Design Patterns GoF

Structural Patterns

Структурні патерни

- Adapter
- Facade
- Decorator
- Bridge
- Composite
- Flyweight
- Proxy

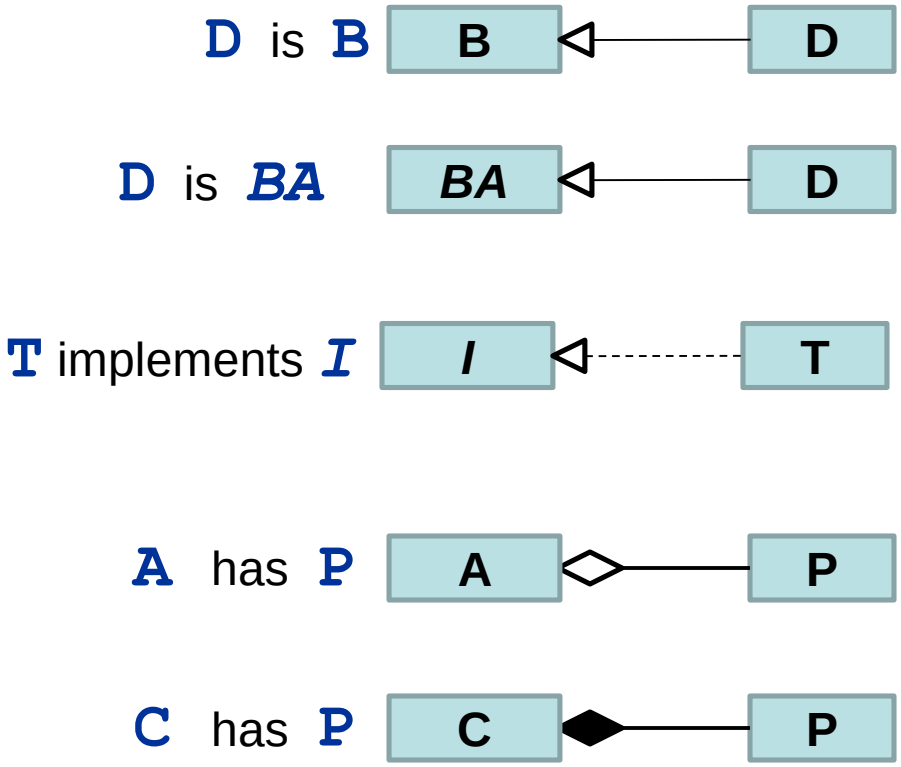
Означення

Патерн проектування:

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

??? Structural Patterns ???

Структурні наслідки об’єктно-орієнтованої парадигми



Стандартизація інформації патернами

Патерн проектування – це опис взаємодії об’єктів і класів, розроблених для вирішення загальної задачі проектування в конкретному контексті:

- Назва та класифікація
- Призначення
- Відомий також за назвою
- Мотивація
- Застосування
- Структура
- Учасники
- Відношення
- Результати
- Реалізація
- Приклад коду
- Відомі застосування
- Споріднені патерни

Патерн `Adapter`

- Назва та класифікація

`Adapter` (Адаптер) – патерн, який застосовують для структурування класів та об'єктів

- Призначення

Перетворює (*надає новий*) інтерфейс одного класу до іншого інтерфейсу, який влаштовує клієнта; забезпечує сумісну роботу класів з несумісними інтерфейсами, яка без нього була б неможливою

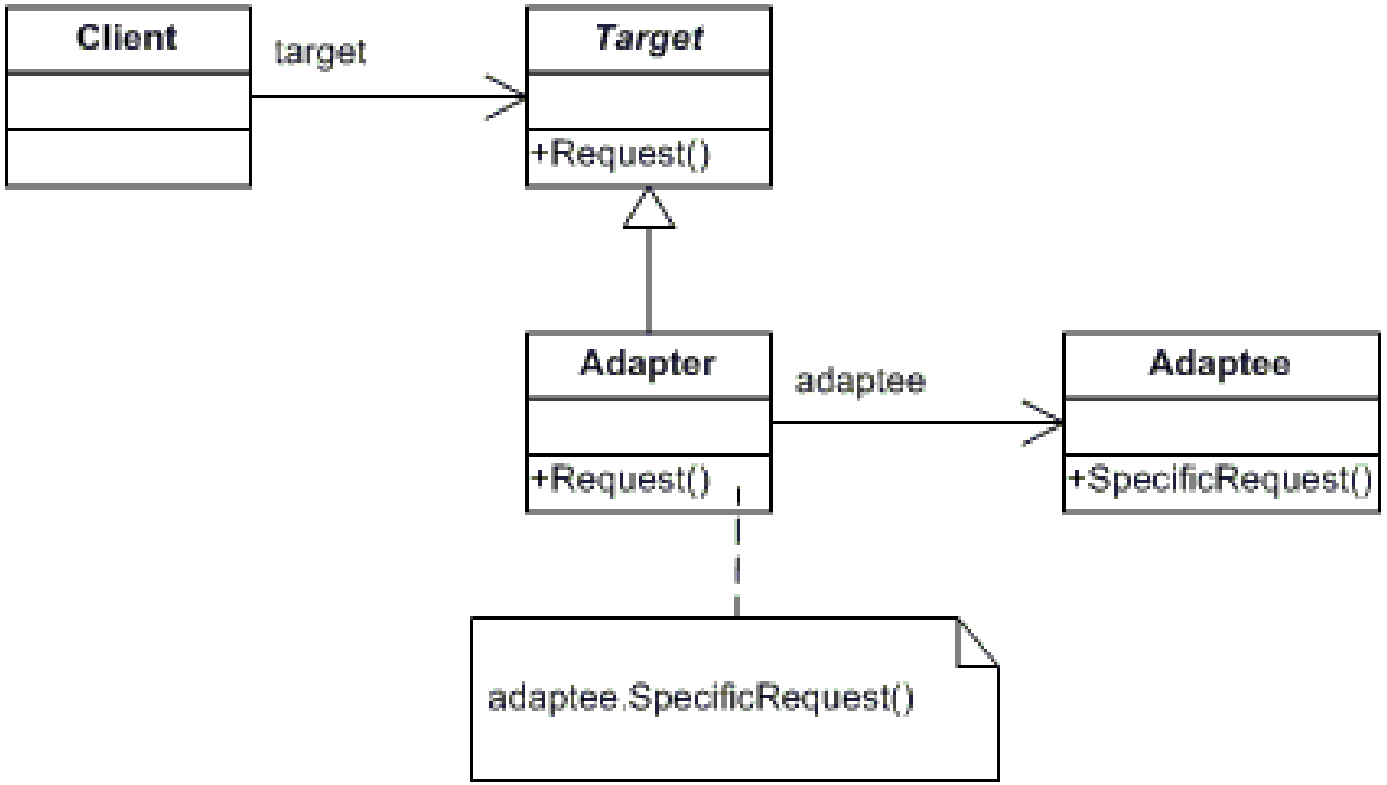
- Мотивація

У випадках, коли інтерфейс вже існуючих (напр., бібліотечних) класів не відповідає способу використання іншим кодом і не може бути перевизначеним, або певна сукуптість класів має відмінний інтерфейс, то для цих класів можна визначити клас-обгортку з потрібним інтерфейсом

Патерн Adapter

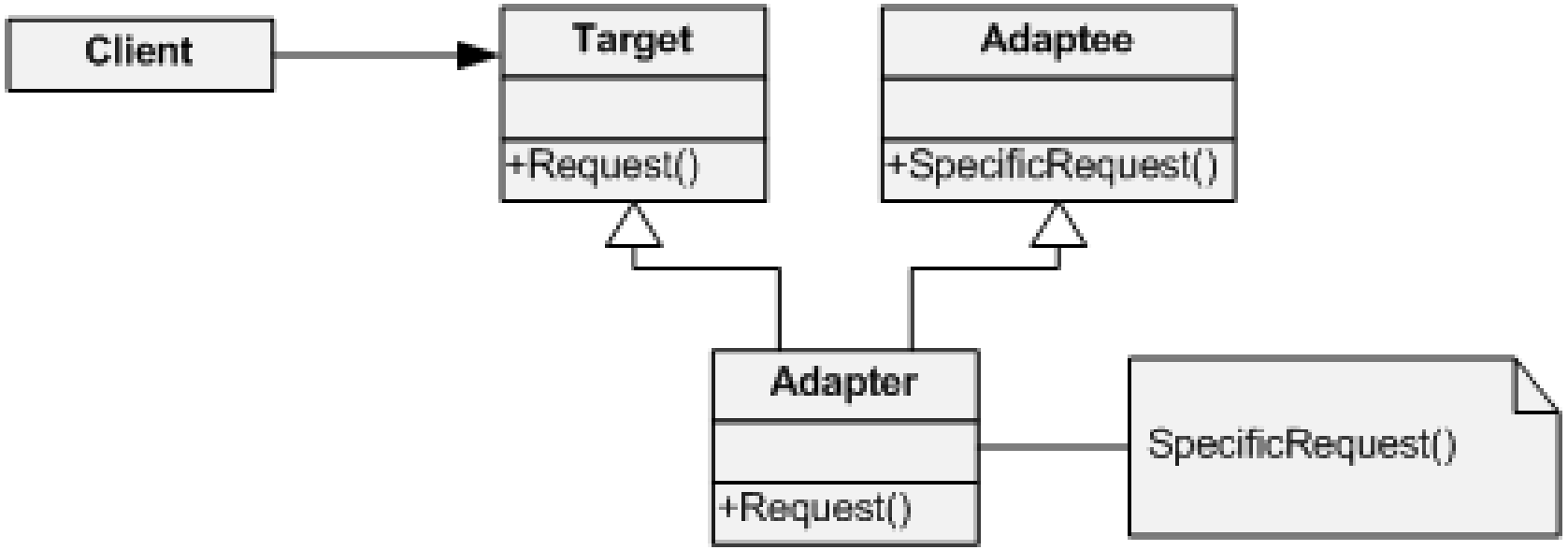
- Застосування
  - Інтерфейс існуючого класу не можна змінити, але його функціональність (повністю або частково) має використовуватися в інший спосіб
  - Треба утворити клас, який допускатиме повторне використання і взаємодію з наперед невідомими або незв’язаними з ним типами, які матимуть несумісні інтерфейси
  - *Випадок адаптера об’єктів*: треба використати кілька похідних типів однієї ієрархії, але недоцільно уніфіковувати їх інтерфейси шляхом подальшого утворення похідних типів

Патерн (Object) Adapter

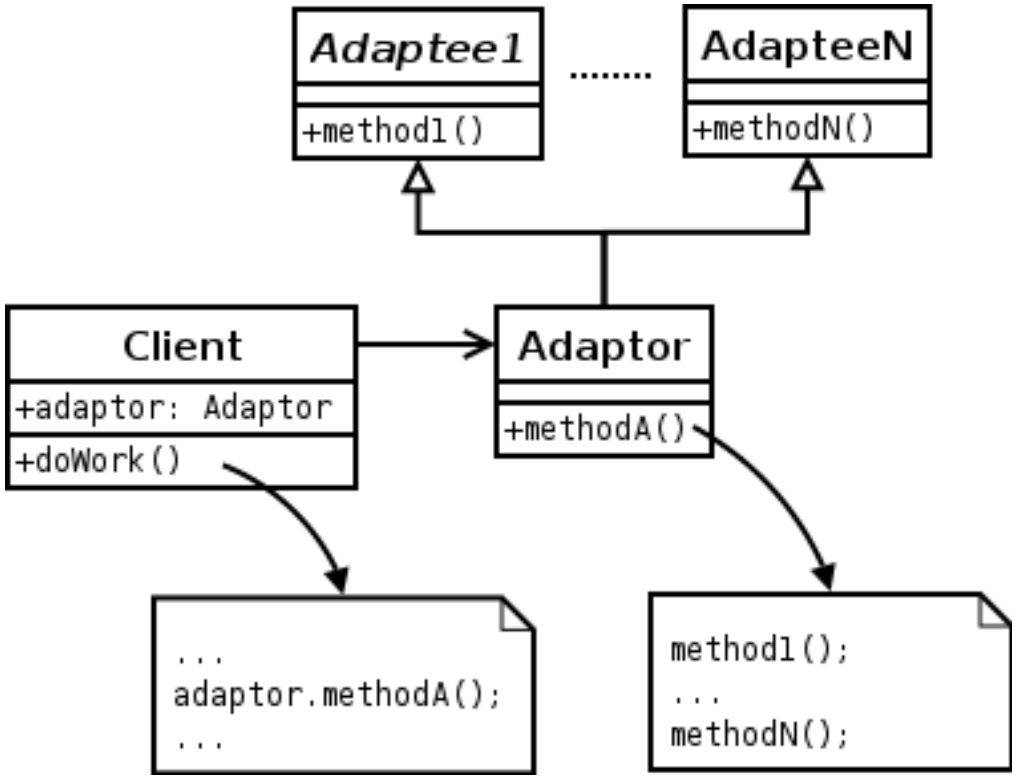


AdapterClassic

Патерн (Class) Adapter



Adapter


Adaptee1
+method1()
.....
AdapteeN
+methodN()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+methodA()
...
adaptor.methodA();
...
method1();
...
methodN();

Патерн Adapter

- **Учасники**:
  - **Target** визначає специфічний для області інтерфейс, який використовує Client
  - **Adapter** адаптує інтерфейс Adaptee до інтерфейсу Target
  - **Adaptee** визначає існуючий інтерфейс, який треба адаптувати
  - **Client** співпрацює з об’єктами через інтерфейс Target

- **Відношення** Клієнти викликають методи адаптера і в середині їх використовують функціональність адаптованого класу:
  - (адаптер об’єкта) через адаптований об’єкт
  - (адаптер класу) в методах нового класу

(Class) Adapter Результати

- Адаптований **Adaptee** до **Target**, який делегує дії конкретному класові **Adaptee**. Адаптація не поширюватиметься одночасно на похідні класи **Adapter**.
- Клас адаптера **Adapter** може перевизначити деякі методи адаптованого класу **Adaptee**
- Об’єкт адаптованого класу стає безпосередньо частиною адаптера, а його інтерфейс є безпосередньо доступним в методах адаптера (без вказівника)

(Object) Adapter Результати

- Доступ до адаптованого об’єкта в методах адаптера реалізується **через вказівник** (посилання), тому можна адаптувати і об’єкти похідних типів **Adaptee**, у тому числі з використанням поліморфізму
- Методи адаптованого класу (безпосередньо) не перевизначаються

Adapter Реалізація

- Наслідування C++

```
class Adapter: public Target, private Adaptee
```

- Змінні адаптери для вузького інтерфейсу з використанням:
  - абстрактних операцій
  - різних типів об’єктів-уповноважених
  - параметризовані адаптери

AdapterExternalPolymorphism

Патерн Facade

- Назва та класифікація

Facade (Фасад) – патерн для структурування об'єктів

- Призначення

Надає (*новий*) уніфікований інтерфейс деякої системи замість кількох (*багатьох*) інтерфейсів її типів (компонентів). Це інтерфейс вищого рівня, який спрощує використання системи.

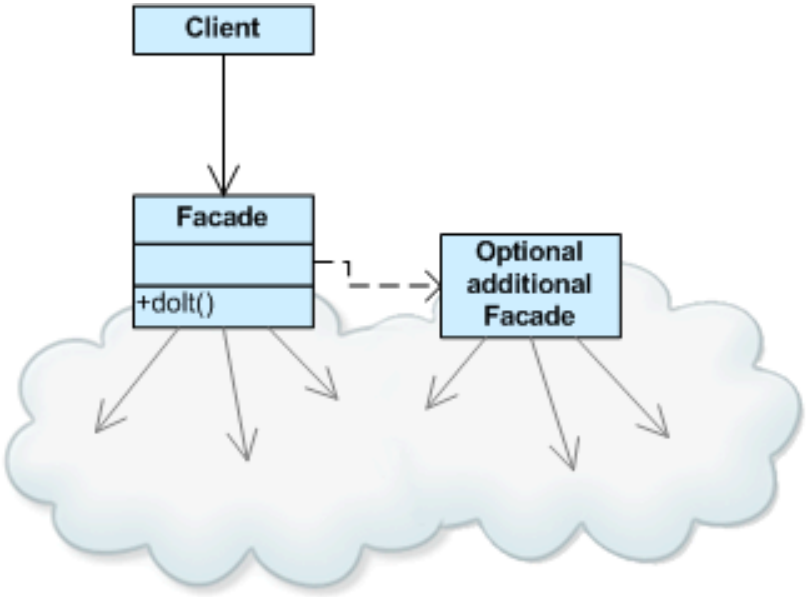
- Мотивація

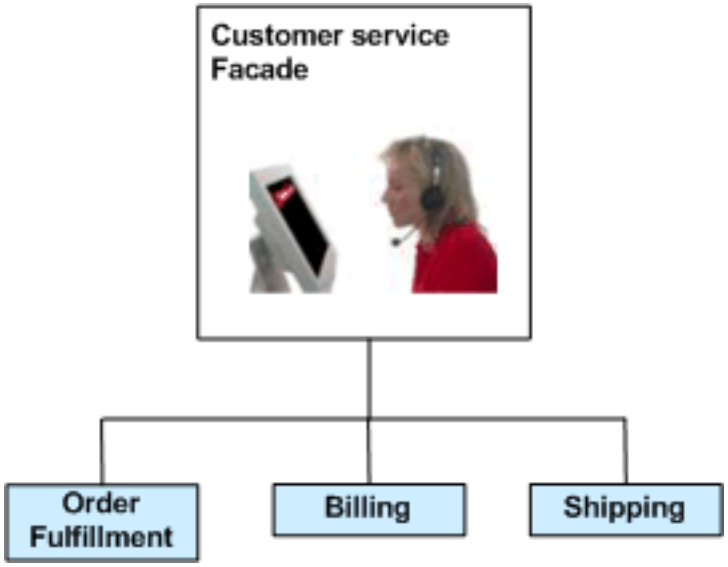
Якщо значна частина функціональності деякої системи не використовується безпосередньо клієнтом, або використовується згідно певних сценаріїв, то відповідні низькорівневі інтерфейси можна приховати за додатковим інтерфейсом вищого рівня

Патерн Facade

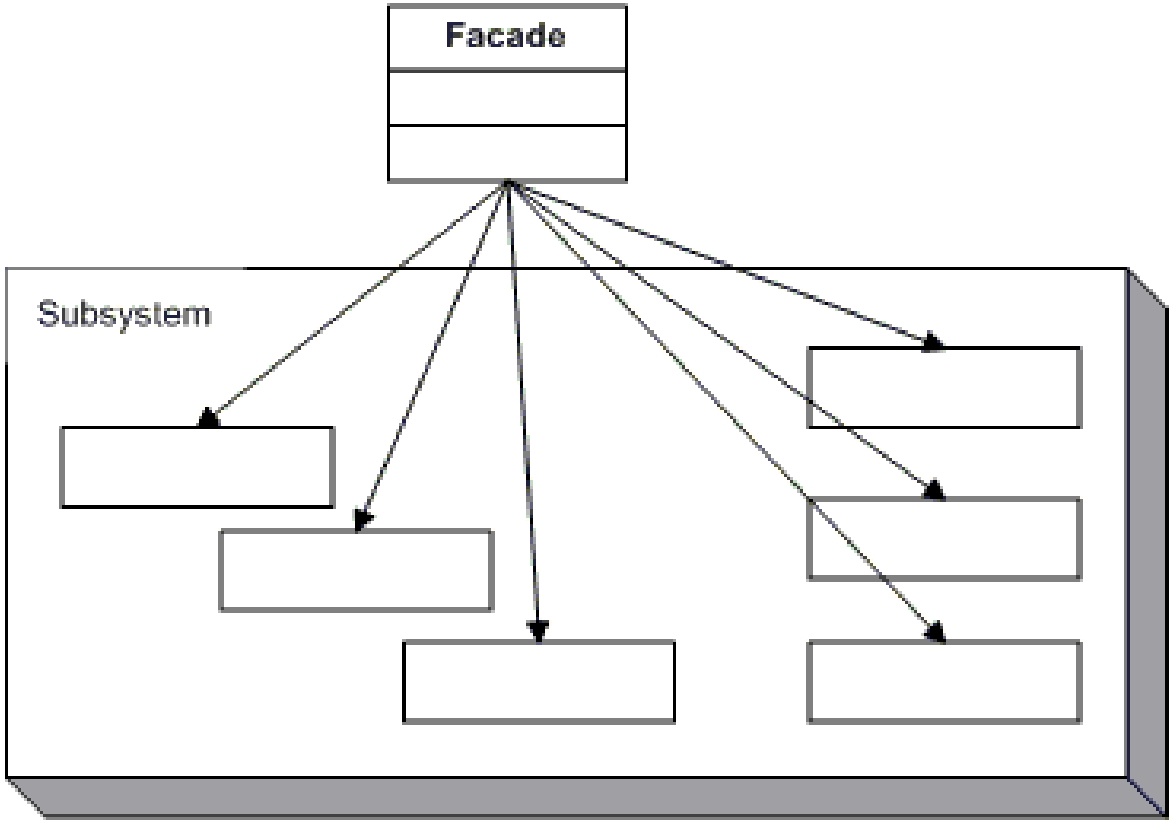
- Застосування
  - для надання за замовчуванням (для більшості клієнтів) простого інтерфейсу до складної системи; в окремих випадках клієнтам залишають доступ до інтерфейсів нижчого рівня
  - для відділення частини низькорівневого інтерфейсу від її реалізації
  - для розкладу системи на кілька шарів з окремими точками входу

Facade Структура





Facade Структура



Facade

- **Учасники**

- **Facade**
  - знає, які класи підсистем відповідають за запит
  - делегує запит від кієнта відповідним об’єктам підсистеми

- **Subsystem classes**
  - реалізують функціональність підсистем
  - опрацьовують завдання, призначені об’єктом **Facade**
  - не мають посилань на **Facade** і жодної інформації про його існування

- **Відношення**
  - Клієнти звертаються до підсистеми, надсилаючи запити **Facade**, який переадресовує їх відповідним об’єктам підсистеми, вони й виконують основну роботу
  - Клієнти, які використовують **Facade**, не мають доступу до інтерфейсів нижчого рівня

Facade Результати

- Ізолювання клієнтів підсистеми від її компонентів
- Спрощення роботи з підсистемою
- Послаблення зв’язанності клієнта і підсистеми, її реалізацію можна змінювати незалежно, у тому числі для перенесення на інші платформи
- Фасад не забороняє роботу клієнтів з інтерфейсами нижчого рівня (якщо це потрібно)

Facade Реалізація

- Використання відкритих і закритих класів підсистеми, а також просторів назв
- Зменшити зв’язаність можна за рахунок абстрактного класу **Facade**, тоді його конкретні похідні класи відповідатимуть конкретним реалізаціям

Патерн Decorator

- Назва та класифікація

  Decorator (Декоратор) – патерн, який застосовують для структурування ***об'єктів***

- Призначення

  Динамічно надає **об’єктові** додаткову функціональність. Альтернатива (статичному) розширенню функціональності за рахунок похідних класів

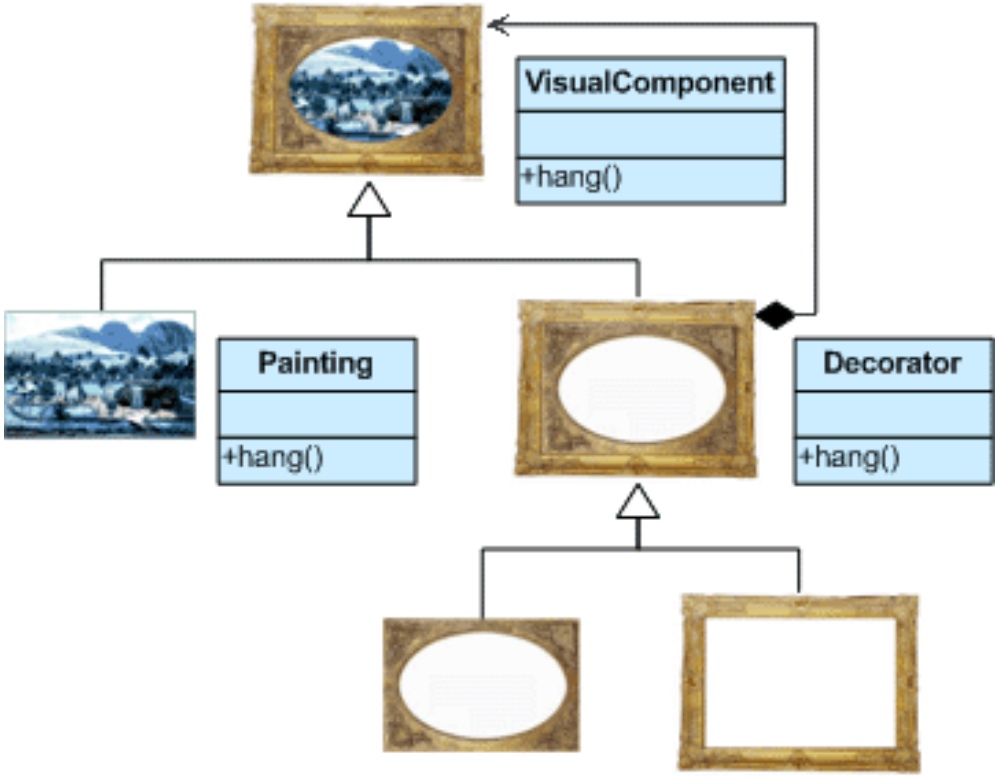
- Відомий також як **Wrapper**(Обгортка)

- Мотивація

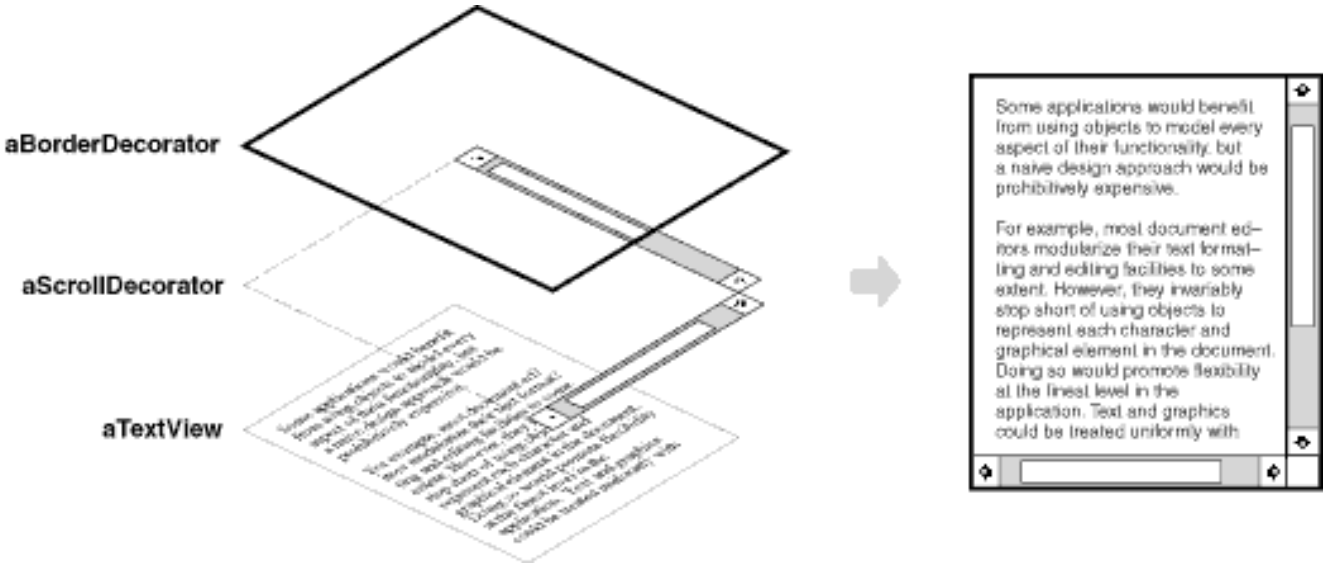
  Надання додаткової функціональності **не цілому класові, а окремим об’єктам.**

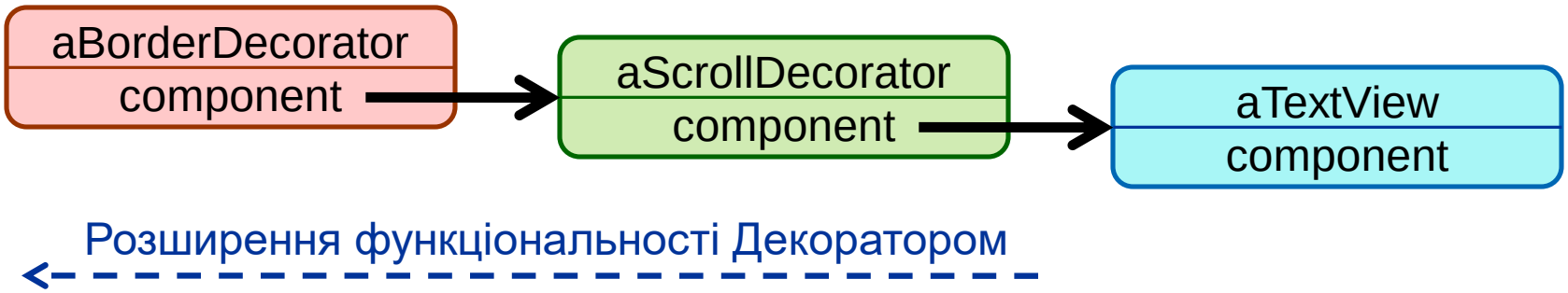
  Приклад: в межах графічної бібліотеки можливість в окремих випадках додати скролінг (чи якусь іншу функціональність) довільному графічному елементу. Варіант похідних типів ускладнює класи і супровід бібліотеки.

Decorator ++Мотивація

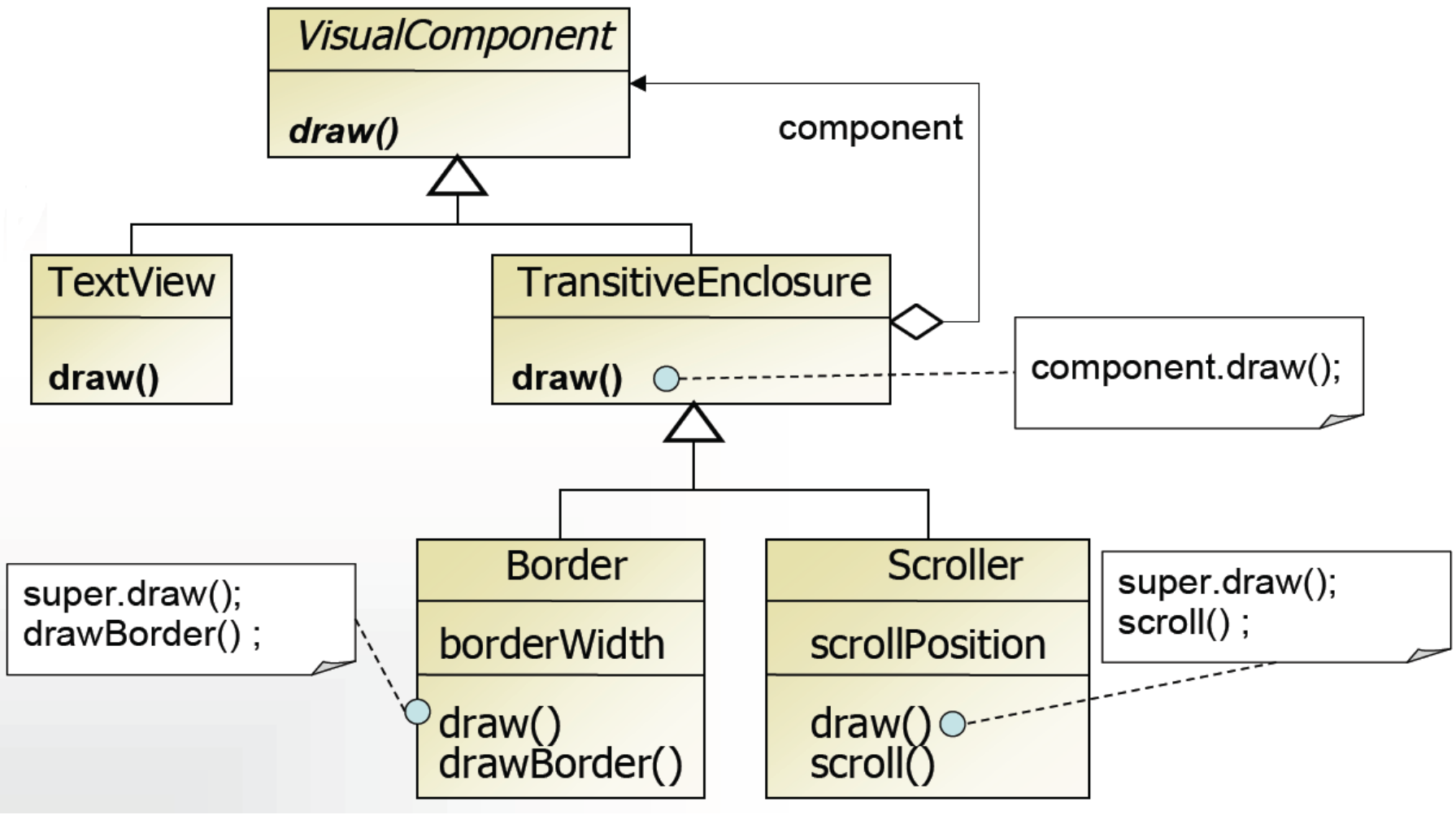


Decorator ++Мотивація

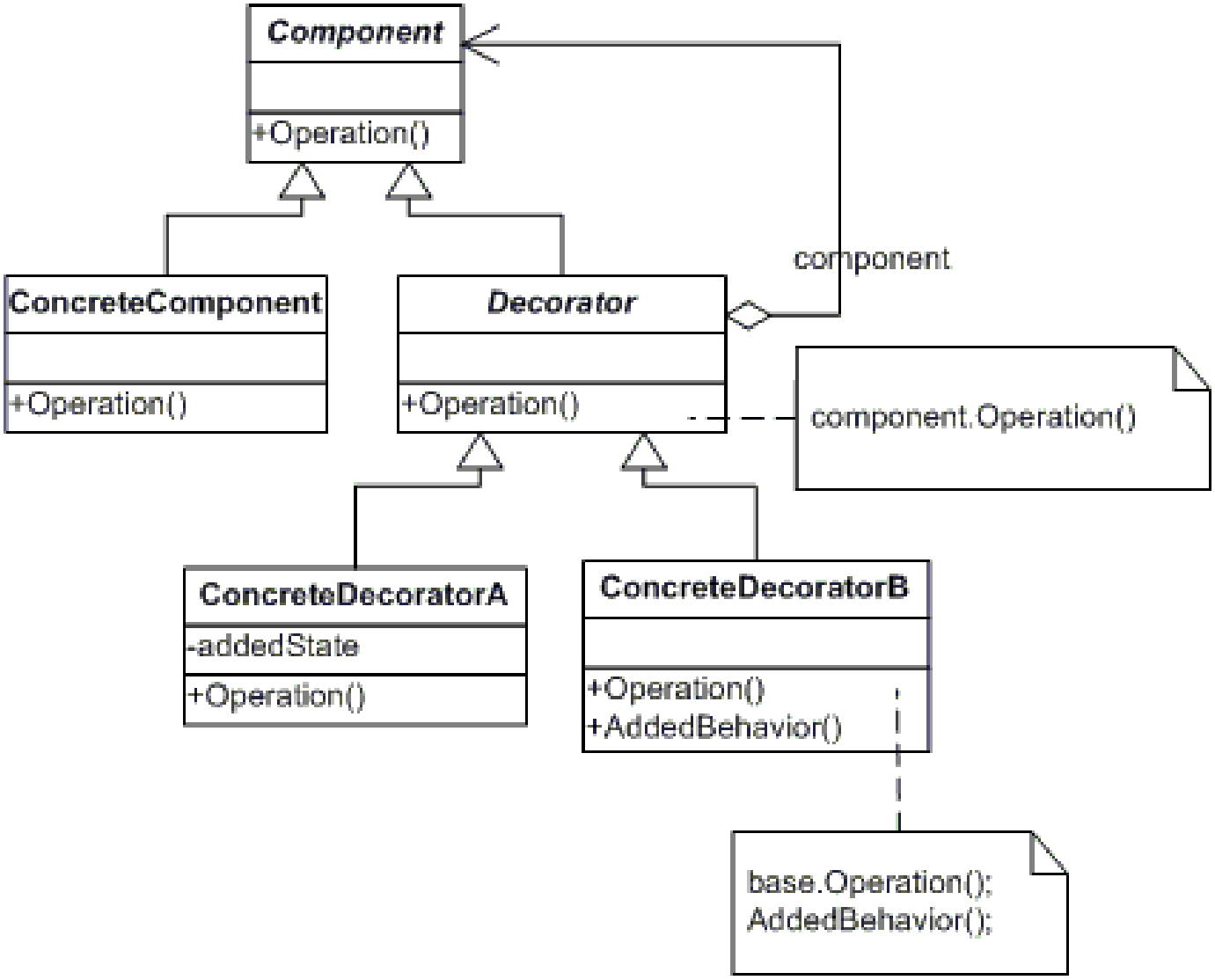




Decorator Структура



Decorator ++Структура



DecoratorClassic

Decorator

❑ Учасники:

- **Component** (`VisualComponent`)

  визначає інтерфейс для об'єктів, які можуть мати обов'язки, що додаються до них динамічно

- **ConcreteComponent** (`TextView`)

  визначає об’єкт, до якого можуть бути додані додаткові обов'язки (відповідальності)

- **Decorator** (`Decorator`)

  підтримує посилання на об’єкт Component і визначає інтерфейс, який відповідає інтерфейсу Component

- **ConcreteDecorator** (`BorderDecorator`, `ScrollDecorator`)

  додає нові відповідальності до компонента

❑ Відношення

Decorator переадресовує запити об’єктові **Component**, який може виконувати також додаткові операції до і після переадресування

DecoratorClassicWidget

Патерн Bridge

- Назва та класифікація

  Bridge (Міст) – патерн, який застосовують для структурування ***об'єктів***

- Призначення

  Відокремити абстракцію від її реалізації таким чином, щоб перше та друге можна було змінювати незалежно одне від одного.

- Відомий також як **Handle/Body**

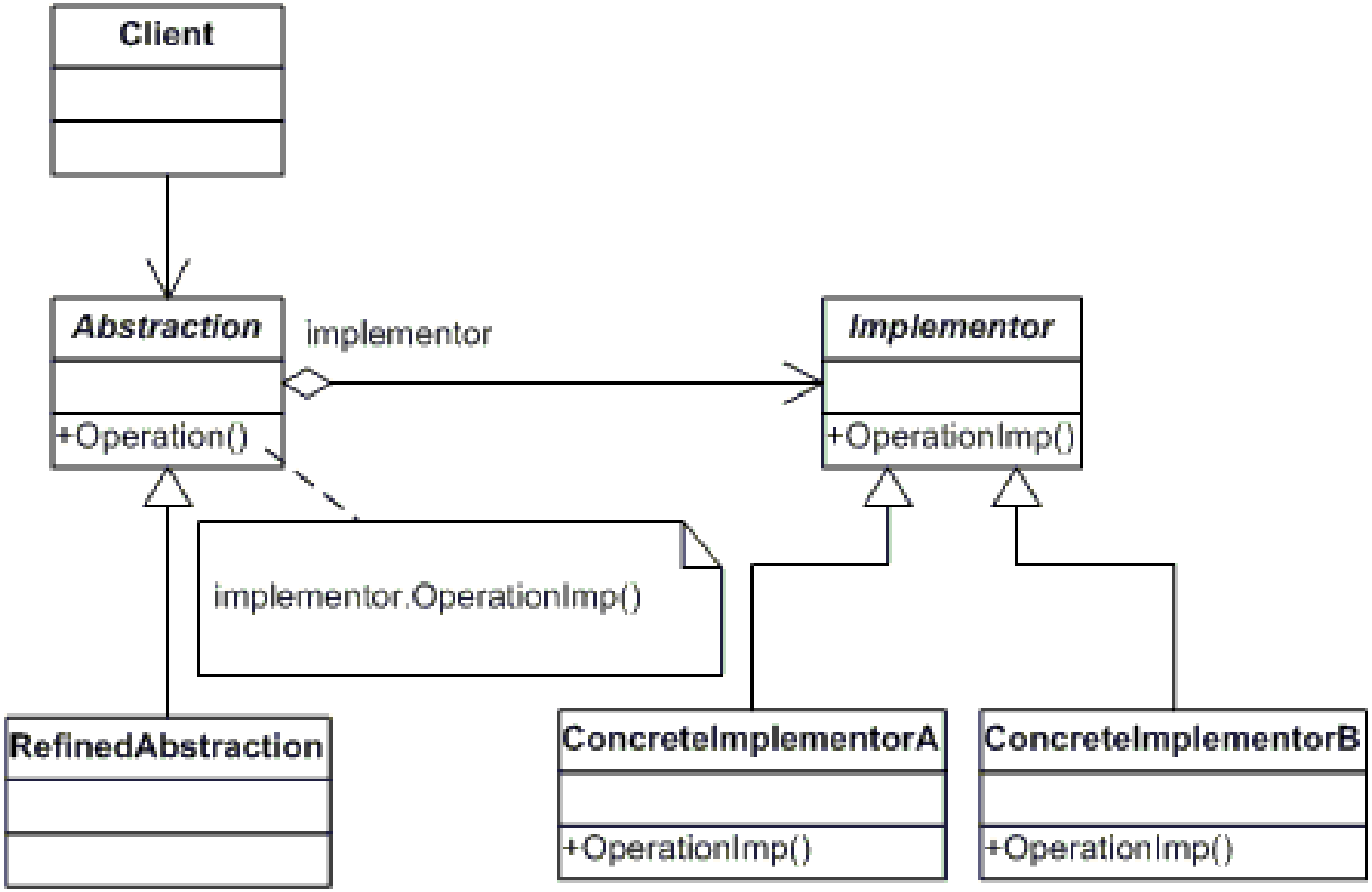
- Мотивація

  Якщо для деякої абстракції можливо кілька реалізацій, зазвичай застосовують наслідування. Абстрактний клас визначає інтерфейс абстракції, а його конкретні підкласи по-різному реалізують його. Але наслідування жорстко прив'язує реалізацію до абстракції, що перешкоджає незалежній модифікації, розширенню та повторному використанню абстракції та її реалізації.

Патерн Bridge

- Застосування
  - треба запобігти постійній прив'язці абстракції до реалізації; напр., коли реалізацію необхідно обрати під час виконання програми
  - як абстракції, так і реалізації забезпечують розширення новими підкласами; у цьому разі **Bridge** дає змогу комбінувати різні абстракції та реалізації та змінювати їх незалежно одне від одного
  - зміни у реалізації не повинні впливати на клієнтів, тобто **клієнтський** код не повинен перекомпільовуватись
  - треба повністю сховати від клієнтів реалізацію абстракції
  - треба розподілити одну реалізацію поміж кількох об'єктів (можливо застосовуючи підрахунок посилань), і при цьому приховати це від клієнта

Bridge Структура



Bridge

❑**Учасники**:

Abstraction – абстракція:

визначає інтерфейс абстракції;

зберігає посилання на об'єкт типу **Implementor**;

RefinedAbstraction – уточнена абстракція:

розширює інтерфейс, означений абстракцією **Abstraction**;

Implementor – реалізатор:

визначає інтерфейс для класів реалізації. Він не зобов'язаний точно відповідати інтерфейсу класу **Abstraction**. Насправді обидва інтерфейси можуть бути зовсім різними. Зазвичай, інтерфейс класу **Implementor** надає тільки примітивні операції, а клас **Abstraction** визначає операції більш високого рівня, що базуються на цих примітивах;

ConcreteImplementor – конкретний реалізатор:

містить конкретну реалізацію інтерфейсу класу **Implementor**.

❑**Відношення**

Abstraction містить в собі об’єкт **Implementor** і переадресовує йому запити

Патерн Composite

- Назва та класифікація

  Composite (Композит)

  патерн для структурування ***об'єктів***

- Призначення

  Визначає дерево для подання ієрархічної залежності “частина-ціле”, при цьому однаково трактуються усі вузли: як кінцеві (**примітиви**,“листки”), так і ті, що об’єднують групу об’єктів (**контейнери**)

Composite

- Мотивація

  Приклад: в редакторах усі графічні об’єкти – як прості, так і контейнери – є похідними від одного абстрактного класу, інтерфейс якого становлять як специфічні для кожного об’єкта методи (**Draw**), так і спільні для всіх(напр., доступ і менеджмент нащадками).

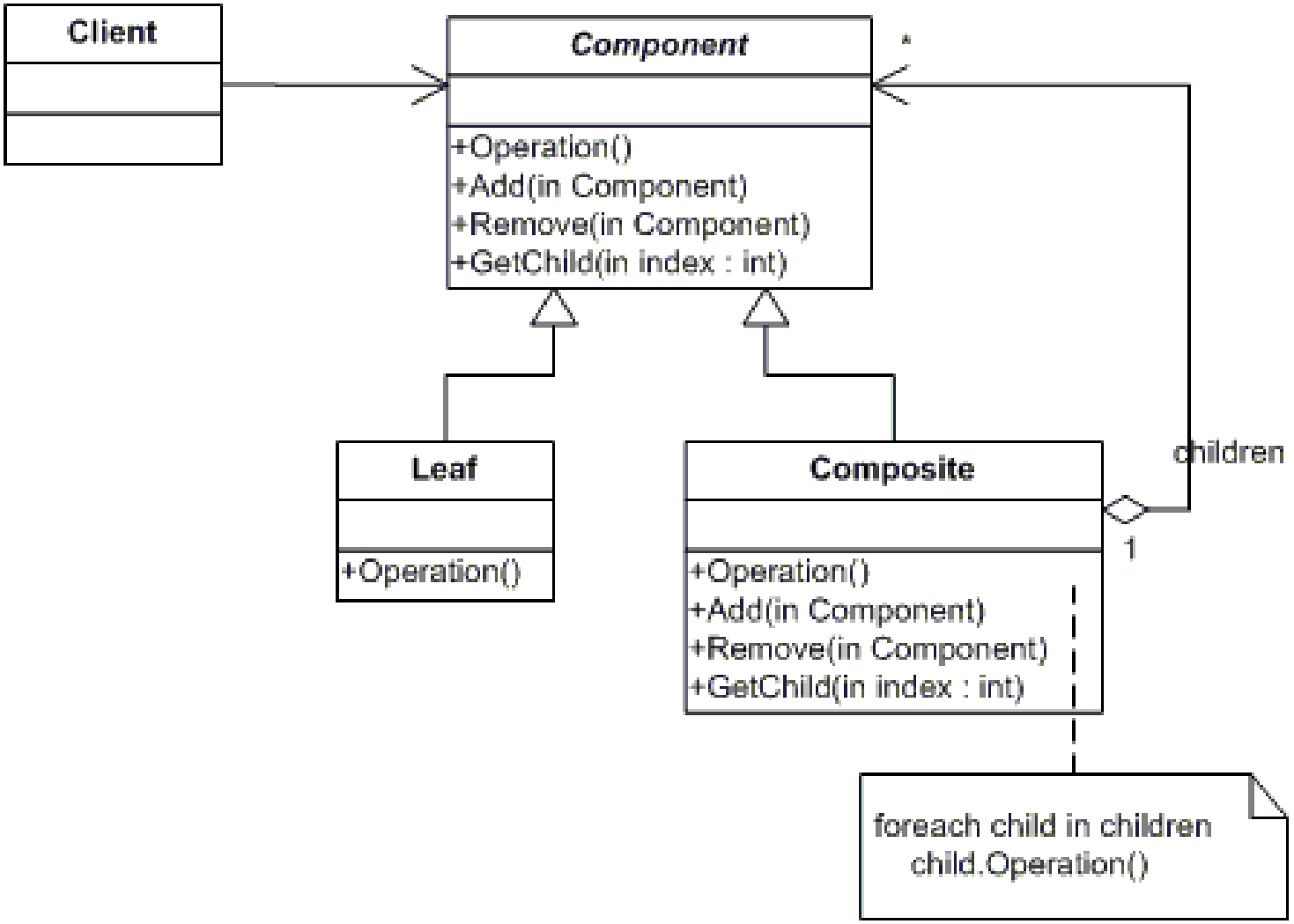
  Інші прикл.: меню, елементами якого є інші меню; каталоги файлової системи, які крім безпосередньо файлів містять підкаталоги; контейнери, елементами яких є контейнери ...

- **Застосування**

  - для подання у вигляді дерева ієрархії об’єктів із відношенням “частина”-”ціле”
  - клієнт однаково трактує усі вузли – кінцеві (примітиви, “листки”) і композити (контейнери), що об’єднують групу об’єктів

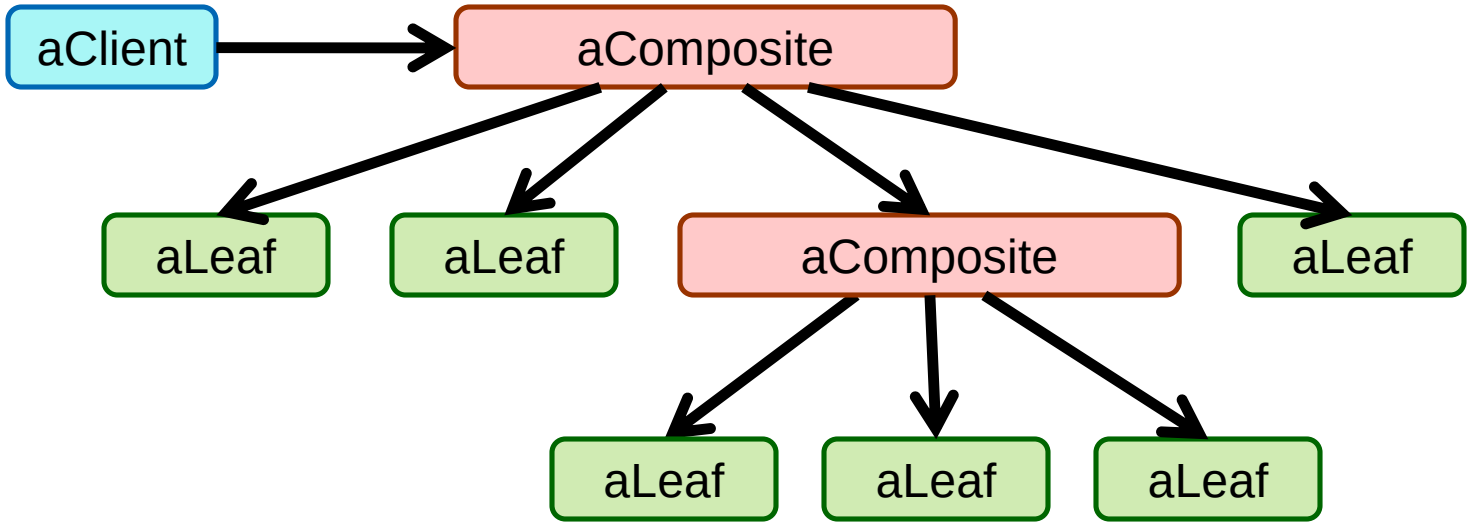
Composite

Структура: діаграма класів



Composite

Структура: діаграма об’єктів



Composite

❑**Учасники**:

- **Component** (**DrawingElement**)
  - оголошує інтерфейс для об'єктів у композиції
  - реалізує поведінку за замовчуванням для інтерфейсу, спільного для всіх класів
  - оголошує інтерфейс для доступу та керування своїми вкладеними компонентами
- **Leaf** (**PrimitiveElement**)
  - є листковим об'єктом у композиції, який вже не містить компонентів
  - визначає поведінку примітивних об'єктів у композиції
- **Composite** (**CompositeElement**)
  - визначає поведінку компонентів, які включають інші компоненти
  - реалізує операції інтерфейсу **Component** над вкладеними компонентами
- **Client** (**CompositeApp**)

  маніпулює об'єктами у композиті через інтерфейс **Component**

Composite

Відношення

Client використовує інтерфейс **Component** для взаємодії з об’єктами.

Якщо отримувачем запиту виявиться об’єкт **Leaf**, то він і опрацює запит.

Якщо ж ним виявиться об’єкт **Composite**, то він перенаправляє запит своїм об’єктам-нащадкам, за потребою виконуючи якусь дію до або після перенаправлення.

CompositeClassic_4_csh
CompositeClassic_3_Safety

Патерн Flyweight

- Назва та класифікація

  Flyweight (Легковаговик)

  патерн для структурування ***об'єктів***

- Призначення

  З метою ефективної підтримки множини дрібних об'єктів використовує поділ набору атрибутів об’єкта на дві частини (стани):

  - внутрішній (*intrinsic state*) – зберігається у **Flyweight**-об’єкті ним самим
  - зовнішній (*extrinsic state*) – зберігається поза **Flyweight**-об’єктом або обчислюється за потребою

  Внутрішній стан спільно використовується об’єктами певної множини.
  Такі об’єкти може надавати фабрика, їх значно менше усіх елементів множини

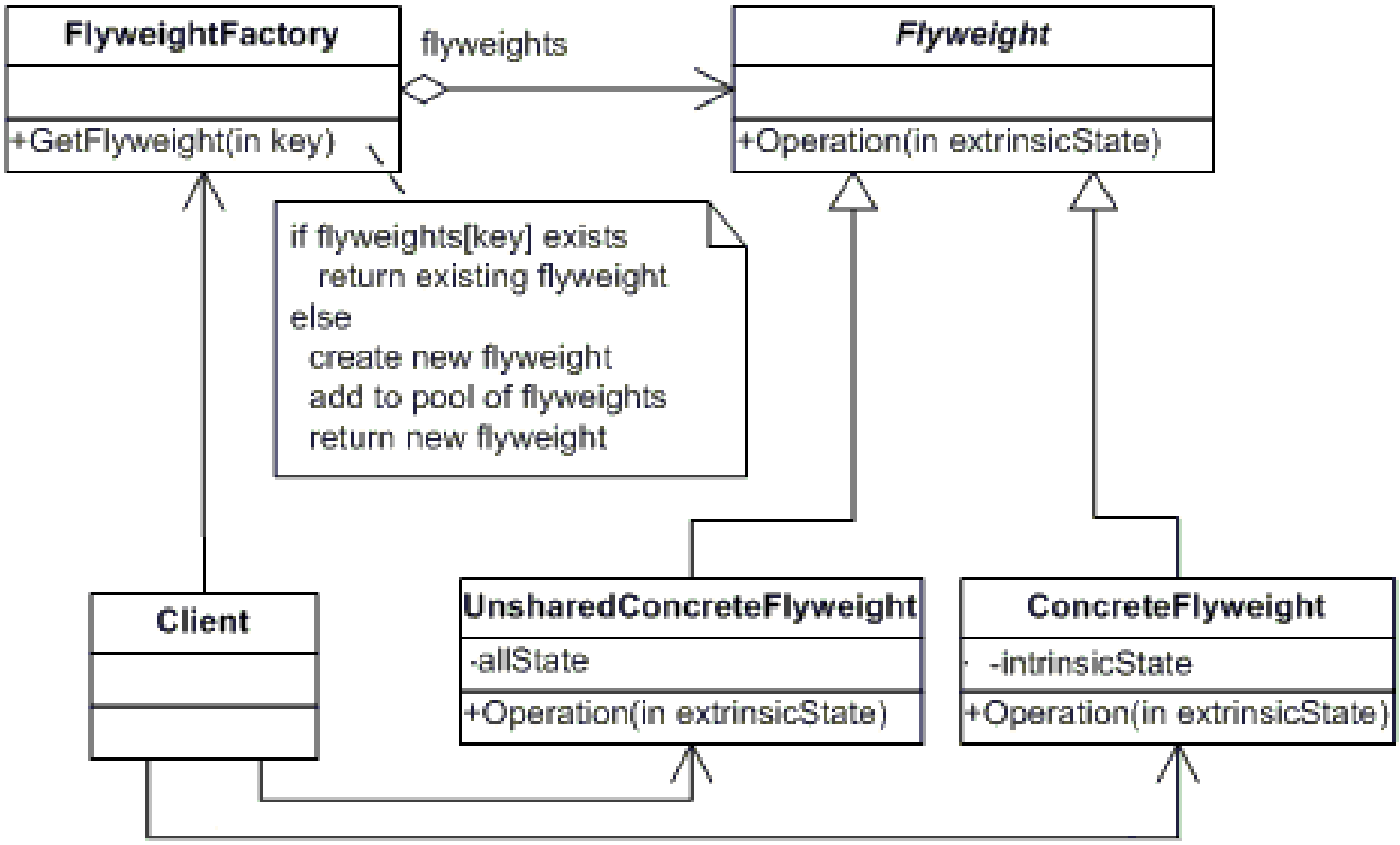
- Мотивація

  Приклад: в текстових редакторах **Flyweight**-об’єктом може бути glyph, який володіє даними (внутрішній стан) і методами для графічного зображення символа. Додаткові дані надає контекст (координати, колір тощо – зовнішній стан). Таких об’єктів значно менше, ніж всього символів у документі.

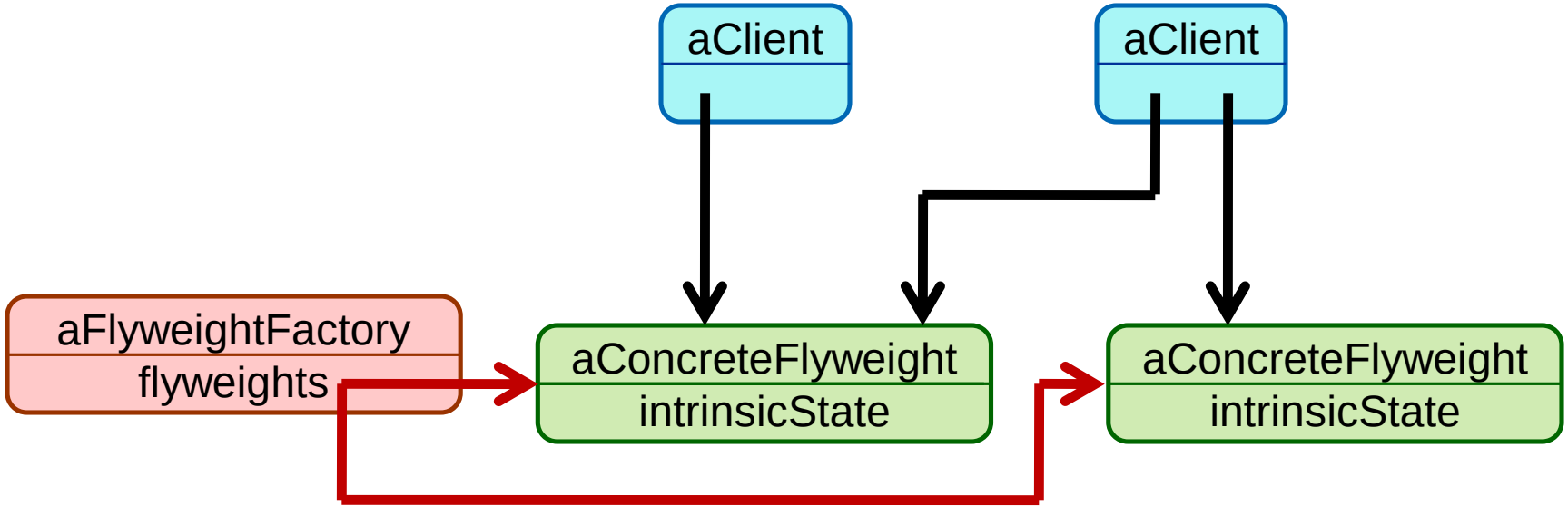
Flyweight

- Застосування
  - у програмі задіяна велика кількість об’єктів
  - велика кількість об’єктів зумовлює зростання накладних затрат на їхнє зберігання
  - більшу частину параметрів стану об’єктів можна винести назовні
  - значна кількість груп об'єктів може бути замінена відносно малою кількістю об'єктів, які будуть використовуватися для визначення даних, що стосуються внутрішнього стану
  - програма не залежить від ідентичності об'єктів, тобто в силу спільного використання **Flyweight**-об’єктів тести на ідентичність будуть повертати значення **true** для концептуально різних об'єктів

Flyweight Структура



Flyweight ++Структура



Flyweight

❑**Учасники**:

- **Flyweight** – визначає інтерфейс для отримання легковаговиком даних зовнішнього стану та виконанням дій над ними;
- **ConcreteFlyweight** – реалізує інтерфейс **Flyweight** і додає дані внутрішнього стану; його об’єкти матимуть спільне використання в різних контекстах, але є незалежними від контексту
- **UnsharedConcreteFlyweight** інтерфейс **Flyweight** допускає спільне використання об’єктів у різних контекстах, але не в усіх реалізацій це має бути, напр., класи **Row** і **Column**
- **FlyweightFactory** – утворює об’єкти-легковики і управляє ними;
- **Client** – клієнт: зберігає посилання на одного чи кількох об’єктів-легковиків, відповідає за формування і зберігання даних зовнішнього стану

❑**Відношення**

Flyweight містить в собі дані внутрішнього стану, **Client** – дані зовнішнього і передає їх **Flyweight** . **Client** напряму не може утворювати об’єкти **Flyweight** , а отримує їх від фабрики **FlyweightFactory**

Flyweight Результати

- Використання **Flyweight** не виключає затрат на передачу, пошук чи обчислення внутрішнього стану, однак це компенсується економією пам'яті за рахунок спільного використання легковаговиків (зменшується кількість об'єктів, пам'ять для зберігання їхнього внутрішнього стану, а дані зовнішнішнього часто лише обчислюються)
- Патерн **Flyweight** часто використовується разом з **Composite** для задання графа з листовими вузлами, які спільно використовуються (при цьому використовують вказівник на вузол верхнього рівня, який є елементом зовнішнього стану, що визначає специфіку алгоритмів)
- якщо затрати на повторне створення об'єктів внутрішнього стану невеликі, то замість **Flyweight** можна використовувати **Prototype**

Flyweight Реалізація

- Ефективність використання патерну залежить від *можливості винесення зовнішнього стану* і його формування.
- Оскільки об’єкти спільно використовуються, то клієнт не може їх безпосередньо утворювати. Їх повертає фабрика, яка для зберігання може використовувати асоційований масив, де ключем є об’єкт внутрішнього стану або його характерна ознака. Можна також вести облік кількості посилань з можливістю знищення об’єкта.

FlyweightGazillion

Патерн Proxy

- Назва та класифікація

  Proxy(Замісник, Вповноважений)

  патерн для структурування ***об'єктів***

- Призначення

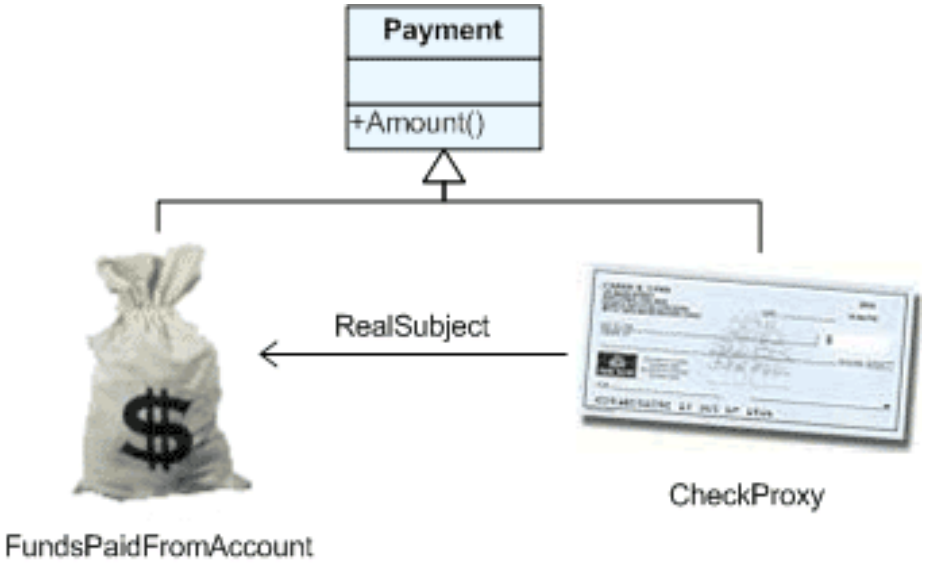
  Є сурогатом іншого об’єкта і контролює до нього доступ

- Відомий також як **Surrogate**

- Мотивація

  Приклад: в текстових редакторах складні графічні об’єкти утворювати лише тоді, коли в цьому виникне потреба, а до того часу замість нього використовувати інший об’єкт, який поводиться аналогічно графічному і може його інстанціювати(для цього можна задати команду **Draw**). Після інстанціювання замісник переадресовуватиме всі команди реальному графічному об’єктові через внутрішнє посилання на нього.

Proxy

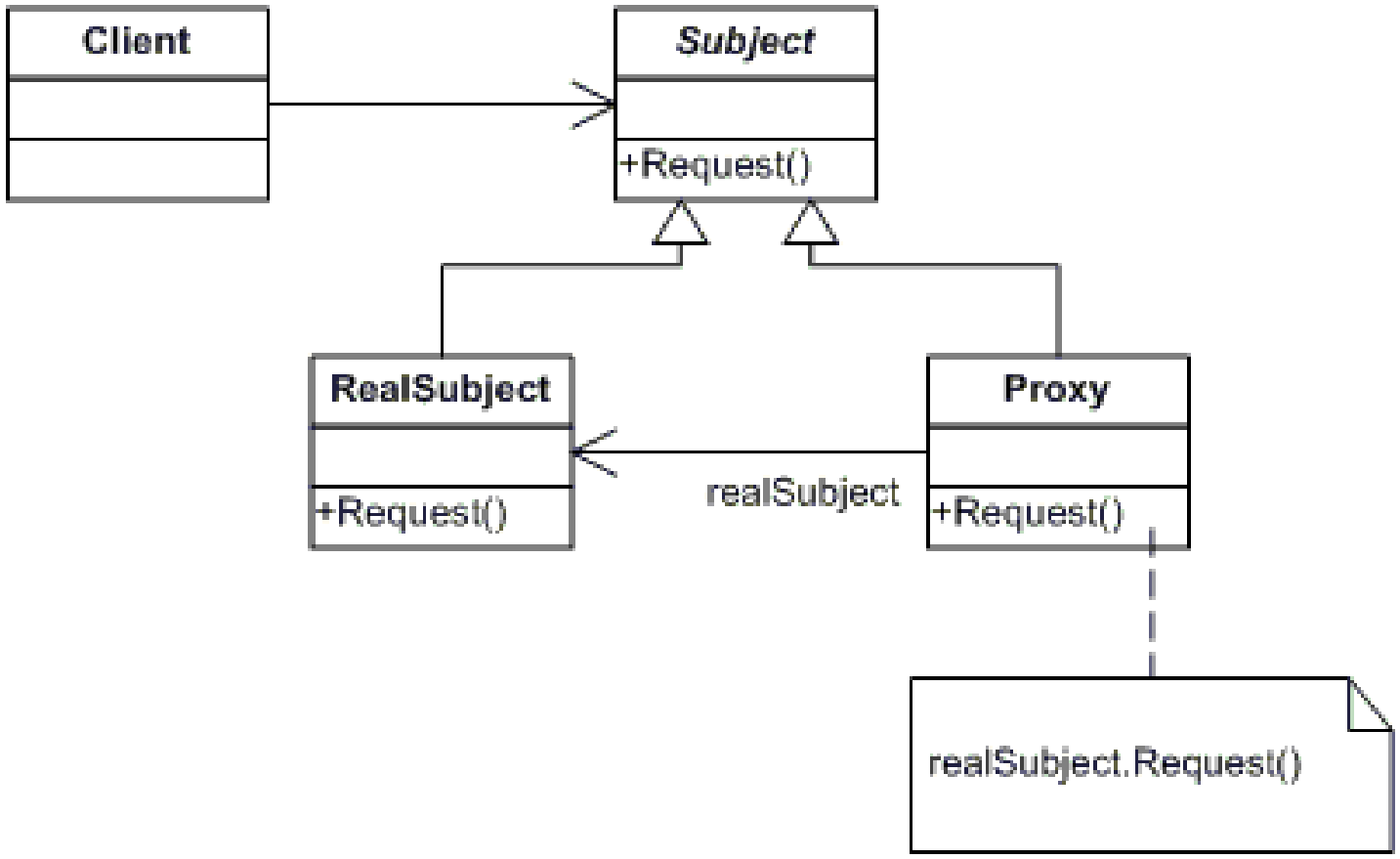

Payment
+Amount()
RealSubject
FundsPaidFromAccount
CheckProxy

Proxy Застосування

- *remote proxy* (віддалений замісник) – локальний представник замість об’єкта, який знаходиться в іншому адресному просторі; може закодовувати запит
- *virtual proxy* – утворює складний об’єкт за вимогою (коли це дійсно потрібно); може попередньо кешувати частину даних об’єкта
- *copy-on-write proxy* – при виконанні команди збереження реально оновлює об’єкт на диску лише за умови, що попередньо збережене значення застаріло
- *protection, firewall proxy* – контролює доступ до деякого об’єкта
- *smart reference proxy* – заміна звичайного вказівника (підрахунок посилань на об’єкт, завантаження в пам’ять за вимогою, перевірка і встановлення блокування на реальний об’єкт, щоб він не був змінений іншим об’єктом…)

Proxy

Структура: діаграма класів



Proxy

❑ **Учасники**:

- **Proxy** – містить посилання, через яке має доступ до реального об’єкта **RealSubject** ;
- **Subject** – задає спільний з **RealSubject** інтерфейс, тому може використовуватись замість нього у відповідному контексті
- **RealSubject** визначає реальний об’єкт, на який може бути посилання в **Proxy**

❑ **Відношення**

Proxy переадресовує при необхідності запити об’єктові **RealSubject**, деталі залежать від виду **Proxy**

Proxy Результати

- Визначає ще один рівень (непрямого) доступу до об’єкта, що може мати накладні затрати при звертанні
- Спосіб реалізації задає різні види **Proxy**, які зумовлюють відповідні переваги та недоліки використання

Proxy Реалізація

- У випадку C++ окремі види (не всі, напр., віртуальний не підходить) **Proxy** зручно реалізовувати, перевантажуючи оператор доступу ->
- Якщо з реальними об’єктами можна працювати через абстрактний інтерфейс, то для них можна розробити ієрархію класів, похідних від **Subject**

patterns

http://www.vincehuston.org
http://sourcemaking.com
http://www.dofactory.com