікація модифікацій, які цікавлять спостерігача:** коли зазначені модифікації відбудуться, суб'єкт посилатиме повідомлення лише тим спостерігачам, які проявили при реєстрації до них інтерес

```
void Subject::Attach(Observer*,Aspect& interest);

void Observer::Update(Subject*,Aspect& interest);
```

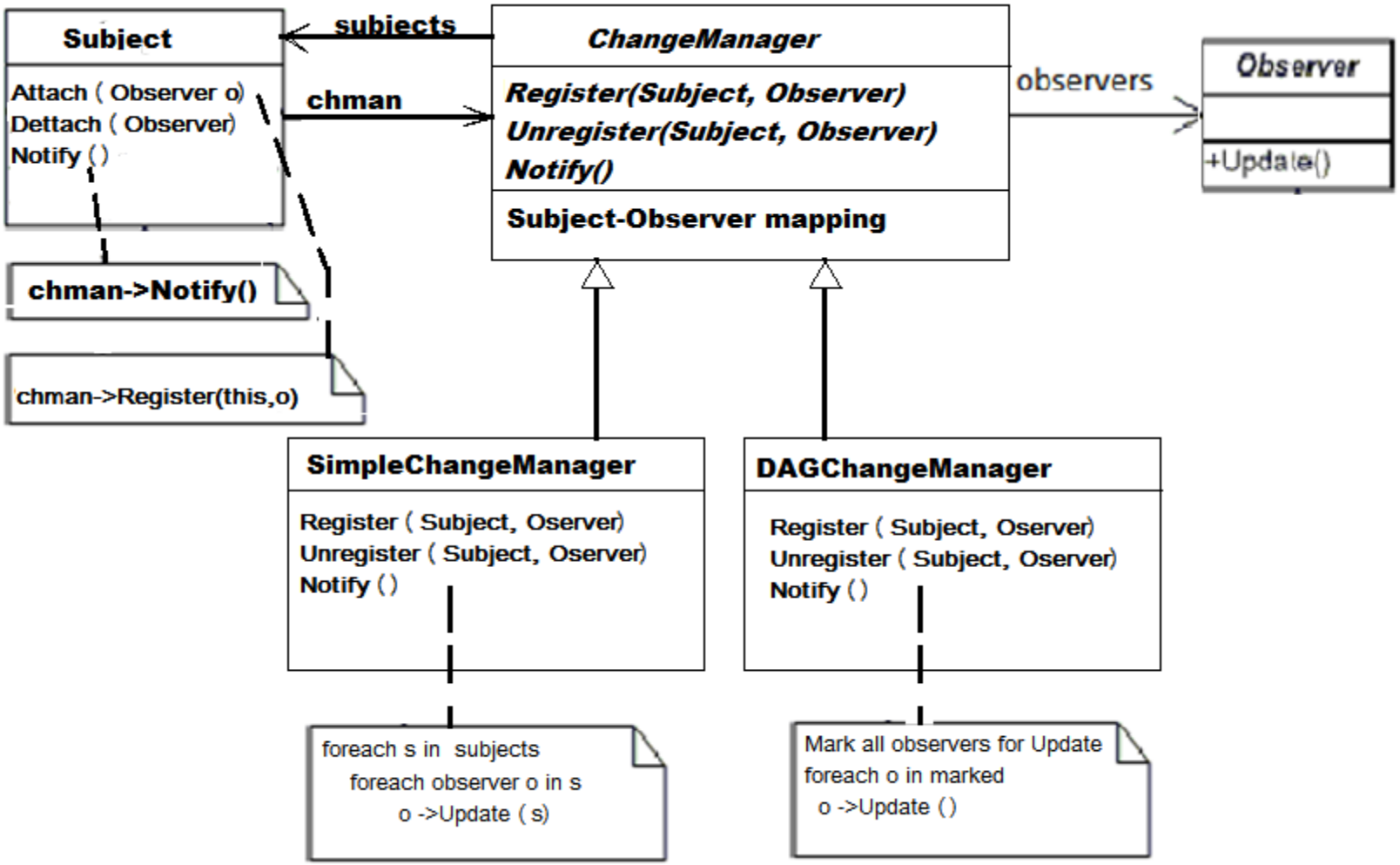
Observer

Реалізація механізму залежностей – V

- **Інкапсуляція складної семантики оновлення:** при складному відношенні між суб'єктом і спостерігачами вводиться додатковий об'єкт(**Mediator-Singleton**):
  - мінімізує кількість дій, необхідних для відображення спостерігачами змін у суб'єкті
  - будує відображення між суб'єктом і спостерігачами та надає інтерфейс для підтримки цього відображення в актуальному стані
  - дає змогу суб'єктам позбутися посилань на своїх спостерігачів і навпаки
  - визначає конкретну стратегію оновлення
  - оновлює усіх залежних спостерігачів за запитом суб'єкта

    SimpleChangeManager завжди оновлює усіх залежних спостерігачів кожного суб'єкта
    DAGChangeManager опрацьовує направлені ациклічні графи залежностей між суб'єктами та їхніми спостерігачами і гарантує, що при зміні кількох суб'єктів відповідний спостерігач отримає лише одне сповіщення

Observer


Subject
Attach (Observer o)
Dettach (Observer)
Notify ( )
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject-Observer mapping
observers
Observer
+Update()
chman->Notify()
chman->Register(this,o)
SimpleChangeManager
Register (Subject, Oserver)
Unregister (Subject, Oserver)
Notify ( )
DAGChangeManager
Register (Subject, Oserver)
Unregister (Subject, Oserver)
Notify ( )
foreach s in subjects
foreach observer o in s
o ->Update (s)
Mark all observers for Update
foreach o in marked
o ->Update ( )

.NET Framework 4

IObserver & IObservable

```
public interface IObserver<in T>
{
     void OnNext( T value );
     void OnCompleted();
     void OnError( Exception error );
}
public interface IObservable<out T>
{
   IDisposable Subscribe(IObserver<T> observer);
}
```

Патерн
Chain of Responsibility

- Назва та класифікація

  Chain of Responsibility – патерн поведінки

- Призначення

  Дає змогу уникнути прив'язки відправника запиту до його конкретного обробітника; запит передається вздовж ланцюжка об'єктів, поки не трапиться об'єкт, який може обробити запит

- Мотивація

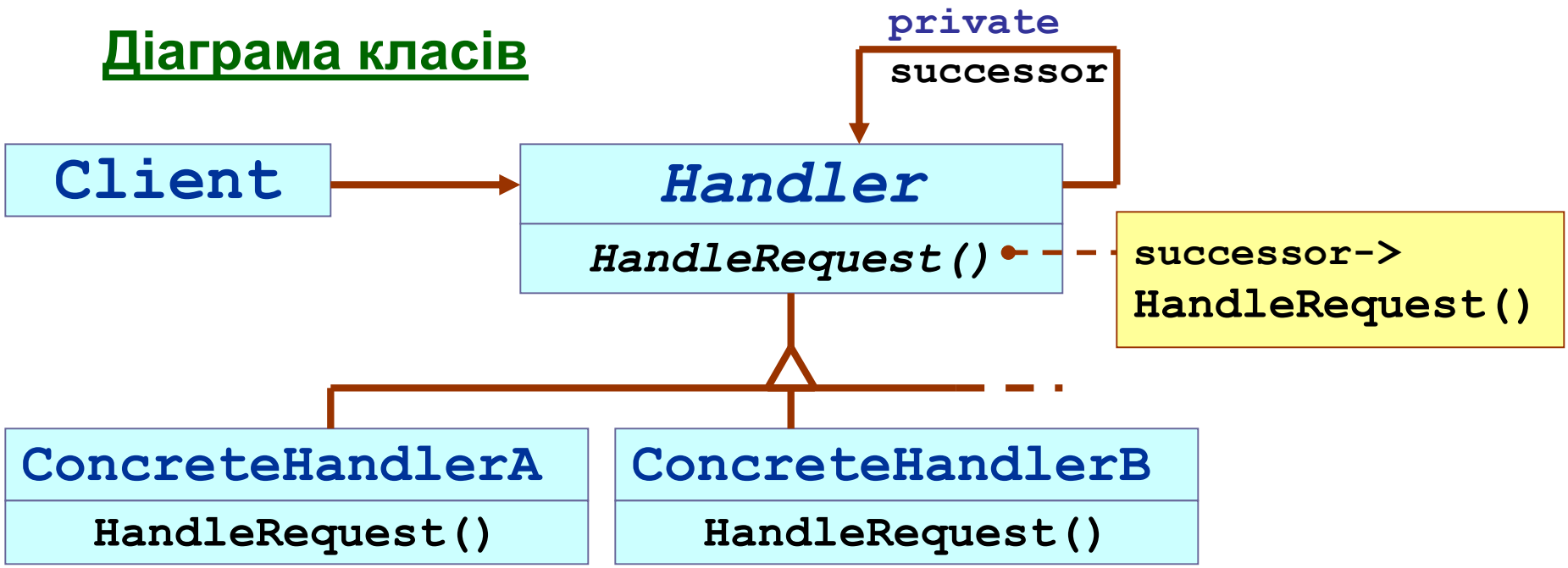
  У момент ініціалізації запиту клієнтові не відомо, який конкретний об'єкт(implicit receiver) може обробити запит. Щоб гарантувати переміщення запиту вздовж ланцюжка об'єктів, усі вони повинні підтримувати єдиний інтерфейс виклику методу-обробітника. Приклад – довідкова система до елементів GUI.

Chain of Responsibility Застосовність

- існує більше одного об'єкта, здатного обробити запит, причому конкретний обробітник наперед невідомий і підбирається автоматично
- треба відправити запит одному з кількох об'єктів, не вказуючи його конкретно
- набір об'єктів-обробітників задається динамічно

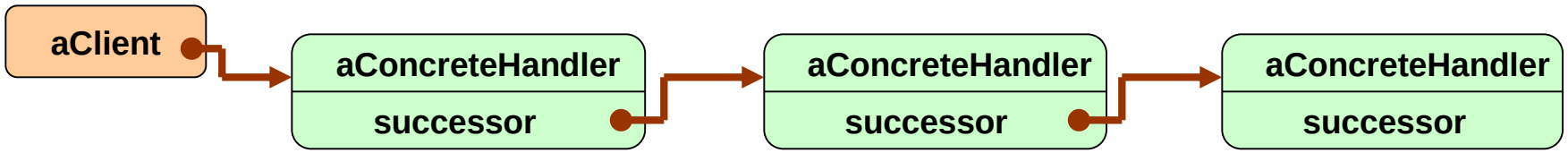
Chain of Responsibility Структура

Діаграма класів



Діаграма об'єктів

ChainofResponsibilityClassic
ChainofResposibilityCircle



Chain of Responsibility Учасники

- ***Handler*** – обробітник
  - задає спільний інтерфейс для обробки запитів;
  - (необов'язково) реалізує зв'язок з наступником
- **ConcreteHandlerA, ConcreteHandlerB, …** – похідні класи ***Handler***, які реалізують конкретних обробітників
  - мають доступ до свого наступника
  - якщо відповідають за запит, то обробляють його, інакше передають запит наступному об'єкту у списку
- **Client** – клієнт
  - робить запит до об'єкта

Chain of Responsibility Відношення

- Ініційований клієнтом запит передається об'єктами вздовж ланцюжка, поки не трапиться один з них, який може обробити запит
- якщо такий об'єкт не трапиться, то запит пропадає

Chain of Responsibility Результати

- **Послаблення зв'язності:**
  - Об'єкт-ініціатор запиту не знає, який об'єкт його опрацює цей запит
  - відправнику і кінцевому отримувачу нічого невідомо один про одного
  - елементи ланцюжка знають лише свого безпосереднього наступника, їм не потрібно знати ні конкретного обробітника, ні всієї множини обробітників
- **Додаткова гнучкість при розподіленні обов'язків між об'єктами:** Ланцюжок легко модифікувати, додаючи чи вилучаючи відповідні об'єкти з ланцюжка
- **Виконання запиту не гарантоване:**
  - немає гарантії, що потрібний обробітник трапиться в ланцюжку
  - запит може досягти кінця ланцюжка і пропасти

Chain of Responsibility Реалізація

- **Реалізація ланцюжка:**
  - визначення нових зв'язків за рахунок вказівника в базовому класі ***Handler***
    - якщо класи предметної області не передбачають таких даних
    - дані в класах не дають змогу визначити потрібний ланцюжок
    - щоб ввести обробітник за замовчуванням; тоді його перевизначають лише у тих похідних типах, де це потрібно
  - використання існуючих зв'язків:
    - якщо вони підтримують ланцюжок, наприклад, у відношенні *"частина-ціле"*
    - уникання надмірних зв'язків
- **Реалізація запитів:**
  - в ієрархії класів для цього відводять окремий метод
  - лише одна функція для ієрархії, яка як параметр отримує код запиту
    - число або рядок
    - об'єкт спеціального типу, який відповідатиме за диспетчеризацію

Патерн Command

- Назва та класифікація

  Command – патерн поведінки

- Призначення

  Інкапсулює запит як об'єкт, який може зберігати параметри клієнтів для обробки відповідних запитів, ставити запити в чергу чи протоколювати їх, а також підтримувати відміну операцій

- Відомий також під іншою назвою

  Action(дія), **Transaction**(транзакція)

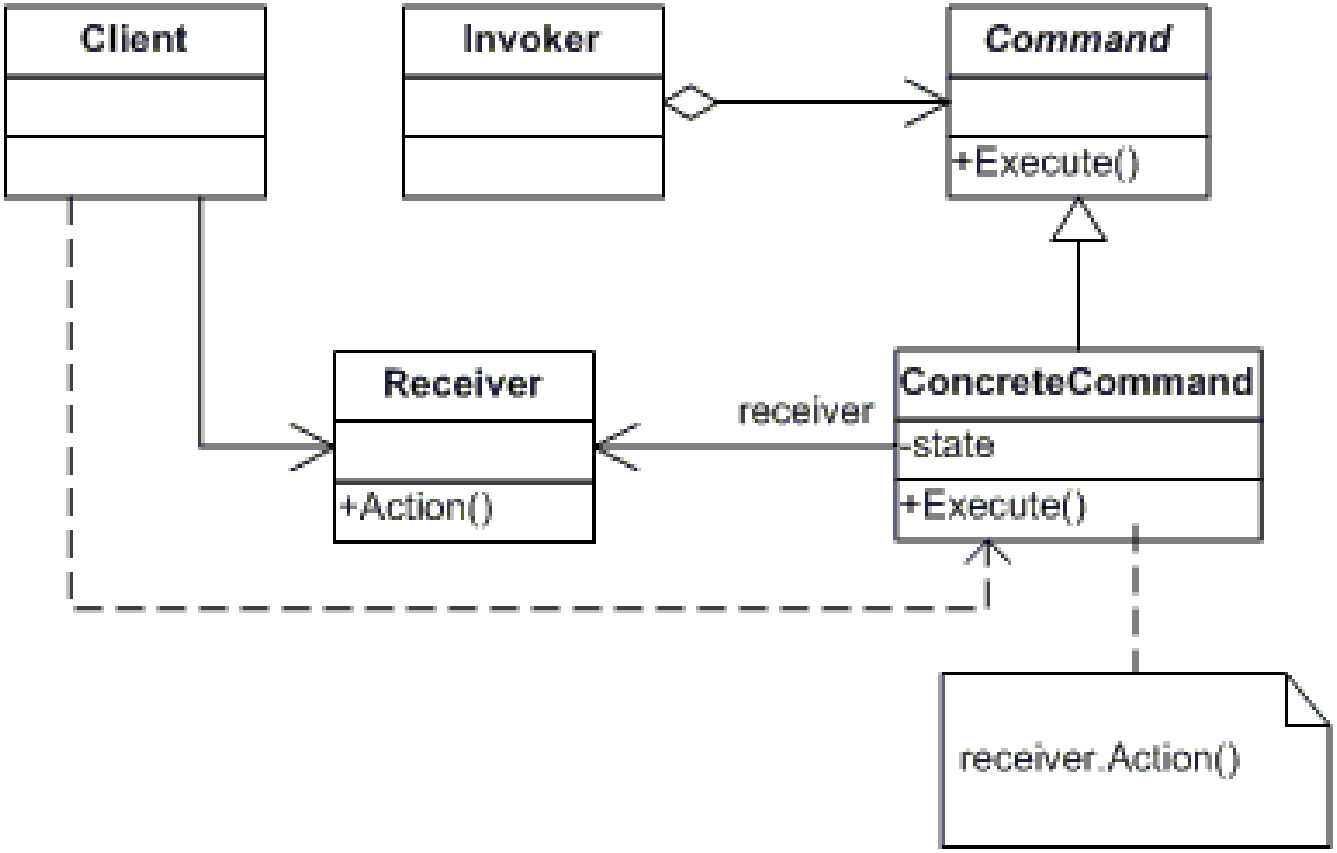
- Мотивація

Відокремлення об’єкта, який ініціює операцію, від об’єкта, який знає, як її виконати. Сам запит перетворюється в об’єкт, який можна зберігати і передавати

Command Застосовність

- параметризація об'єкта щодо виконуваної дії; об'єктно-орієнтована альтернатива (з відмінностями) функцій зворотнього виклику (callback function)
- визначення, постановка в чергу і виконання запитів у різний час
- структурування системи за допомогою високорівневих операцій (макрокоманд)
- підтримка відміни операції

Command Структура



CommandClassic
CommandDoUndoRedo

Command Учасники

- ***Command*** – команда
  - задає інтерфейс для виконання операції;
- **ConcreteCommand** – похідні класи ***Command*** , які реалізують конкретні команди
  - визначають зв'язок між об'єктом-отримувачем **Receiver** і дією
  - реалізує операцію **Execute** шляхом виклику відповідних операцій отримувача
- **Client** – клієнт
  - робить запит до об'єкта
- **Invoker** – ініціатор
  - звертається до команди для виконнання запиту
- **Receiver** – отримувач
  - володіє інформацією про способи виконання операцій, необхідних для задоволення запиту
  - в ролі отримувача може виступати об’єкт довільного класу

Command Відношення

- клієнт створює об'єкт **ConcreteCommand** і встановлює для нього отримувача
- **Invoker** зберігає об'єкт **ConcreteCommand**
- **Invoker** відправляє запит, викликаючи метод **Execute** команди**;** якщо підтримується режим анулювання виконаних дій(відкат) то **ConcreteCommand** перед викликом **Execute** зберігає інформацію про стан, достатню для виконання відкату
- об'єкт **ConcreteCommand** викликає методи **Receiver** для виконання запиту

Command Результати

- Розрив зв'язку між об'єктом, який ініціює виконання операції, і об'єктом, який має дані і функціональність для її виконання
- Команди – об'єкти : доступні всі технології ООП
- Створення макрокоманд – перехід на функціональність високого рівня
- Просте долучення до програми нових команд – існуючі класи не змінюються

Патерн State

- Назва та класифікація

  State – патерн поведінки

- Призначення

  змінює поведінку (функціональність) об'єкта залежно від його внутрішнього стану; зовні виглядає як зміна типу об'єкта

- Мотивація

  Абстрактний клас (інтерфейс) стану об'єкта визначає вид поведінки; через вказівник на цей клас за допомогою конкретних похідних класів стану деякий контекст визначає поведінку об'єкта вцілому. Об'єкт, який задає стан, може змінювати своє значення, моделюючи зміну типу

State Застосовність

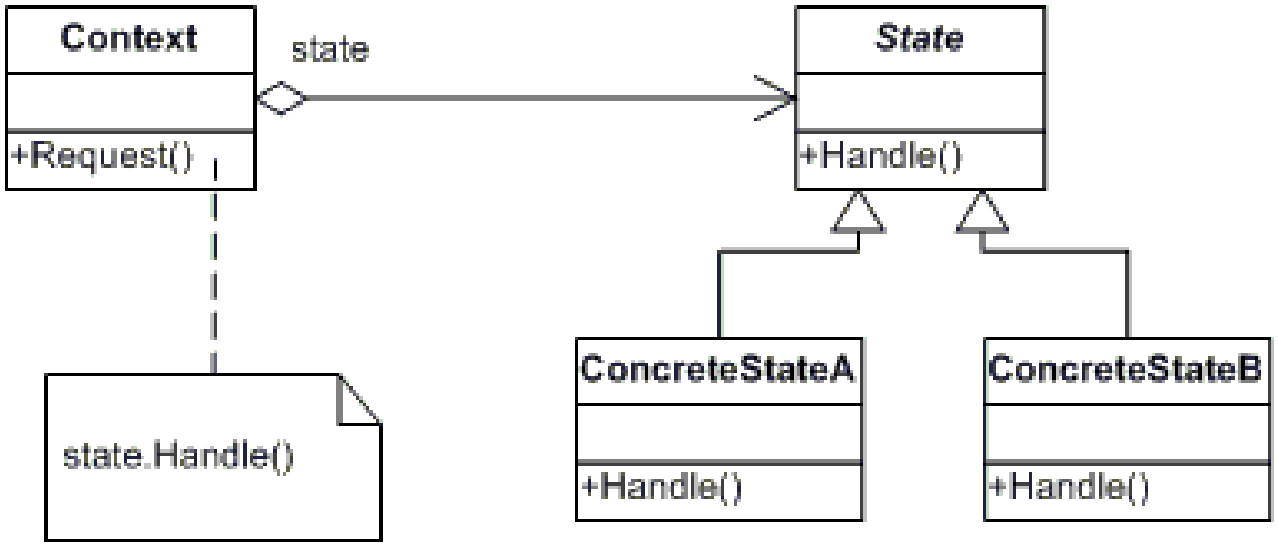
- поведінка об'єкта залежить від його внутрішнього стану і змінюється під час виконання програми
- алгоритми операцій мають багато галужень:
  - вибір вітки виконання алгоритму залежить від стану об'єкта
  - стани можуть ідентифікуватися константами
  - окрему вітку можна помістити в окремий клас, який трактують як самостійний об'єкт

Приклади:

вибір інструменту в графічному редакторі;

реалізація об’єктів протоколу TCP

State Структура



StateClassic
StateFSM

State Учасники

- **Context**
  - задає інтерфейс, який буде використано клієнтами
  - зберігає через вказівник об'єкт класу, похідного від ***State***
- ***State*** – визначає інтерфейс для інкапсуляції поведінки, асоційованої з конкретним станом контексту **Context**
- **ConcreteStateA, ConcreteStateB, …** – похідні класи ***State***, які реалізують поведінку, асоційовану конкретним станом контексту **Context**

State Відношення

- клас **Context** делегує залежні від стану запити поточному об'єкту **ConcreteState**
- клас **Context** може передавати себе як аргумент об'єкту ***State***; у такий спосіб об'єкт стану може отримувати доступ до даних контексту
- клас **Context** визначає основний інтерфейс для клієнтів; клієнт лише раз конфігурує контекст об'єктом стану, після чого клієнт не повинен напряму визначати значення стану
- об'єкти або класу **Context**, або похідних класів **ConcreteStateA**, **ConcreteStateB**, **…** визначають, за яких умов і як саме відбувається зміна станів

State Результати

- Локалізація залежної від стану поведінки:
  - поділ поведінки на частини, які відповідають станам, у вигляді окремих класів
  - щоб додати нові стани і можливі переходи треба долучити нові класи
- Вибір вітки коду, який відповідає конкретному станові:
  - виконується автоматично
  - оптимальний варіант – поліморфізм, без інструкцій вибору (тоді вони були б розкидані по коду контексту)
- Переходи між станами є явними, їх реалізують інструкціями присвоєння з використанням конкретних об'єктів стану.
- Якщо об'єкт стану не містить спеціальних даних, які описують стан, тобто стан ідентифікується лише типом, то різні контексти можуть використовувати один і той же об'єкт стану

State Реалізація

- Скінченні автомати:
  - якщо критерії переходів між станами зафіксовані, то їх можна реалізувати безпосередньо в класі **Context**
  - інакше – децентралізована логіка – похідні класи ***State*** визначають наступний стан, а також момент переходу згідно інтерфейсу, який надає клас **Context**; недолік – виникає залежність між похідними класами ***State*** , оскільки кожен з них повинен знати принаймні ще один інший.
- Переходи можна задавати таблично, відображаючи деякі вхідні дані на переходи між станами
- Утворення та знищення об'єктів стану:
  - коли виникає в них потреба з негайним знищенням після використання
  - утворення зазделегідь і назавжди (якщо утворення/знищення дороге)
- У мовах, які підтримують динамічне наслідування (Self), патерн **State** реалізують напряму

Патерн Strategy

- Назва та класифікація

  Strategy – патерн поведінки

- Призначення

  Визначає сімейство алгоритмів, інкапсулює кожен з них і робить взаємозамінними

- Відомий також під іншою назвою

  Policy – політика

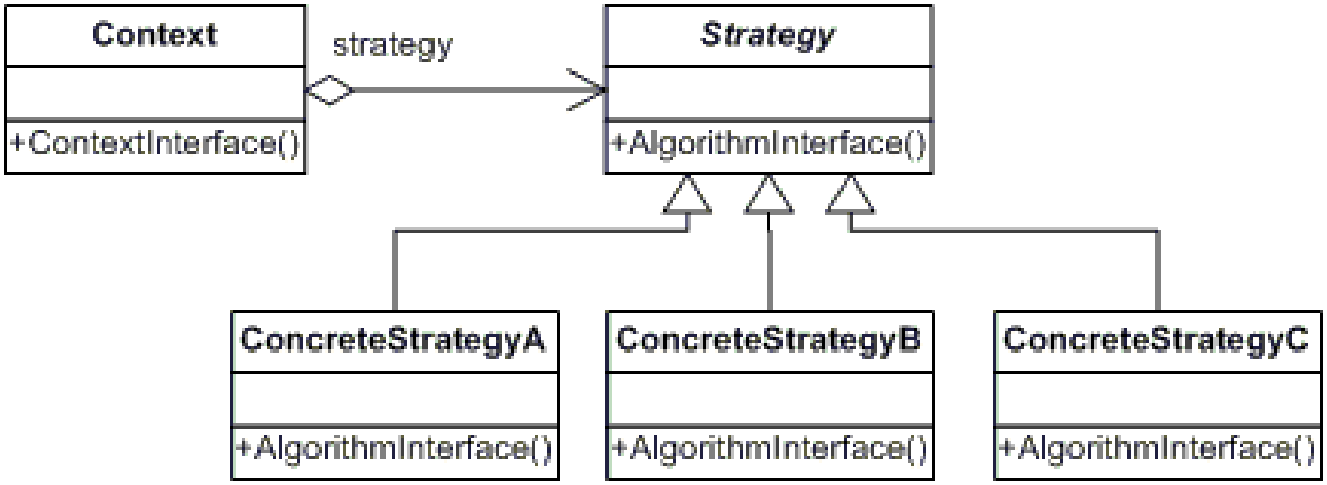
- Мотивація

  Відокремлення змінного коду (конкретних **реалізацій алгоритму**) від постійного (**контексту**, який може передавати потрібні алгоритму дані про стан контексту), що виконує виклик алгоритму. Зв'язування відбувається при виконанні програми.

Strategy Застосовність

- множина подібних типів (однієї ієрархії) відрізняється лише поведінкою об'єктів – тоді об'єкт конфігурують під час виконання програми за допомогою класу, який інкапсулює потрібну поведінку
- алгоритми можуть мати різну реалізацію залежно від контексту їхнього використання
- алгоритми можуть містити дані, які треба приховати від клієнта
- існує багато варіантів поведінки об‘єкта, що може зумовлювати в коді велику кількість інструкцій вибору; альтернатива – інкапсулювання конкретних віток виконання в окремі класи стратегій

Strategy Структура



StrategySortPolymorphism
StrategySortTemplate

Strategy Учасники

- ***Strategy*** задає спільний для вієї множини алгоритмів інтерфейс; клас **Context** використовує цей інтерфейс для виклику конкретного алгоритму, визначеного в одному з похідних класів стратегії

- **ConcreteStrategyA, ConcreteStrategyB, …**– похідні класи ***Strategy***, які реалізують алгоритм з інтерфейсом, оголошеним в класі ***Strategy***

- **Context**
  - зберігає посилання на об'єкт класу ***Strategy***
  - ініціалізує посилання на об'єкт класу, похідного від ***Strategy***
  - може визначати інтерфейс для доступу стратегії до даних контексту

Strategy Відношення

- класи ***Strategy*** і **Context**
  - взаємодіють для реалізації вибраного алгоритму
  - **Context** може передавати стратегії потрібні дані в момент виклику методу, або передавати посилання на самого себе методам стратегії
- клас **Context** переадресовує запити своїх клієнтів об'єкту-стратегії
- попередньо клієнт створює об’єкт стратегії і передає контексту

Strategy Результати

- Утворення сімейства споріднених алгоритмів або поведінок, які можна використовуватиі в інших контекстах
- Альтернатива похідним класам контексту, коли поведінка прошивається в класі
- Уникання інструкцій умовного вибору.
- Стратегії можуть задавати різні варіанти реалізації алгоритмів.
- Клієнт повинен знати множину стратегій, з яких робить вибір.
- Різні варіанти обміну даними між контекстом і стратегією.

Strategy Реалізація

- Використання інтерфейсів (абстрактних класів):
  - для взаємного доступу до даних контексту і стратегії
  - передача контексту в аргументі методу стратегії
- Контекст – параметризований стратегією клас, конфігурується конкретним типом стратегії при інстанціюванні
- Деякий об’єкт стратегії може задаватися за замовчуванням і використовуватися клієнтом, якщо це його влаштовує

Патерн Iterator

- Назва та класифікація

  Iterator– патерн поведінки

- Призначення

  Надання послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи клієнтові особливостей його внутрішньої будови

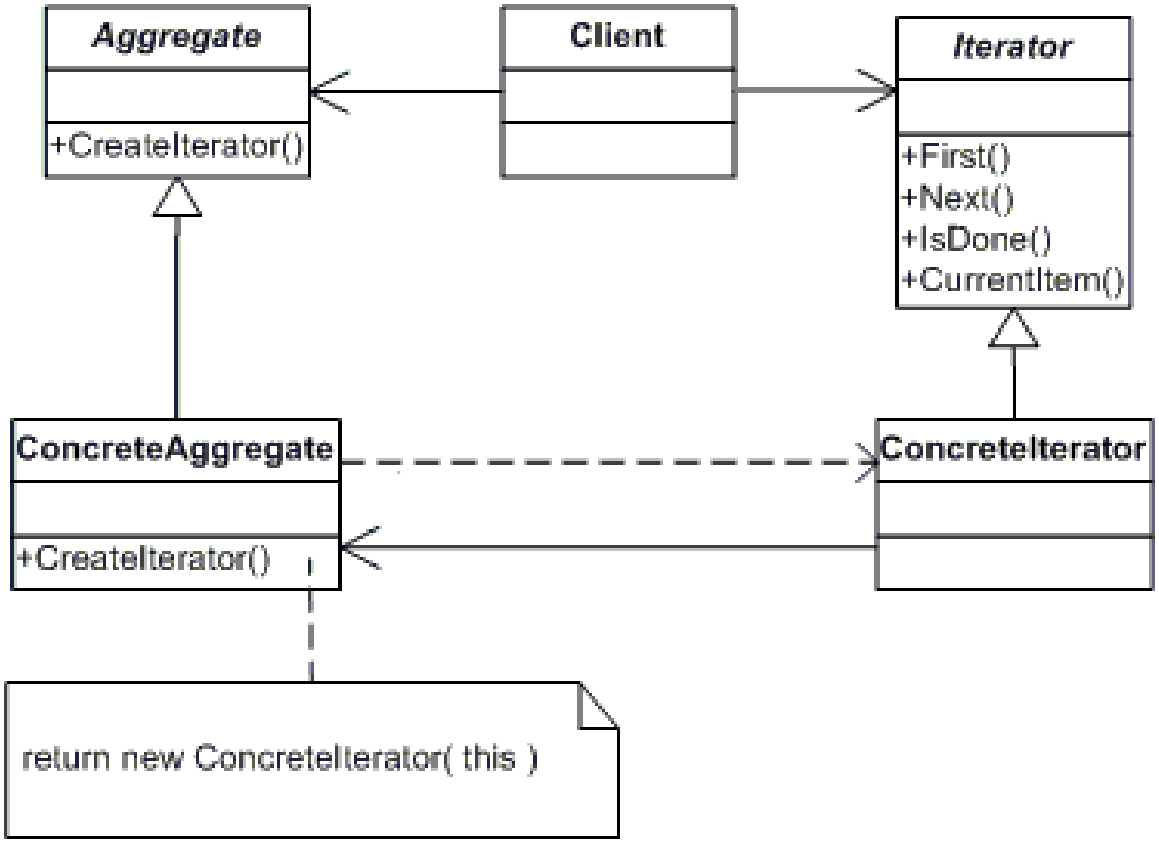
- Мотивація

  Складений об'єкт, напр., список, має надавати доступ (можливо у різні способи) до своїх елементів, не розкриваючи їхню внутрішню структуру. Більш того, така функціональність не повинна бути частиною інтерфейсу списку, тобто за доступ до елементів та обхід списку відповідає не сам список, а окремий об'єкт-ітератор.

Iterator Застосовність

- Для доступу до елементів складених об'єктів без розкриття їхньої внутрішньої будови.
- Для підтримки декількох активних обходів одного й того ж складеного об'єкта;
- Для подання уніфікованого інтерфейсу з метою обходу різноманітних складених структур (тобто для підтримки поліморфної ітерації).

Iterator Структура



Iterator Учасники

- **Iterator**
  - визначає інтерфейс для доступу та обходу елементів
- **ConcreteIterator**
  - реалізує інтерфейс класу **Iterator** ;
  - слідкує за поточною позицією під час обходу агрегату;
- **Aggregate**
  - визначає інтерфейс для створення об'єкта-ітератора;
- **ConcreteAggregate**
  - реалізує інтерфейс створення ітератора та повертає екземпляр відповідного класу **ConcreteIterator**

- **Відношення**

ConcreteIterator відслідковує поточний об'єкт у агрегаті та може вирахувати наступний.

.NET Framework

```
public interface IEnumerator<out T>
                              : IDisposable, IEnumerator{
         T Current { get; }
}
public interface IEnumerator {
         object Current { get; }
         bool MoveNext();
         void Reset();
}
public interface IEnumerable<out T>
                                        : IEnumerable {
         IEnumerator<T> GetEnumerator();
}
```

Патерн Memento

- Назва та класифікація

  Memento (Нагадування, напоминання, знимок) – патерн поведінки

- Призначення

  Для фіксування і тимчасового зберігання деякого поточного стану об’єкта поза його межами, з можливістю використання в майбутньому для відновлення стану. Не порушуючи інкапсуляцію.

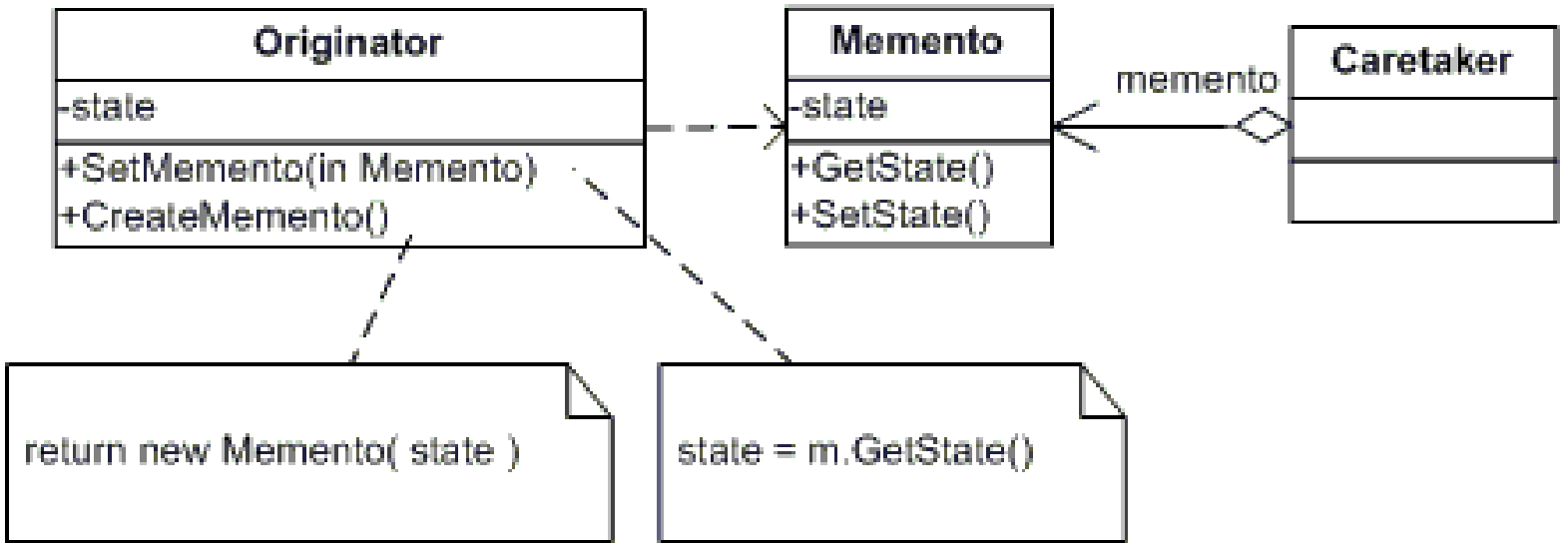
- Мотивація

  Об’єкт може мати потребу у зберіганні своїх поточних станів для оновлення в майбутньому шляхом повернення до зафіксованого раніше стану. Наприклад, відкат операцій

Memento Застосовність

- Для зберігання миттєвого знимку стану об'єкта (або його частини), щоб згодом об'єкт можна було відновити у тому ж самому стані
- Відношення між класами не повинно розкривати деталі реалізації та порушувати інкапсуляцію об'єкта

Memento Структура



MementoClassic
MementoCommand

Memento Учасники

- **Memento** — спогад, знимок, нагадування :
  - зберігає внутрішній стан об'єкта типу **Originator**; вид і обсяг інформації визначається потребами типу **Originator**;
  - на практиці надає два інтерфейси: з боку опікуна **CareTaker** — лише передача знимку іншим об'єктам; з боку **Originator** — широкий інтерфейс, що забезпечує доступ до всіх даних, необхідних для відтворення об'єкта (чи його частини) у попередньому стані. Ідеальний варіант — коли доступ до внутрішнього стану знимку надається лише його творцеві

- **Originator** — оригінал, творець, ініціатор:
  - створює знимок зі свого поточного внутрішнього стану;
  - використовує знимок для відтворення внутрішнього стану;

- **CareTaker** — опікун, хранитель, доглядач:
  - відповідає за зберігання знимку;
  - не проводить жодних операцій над знимком.

- **Відношення**

хранитель отримує знимок у творця, деякий час зберігає його у себе, опісля повертає творцеві; якщо творець не потребує відновлення свого попереднього стану, то повернення об’єкта не відбувається. Знимки пасивні.

Патерн Visitor

- Назва та класифікація

  Visitor– (відвідувач) патерн поведінки

- Призначення

  Визначає операцію, яка виконуватиметься над кожним з об’єктів деякої структури. При цьому класи об’єктів не зазнають змін

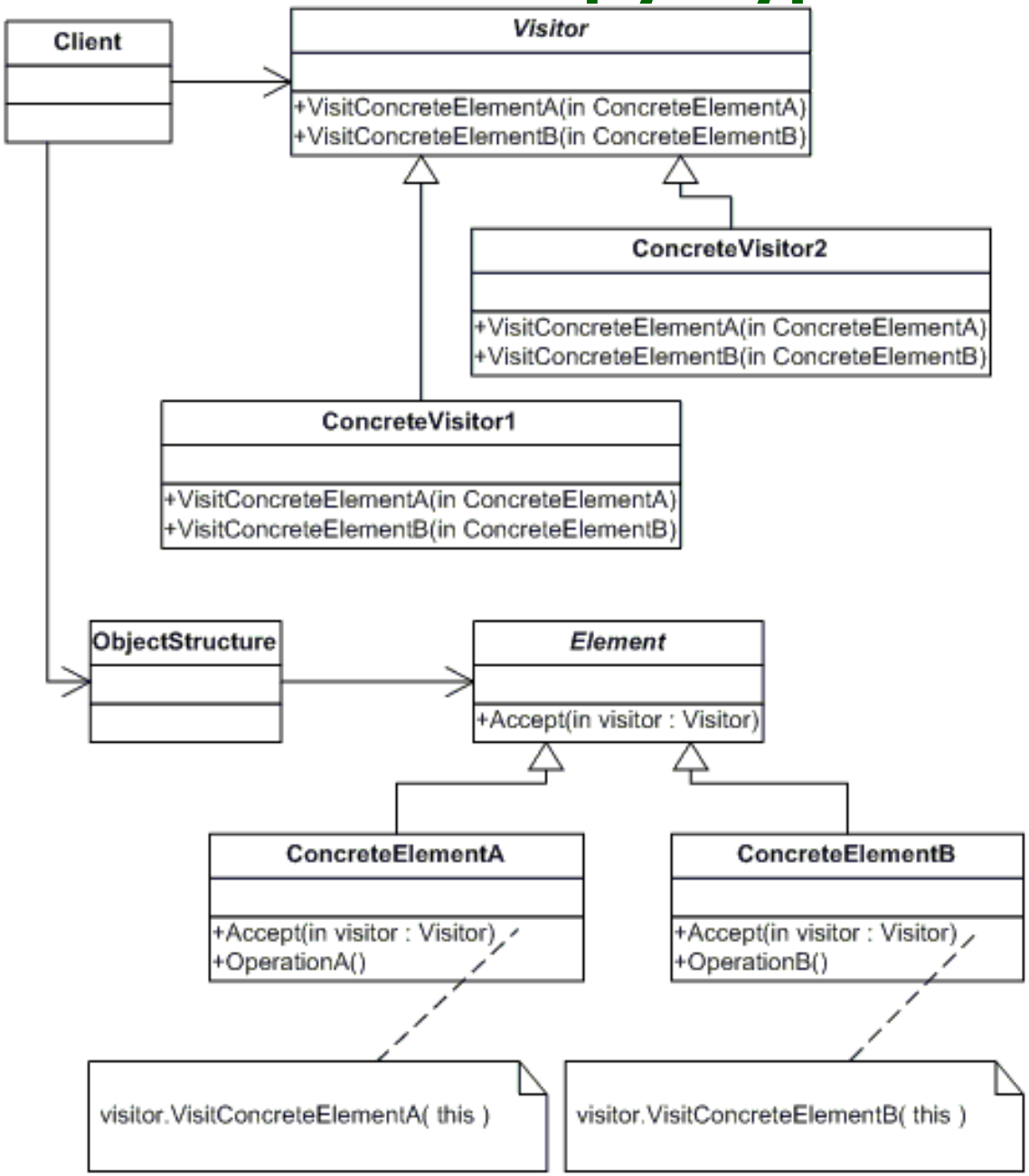
- Мотивація

  Для деякої ієрархії типів, над об’єктами якої треба виконувати деякі операції, будують другу ієрархію, класи якої реалізують ці операції над всіма об’єктами. При незмінній першій ієрархії нова операція вводиться за допомогою нового класу в другій ієрархії.

Visitor Застосовність

- До структури входять об’єкти різних типів, інтерфейси яких не доцільно поповнювати методами операцій
- Самі операції не пов’язані між собою; якщо вони мають якусь спорідненість, то їх реалізують в окремому класі
- Класи першої ієрархії усталені, не змінюються, а операції можуть додаватися нові

Visitor Структура



VisitorClassic_sch
Visitor_with_Composite