XV. Design Patterns

Laurea Triennale in Informatica – Corso di Ingegneria del Software I – A.A. 2006/2007
Andrea Polini

Objectives

- What Design Patterns are
- Why we need Design Patterns?
- Discuss some common examples

Software Reuse

- System getting more and more complex .... **reuse as a way to manage complexity**
- However reuse is far from being an easy task!!
- Reuse at **different level of abstraction**
  - At the code level – e.g. libraries, CBSE, inheritance
  - At the application level – e.g. wrapping of legacy systems
  - ...
- Today focus is on reuse at a **more abstract level**
- Design reusable software requires to define interfaces, inheritance and relations among elements to be adapted in different context
  - Experience make designers good in reusing “*deja-vu effect*”

Design Patterns

- Christopher Alexander says "Each pattern describes a **problem which occurs over and over again** in our environment, and then **describes the core of the solution** to that problem, in such a way that **you can use this solution a million times over**, without ever **doing it the same way twice**"
- ...worth noting that Alexander is a building architect :-)

- The idea is to establish the same concept within the software domain
- Defines design pattern that systematically **names, explains, and evaluates an important and recurring design in OO systems**

Design Patterns definition

- A pattern has four essential elements:
  - The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
  - The **problem** describes when to apply the pattern. It explains the problem and its context.
  - The **solution** describes the elements that make up the design, their relationship, responsibilities, and collaborations.
  - The **consequences** are the result and the trade-offs of applying the pattern
- Definition: design patterns are **descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context**

Documenting Software Using Design Pattern

- A DP identifies the classes and **instances, their roles and collaborations, and the distribution of responsibilities**
- When well established in a particular domain Design Patterns become particularly powerful instrument to **document software**
  - JUnit a framework for testing completely illustrated using patterns

- A bit of history
  - Erich Gamma PhD thesis
  - Successively a four person team become well known with the name of “The Gang of Four” - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - “Design Pattern: elements of reusable Object Oriented software” - Addison Wesley
  - A lot of work on patterns started and many collections can be found

Describing Design Patterns

- Definition of **pattern collections** in specific domains is the first basic step to establish reuse of patterns
- GoF defined the first reasonable catalog of general patterns. Each pattern is defined according to:
  - Name
  - Intent
  - Aka
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns

- GoF defined 23 Design Patterns and classified them according to two different concepts:
  - **Scope – object, class**
  - **Purpose – creational, structural, behavioral**

- Possible to organise them in different ways - e.g. How they relate to each other.

GoF Classification

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	***Adapter (class)***	Interpreter
				Template Method
	Object	***Abstract Factory***	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Façade	Memento
			Flyweight	***Observer***
			Proxy	State
				Strategy
				Visitor

GoF – Design Patterns: elements of reusable OO software

Design Patterns and the Design Phase

- Many object in the design come from the analysis model. But OO designs often end up with **classes that have no counterparts in the real world**. For instance array or at a more abstract level composite pattern. Abstraction make design flexible.
- Design Patterns help you identify **less-obvious abstractions** and the objects that can capture them.
  - For example objects that represent a process or **algorithm don't occur in nature, yet they are a crucial part of flexible designs** (Strategy pattern describe how to implement interchangeable families of algorithms).

OO Design Principles Fostered by DP

- Class inheritance vs. Interface inheritance
  - A Class provide implementation – an interface defines a type
  - In case of class, inheritance is mainly a way to reuse code
  - In case of interface inheritance just define a subtype
- **Many languages does not make really any difference** (C++, Eiffel) **some does** (Java, C#) - many patterns rely on this distinction
- Reuse and subclass relations are not the same concept
  - **Client remain unaware of the specific types of objects they use**, as long as the object adhere to the interface that clients expect
  - Clients remain unaware of the **classes that implement these objects**.

Program to an interface, not an implementation

Inheritance vs. Composition

- Class inheritance kind of white-box reuse
- Object composition kind of black-box reuse
- Inheritance, object composition...which one is better?
- Inheritance:

+ defined statically at compile time

+ directly supported by OO programming languages

+ easy to modify the implementation to be reused

- cannot be changed at run-time

- inheritance breaks encapsulation

- dependences on the ancestor put constraint on the reuse of subclass.

Inheritance vs. Composition

- Object Composition:

+ defined at run-time through objects acquiring references to other objects

+ does not break encapsulation

+ use of interface reduce number of dependencies

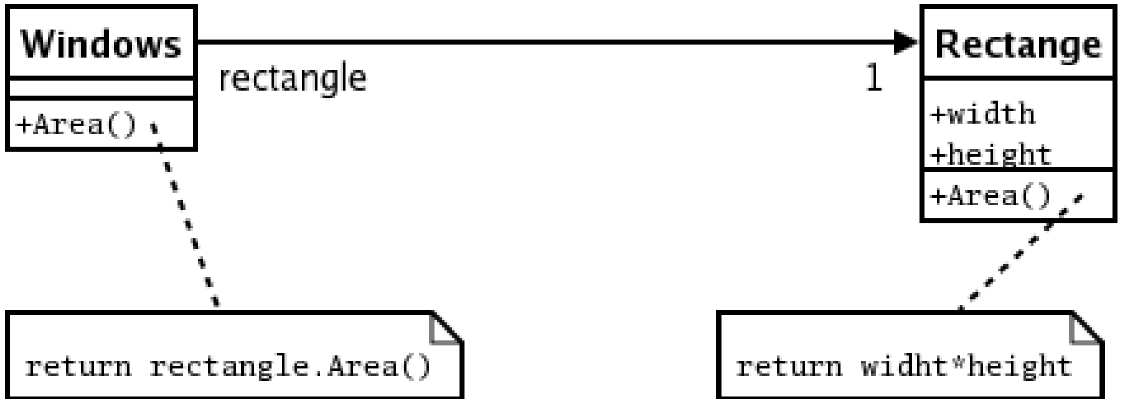
+ helps to keep classes focused on one task

- more objects at run-time and control is more distributed

Favor object composition over inheritance

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance
- Objects receive invocation and **delegate their execution** to other objects (*has* relation, not *is*)



- Main advantage is that it make easy to **compose behaviours at run-time and to change the way they are composed.**
- Highly dynamic software is **harder to understand**

Designing for change

- DP indica le linee evolutive dei sistemi e ne facilità l'evoluzione.

  Problemi comuni nei sistemi OO sono ad esempio:

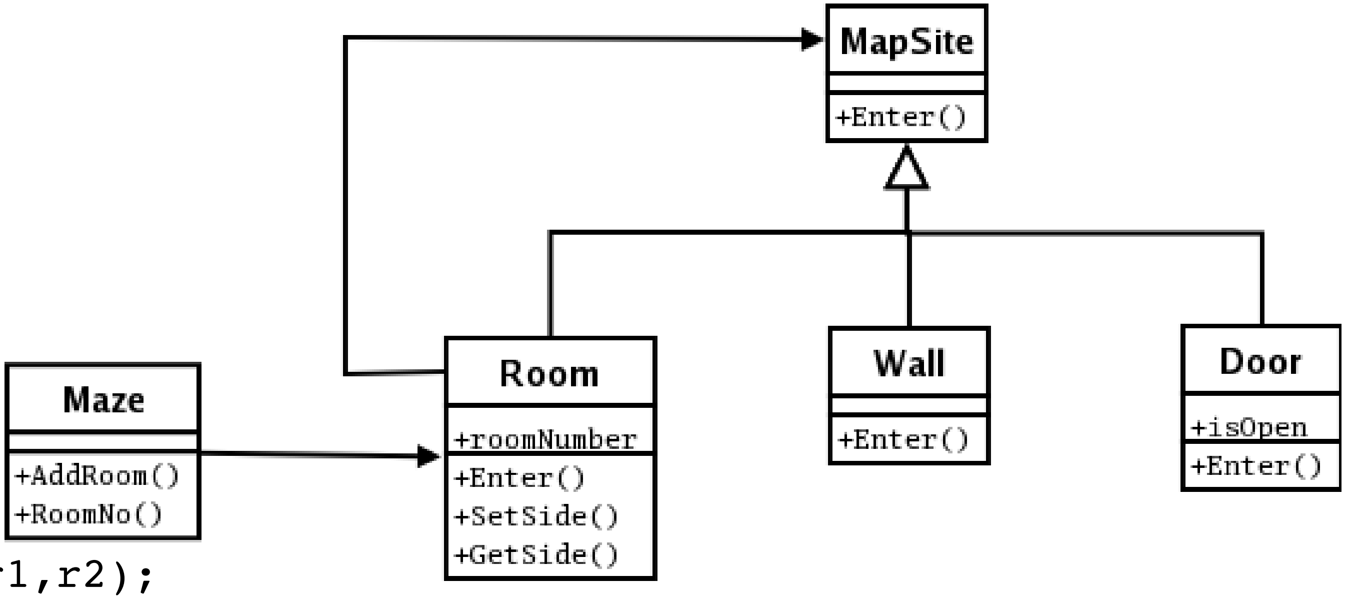
1. Creating a class by specifying a class explicitely – *complicate future changes* (Abstract Factory)
2. Dependence on specific operations – *avoiding hard coded request, you make it easier to change*
3. Dependence on hardware and software platform – *better to limit platform dependences* (Abstract Factory)
4. Dependence on object representations or implementations – *hiding this information from clients keeps changes from cascading* (Abstract Factory, Proxy)
5. Algorithmic dependencies – *better isolate algorithms that will have to change* (Strategy)
6. Tight Coupling – *loose coupling reduces modifications and increase reusability* (Abstract Factory, Observer)
7. Extending functionality by subclassing – *not always easy* (Observer)
8. Inability to alter classes conveniently – *difficult to adapt to many different requests* (Adapter)

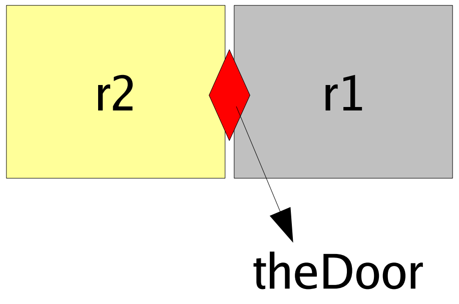
Creational Patterns

- Abstract the instantiation process
- **Objective** of these “pattern category” is to make a system independent on how its objects are created, composed and represented
- They permit to rely more on composition than class inheritance
- **Creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when**
- Two creational patterns will be the subject of our study:
  - Abstract Factory
  - Singleton

Maze example

```
Maze MazeGame() {
  Maze aMaze = new Maze();
  Room r1 = new Room();
  Room r2 = new Room();
  Door theDoor = new Door(r1,r2);

  aMaze.AddRoom(r1); aMaze.AddRoom(r2);

  r1.SetSide(N, new Wall()); r1.SetSide(E, new Wall());
  r1.SetSide(S, new Wall()); r1.SetSide(O, theDoor);

  r1.SetSide(N, new Wall()); r1.SetSide(E, theDoor);
  r1.SetSide(S, new Wall()); r1.SetSide(O, new Wall());
  return aMaze;
}
```





Singleton

- Intent:
  - Ensure a class only has **one instance**, and provide a **global point of access to it**
- Motivation:
  - It's important for some classes to have only one instance.

    There should be only one printer spooler in a system, there should be one window manager, one file system manager.

    How do we ensure that a class has only one instance and that the instance is easily accessible? **Global variable make an object accessible but doesn't avoid the instantiation of more instances**
- Participants:
  - **Singleton:** defines an instance operation that lets clients access its unique instance. May be responsible for creating its own unique instance.

Singleton

- Consequences:
  - **Controlled access to sole instance** – strict control over the clients
  - **Reduced name space** – avoid the usage of global variables
  - **Permits refinement of operations and representation** – singleton may be subclassed
  - **Permits a variable number of instances** – easy to permit more instances
  - **More flexible then using class operations** – easier to maintain

Singleton

- Java Sample Code:

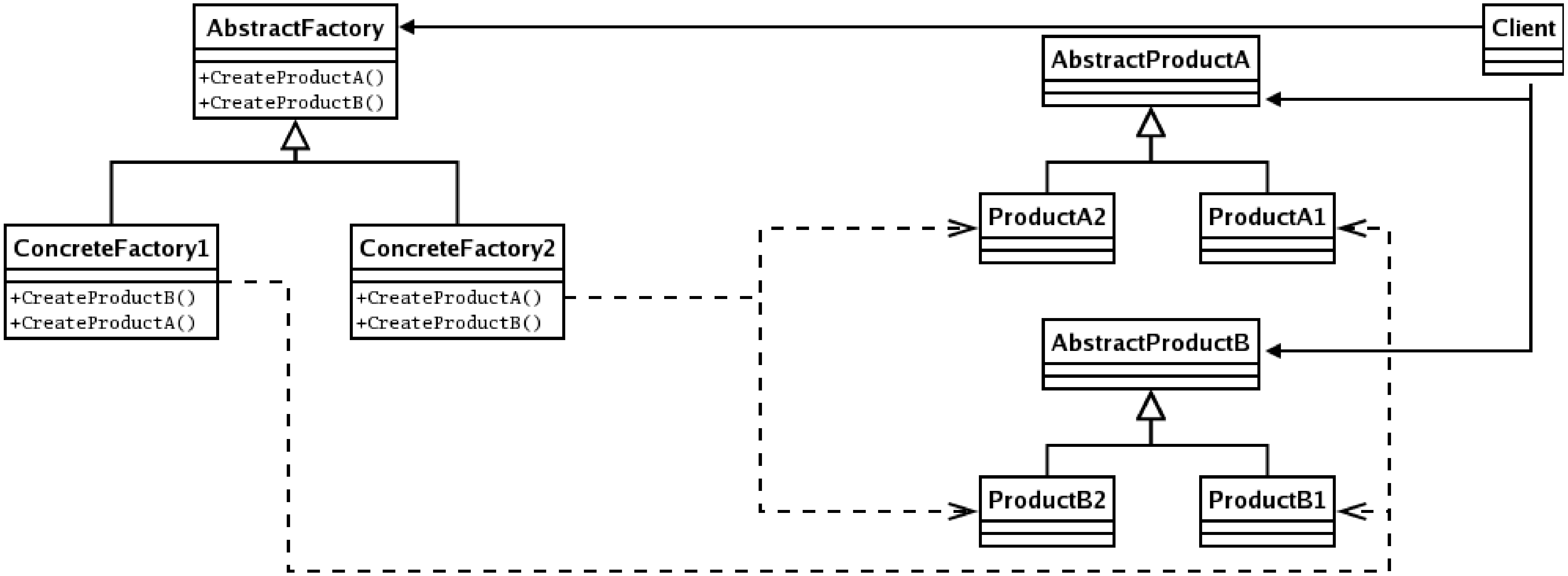
```
public class Singleton {
  protected static Singleton instance = null;
  private static int counter = 0;

  ... // objects attributes and methods

  protected Singleton() { ... }
  public static Singleton Instance() {
    if (counter == 0) {
    instance = new Singleton(); counter++;
    }
    return instance;
  }
}
```

Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Applicability
  - **A system should be independent of how its products are created, composed, and represented**
  - A system should be configured with **one of multiple families of products**
  - A **family of related product objects is designed to be used together, and you need to enforce this constraint**
  - You want to provide a **class library of products, and you want to reveal just their interfaces, not their implementation**

Abstract Factory

- Structure



Abstract Factory

Java Sample Code

```
public interface MazeFactory {
  Maze MakeMaze();
  Wall MakeWall();
  Door MakeDoor();
  Room MakeRoom();
}

public class MazeGame(MazeFactory mf) {
  Maze aMaze = mf.MakeMaze(); Room r1 = mf.MakeRoom(1);
  Room r2 = mf.MakeRoom(2); Door aDoor = mf.MakeDoor(r1,r2);
  aMaze.AddRoom(1); aMaze.addRoom(2);
  r1.SetSide(N,mf.MakeWall()); r1.SetSide(S,mf.MakeWall());
  r1.SetSide(E,mf.MakeWall()); r1.SetSide(O,aDoor);
  r2.SetSide(N,mf.MakeWall()); r2.SetSide(S,mf.MakeWall());
  r2.SetSide(E,aDoor); r2.SetSide(O,mf.MakeWall());
  return aMaze;
}
```

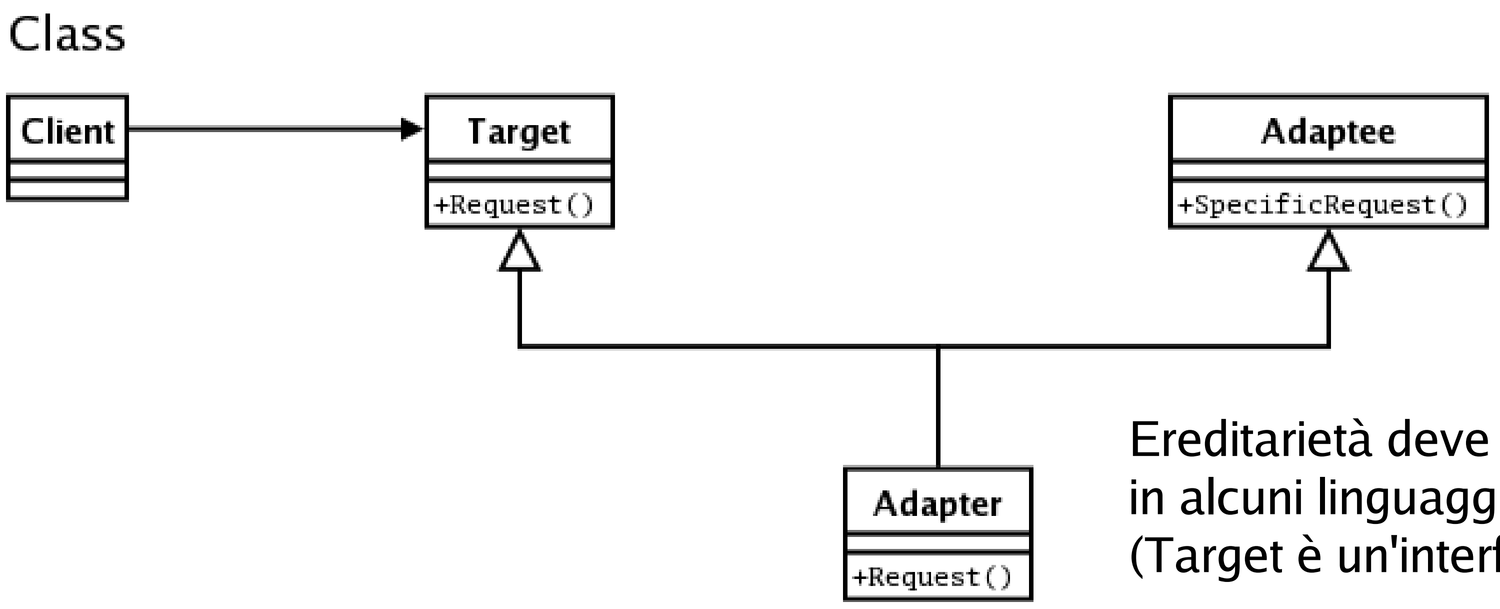
Structural Patterns

- How to compose classes and objects to create more complex structure
  - Adapter
  - Proxy

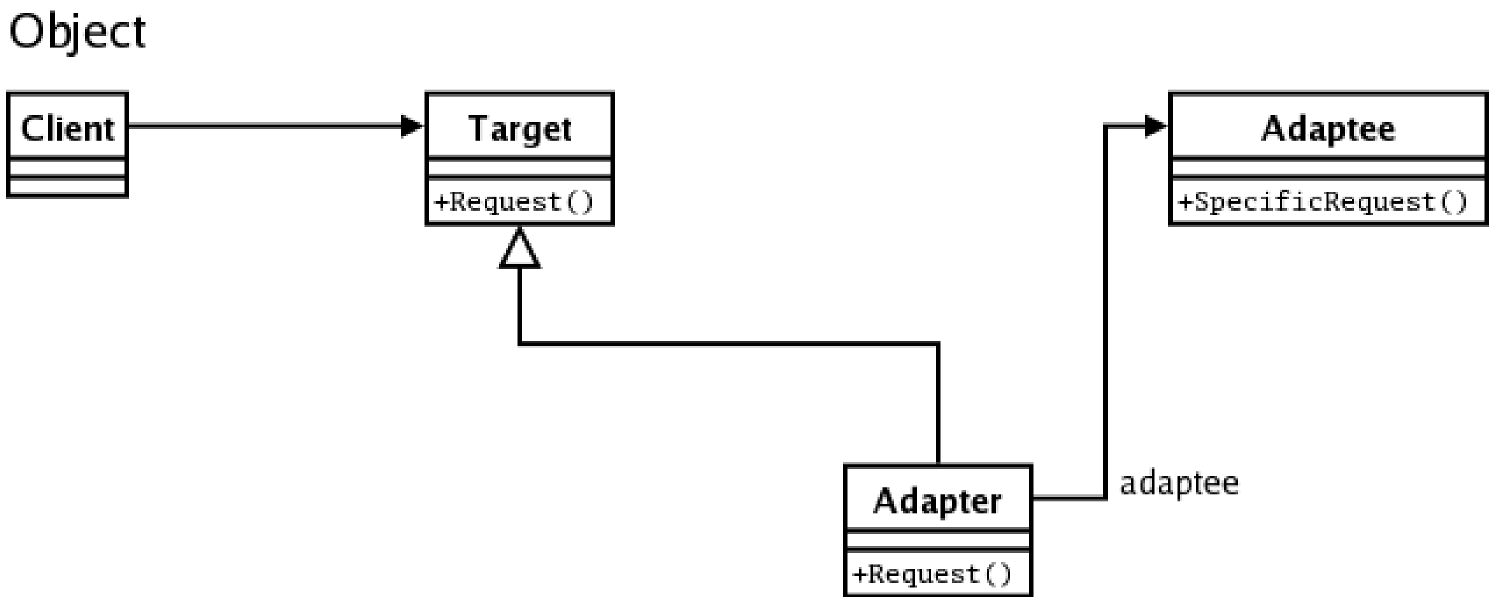
Adapter

- Intent
  - **Convert the interface of a class into another interface clients expect**. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Applicability
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
  - You need to use several existing subclasses, but it is impractical to adapt their interface by subclassing every one. An object adapt the interface of its parent class.

Adapter



Ereditarietà deve essere singola in alcuni linguaggi
(Target è un'interfaccia)

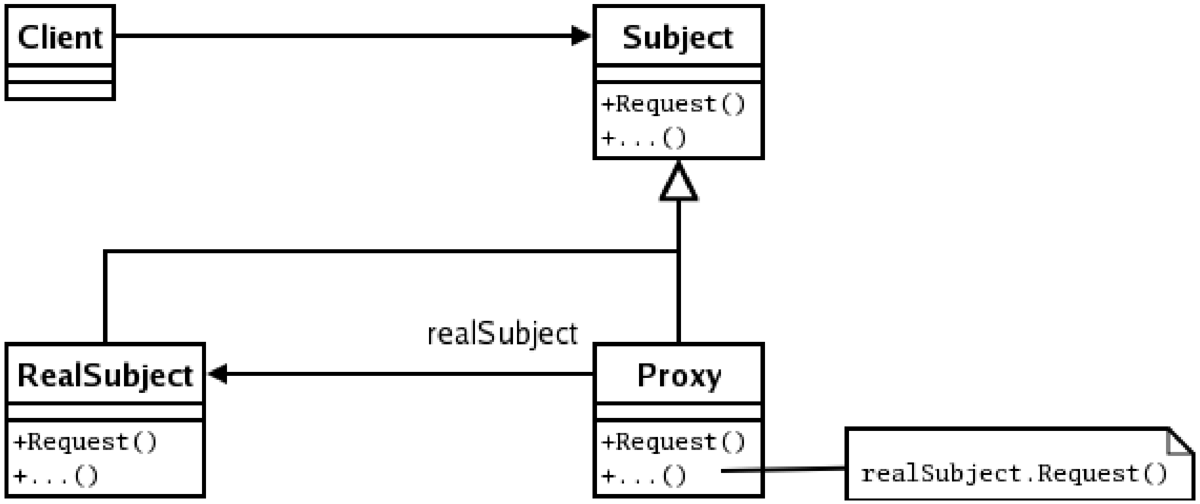


Proxy

- Intent
  - Provide a **surrogate or placeholder for another object to control access to it**
- Motivation
  - One reason for controlling access to an object is to defer full cost of its creation and initialization until we actually need to use it.

    e.g. Big objects not always necessary – put a placeholder without increasing editor complexity
- Applicability
  - A **remote proxy** provides a local representative for an object in a different address space
  - A **virtual proxy** creates expensive objects on demand
  - A **protection proxy** control access to the original object
  - A **smart reference** is a replacement that performs additional actions

Proxy



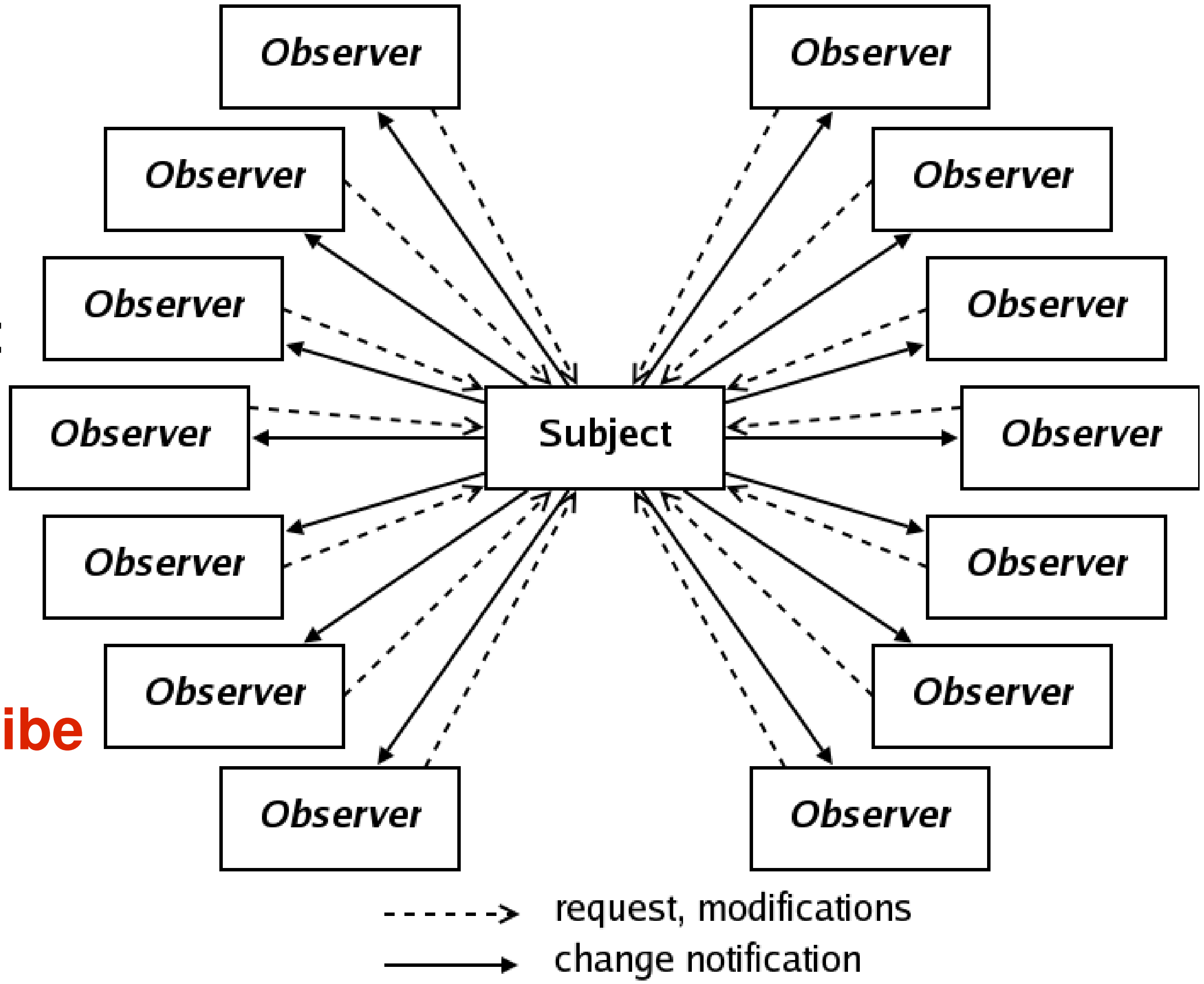
```
public interface Graphic {...}

public class Image implements Gaphic {
  Image(FileInputStream fis) {...}; ...
}

public class ImageProxy implements Graphic {
  String fileName;
  public void Draw(at) { Image i = new Image(fis); i.Draw(at); }
}
```

Behavioral Patterns

- Behavioral Patterns are concerned with algorithms and the assignment of responsibilities between objects.
- These patterns characterize complex control flow that is difficult to follow at run-time – move the focus from control to interconnection
  - Observer
  - Strategy

Observer

- Intent
  - **Define a one-to-many dependency between objects so that when one object change state, all its dependents are notified and updated automatically**
- Motivation
  - Need to maintain consistency between related objects.
- Key objects in this pattern:
  - **Observer and Subject**
- Kind of interaction is also known as **publish-subscribe**

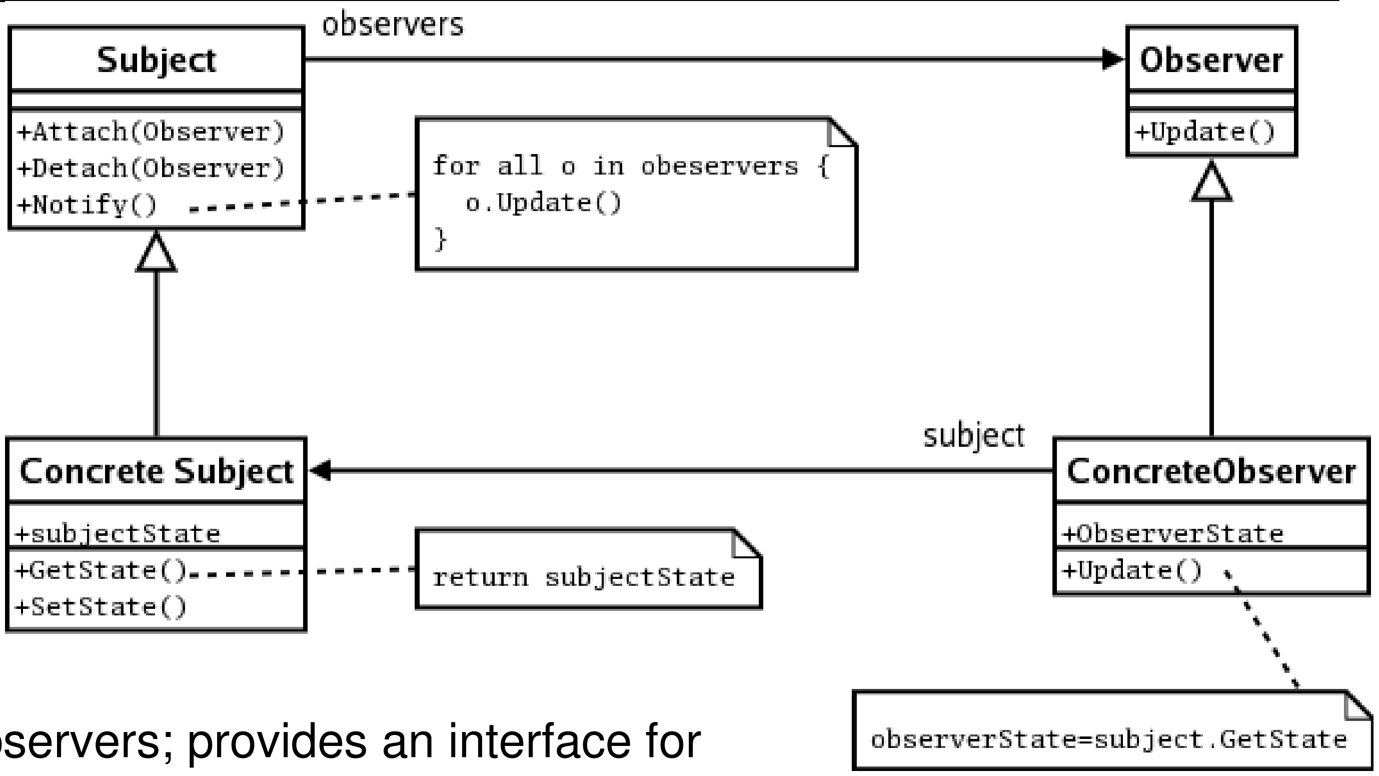


Observer

- Applicability
  - When an abstraction has **two aspects, one dependent on the other**. Encapsulating these aspects in separate objects lets you vary and reuse them independently
  - When a **change to one object requires changing others**, and you don't know how many objects need to be changed
  - When an object **should be able to notify other objects without making assumptions about who these objects are**. In other words, you don't want these objects being tightly coupled

Observer

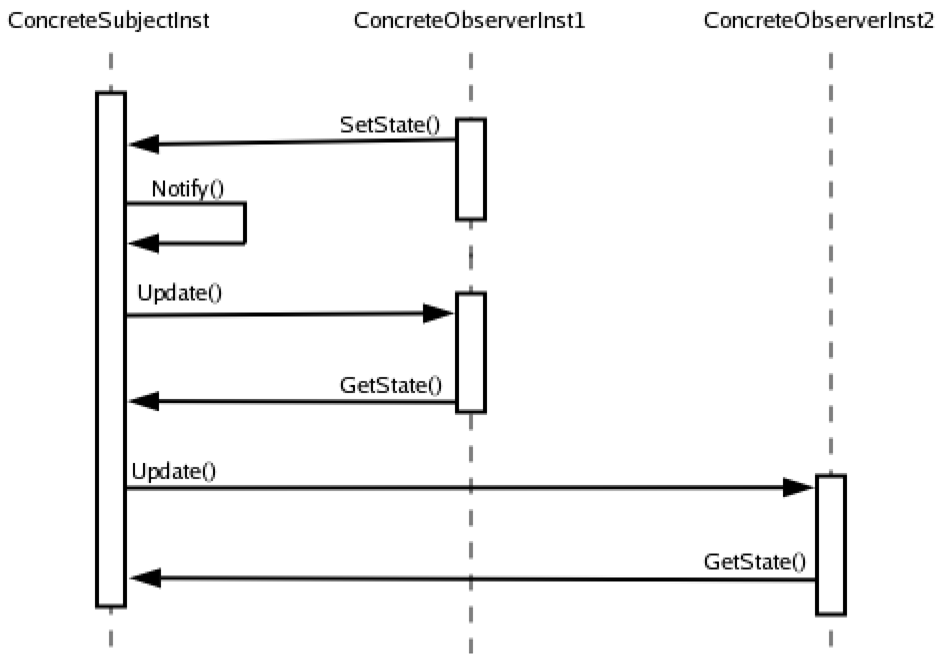
- Structure



- Participants
  - Subject: knows its observers; provides an interface for attaching and detaching
  - Observer: defines an updating interface for objects that should be notified
  - ConcreteSubject: stores state; sends a notification to its observers
  - ConcreteObserver: maintains a reference to a concrete subject; stores state that should stay consistent with the subject's; implements the observer

Observer

- Collaborations:



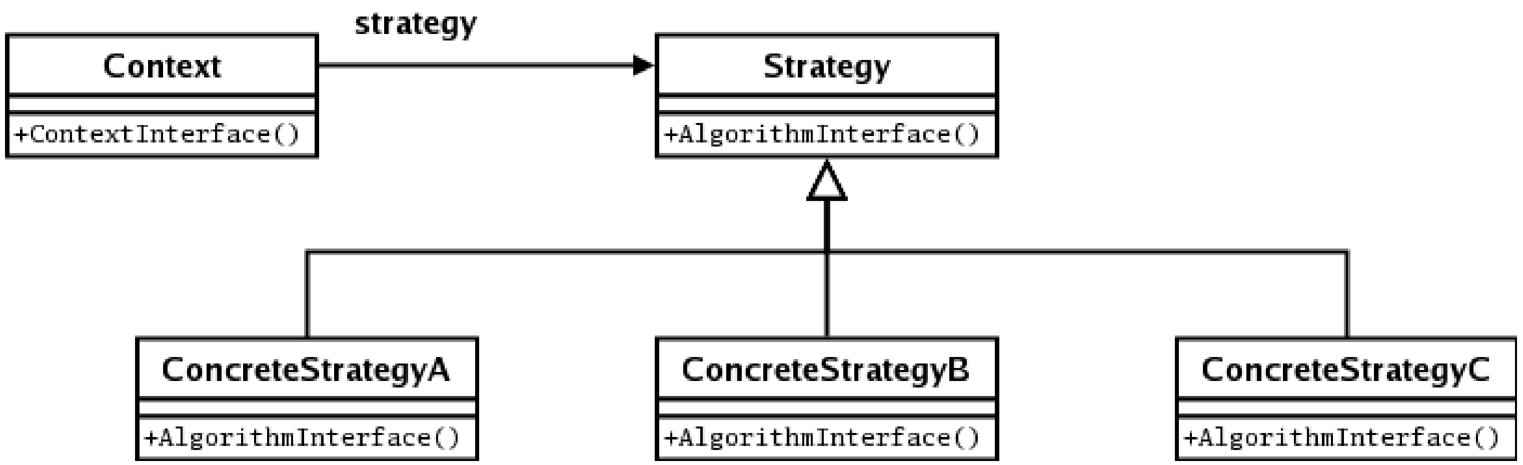
- Implementation:
  - Observing more that one subject
  - Dangling references to delete subjects
  - Pull vs. Push protocol model

Strategy

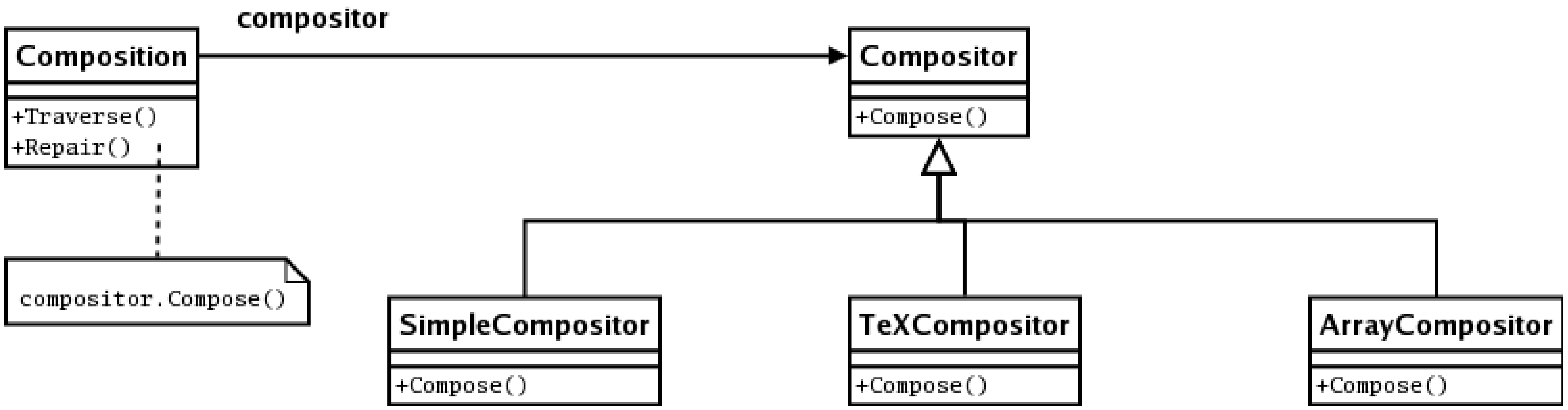
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets algorithm vary independently from clients that use it.
- Applicability
  - Many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours
  - You need different variants of an algorithm.
  - An algorithm uses data that clients shouldn't know about. Use the strategy pattern to avoid exposing complex, algorithm-specific data structures
  - A class define many behaviours, an these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy

- Structure



- Motivating example



Strategy

```
public interface Compositor {
  public FormattedComponent[] Compose(String text,int linewidth,intlineheight);
}

public class Composition {
  private Compositor comp; private int linewidth;
  private int lineheight; private String text;

  public Composition(Compositor c, int lw, int lh) {
    this.comp = c;this.lineheight=lh; this.linewidth=lw;  }

  public FormattedComponent[] Repair() {
    return comp.Compose(text, linewidth, lineheight);
  }
}
```

Strategy

```
public class SimpleCompositor implements Compositor {
  public FormattedComponent[] Compose(String text,int linewidth,int lineheight)
     { // TODO Auto-generated method stub
       return null;  }
}

public class TeXCompositor implements Compositor {
  public FormattedComponent[] Compose(String text,int linewidth,int lineheight)
     { // TODO Auto-generated method stub
       return null;  }
}

public class ArrayCompositor implements Compositor {
  public FormattedComponent[] Compose(String text,int linewidth,int lineheight)
    { // TODO Auto-generated method stub
      return null;  }
}
```

Strategy

```
public class FormatterUser {
  private Composition simple;
  private Composition TeX;
  private Composition array;
  ...
  public void setCompositions() {
    simple = new Composition(new SimpleCompositor(),0,0);
    TeX = new Composition(new TeXCompositor(),0,0);
    array = new Composition(new ArrayCompositor(),0,0);
  }
}
```