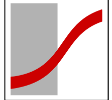
Introducing Design Patterns

> ***Provides pointers to students***



Prof. T.H. Tse
Department of Computer Science
Email: thtse@cs.hku.hk
Web: hku.hk/thtse

The Need for Design Patterns

- ***New designers*** are overburdened by the options available, and fall back to non-OO techniques
- ***Experienced designers*** make good designs
  - They do not solve every problem from first principles
  - No need to reinvent the wheel
  - ***Reuse*** solutions that have worked for them

> ***Let us reuse solutions of experienced designers***

Design Patterns

- Target for a ***general design problem*** in a ***particular context***
- Identify common ***class structures*** for reusable OO designs
- Show ***participating objects***, ***collaborations***, and ***division of labour***

> ***Examples***
> - ***Sort by subject***
> - ***Sort by date***

Usefulness of Design Patterns

- Many objects come from the analysis model
- But designs often need classes with no counterparts in the real world
- Design patterns help to identify less-obvious abstractions and objects

Other Usefulness of Design Patterns

- Relate ***compile-time*** and ***run-time*** structures
- Enhance ***delegation*** — *Via **encapsulation***
- Enhance ***reuse*** — ***Via inheritance***

Other Usefulness of Design Patterns

Design for change: ***Avoid unnecessary redesigns*** due to

- ☹ ***Specific*** object representation
- ☹ ***Specific*** operations, implementation, and algorithms
- ☹ ***Dependence*** on hardware and software platforms
- ☹ ***Tight*** coupling — ***Classes depend on one another***
- ☹ ***Cannot*** replace classes easily

Design Patterns ≠ Frameworks

- ***Frameworks*** are another source of reference for ***experienced*** design
- A framework`k is a partially completed software system customized for a ***specific*** applicationx

  Example
  HSBC e-banking system
- Design patterns are ***more general*** than frameworks
  - A pattern can be used in any kinds of applications

Design Patterns ≠ Frameworks

- Design patterns are ***more fundamental*** than frameworks
  - Frameworks are in compilable programming languages
  - Design patterns are language independent and have to be implemented every time they are used
  - Design patterns also explain the intent, trade-offs, and consequences of a design — ***Learn how to learn***

Note

for those who wish me to write down what I say

- It is fairly difficult to teach design patterns by means of lectures
- It involves hands-on experience with real systems
- I can only go through the concepts, philosophy, and a few examples

Design Patterns

4 Essential Elements

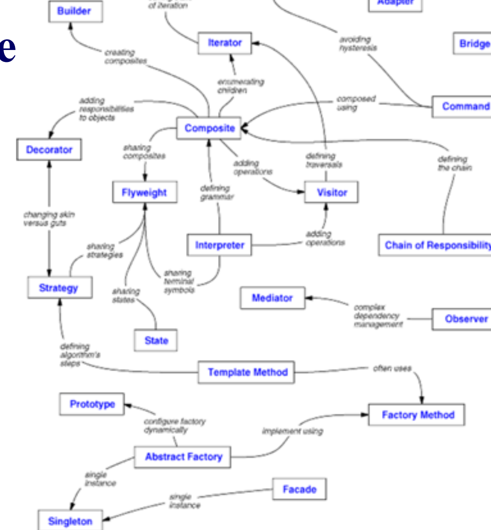
- ***The Problem***
  - Describes when to apply the pattern
  - Explains the problem and its context
  - May include a list of conditions for application
- ***Pattern Name***
  - Allows us to design at a higher level of abstraction
  - Free from the details

Design Patterns

4 Essential Elements (Continued)

- ***Solution***
  - Describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - Not a particular concrete design or implementation, but a general arrangement of elements
- ***Consequences***
  - Results and (space and time) trade-offs, useful for evaluating design alternatives

Example



Example (Continued)

GoF Design Patterns

- ***Gang-of-Four:*** Gamma, Helm, Johnson, and Vlissides
- Three categories
  - ***Behavioural*** How to build powerful behaviour
  - ***Creational*** How to build complex objects
  - ***Structural*** How to build flexible structures .

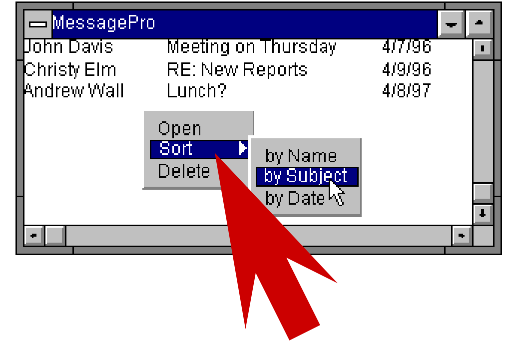
About the GoF Design Patterns

- ***Gang-of-Four:*** Gamma, Helm, Johnson, and Vlissides
- Three categories
  - ***Behavioural*** **How to build powerful behaviour**
  - ***Creational*** How to build complex objects
  - ***Structural*** How to build flexible structures.

Behavioural Pattern Example

Strategy

Consider Sorting of Meeting Schedule



Behavioural Pattern Example

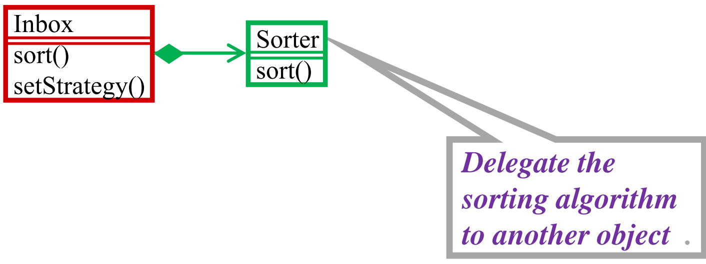
Strategy

Traditional Technique

Behavioural Pattern Example

Strategy

Object-Oriented Design



Behavioural Pattern Example

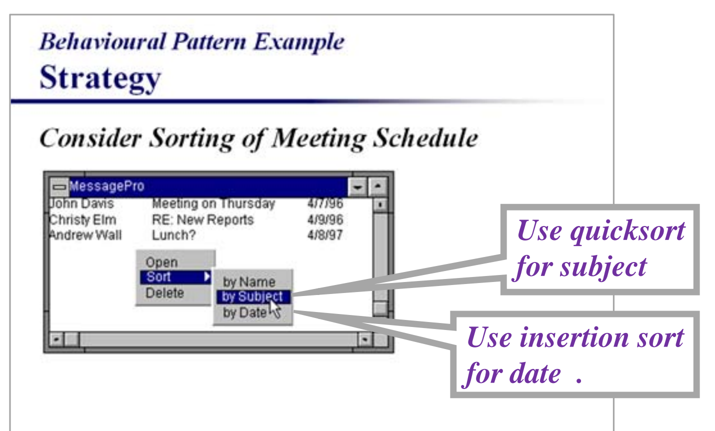
Strategy

Problem

- We want an ***adaptive*** method with ***various*** implementations selected at ***run time***
- We need ***different*** algorithms to accomplish the same task in ***different*** circumstances.

Behavioural Pattern Example

Strategy



Behavioural Pattern Example

Strategy

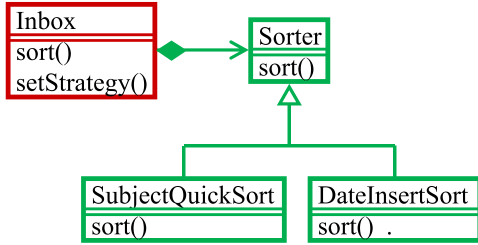
Design Pattern Solution

- Define a ***family*** of algorithms, and ***encapsulate*** each one
- The ***strategy*** pattern lets the algorithm ***vary*** according to clients ...

Behavioural Pattern Example

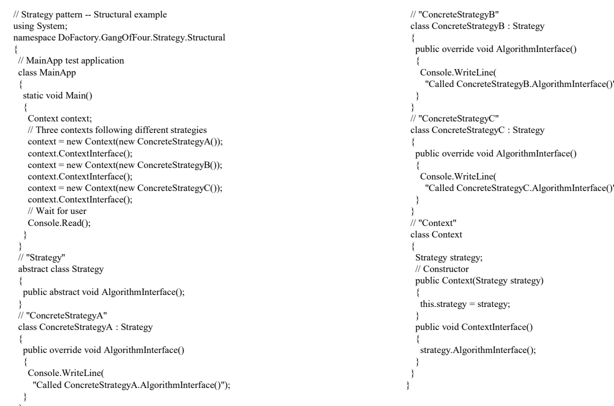
Strategy

Design Pattern Solution (Continued)



Behavioural Pattern Example: Strategy

Potential Implementation in C#



```
// Strategy pattern -- Structural example
using System;
namespace DoFactory.GangOfFour.Strategy.Structural
{
 // MainApp test application
 class MainApp
 {
  static void Main()
  {
   Context context;
   // Three contexts following different strategies
   context = new Context(new ConcreteStrategyA());
   context.ContextInterface();
   context = new Context(new ConcreteStrategyB());
   context.ContextInterface();
   context = new Context(new ConcreteStrategyC());
   context.ContextInterface();
   // Wait for user
   Console.Read();
  }
 }
 // "Strategy"
 abstract class Strategy
 {
  public abstract void AlgorithmInterface();
 }
 // "ConcreteStrategyA"
 class ConcreteStrategyA : Strategy
 {
  public override void AlgorithmInterface()
  {
   Console.WriteLine(
    "Called ConcreteStrategyA.AlgorithmInterface()");
  }
 }
 // "ConcreteStrategyB"
 class ConcreteStrategyB : Strategy
 {
  public override void AlgorithmInterface()
  {
   Console.WriteLine(
    "Called ConcreteStrategyB.AlgorithmInterface()");
  }
 }
 // "ConcreteStrategyC"
 class ConcreteStrategyC : Strategy
 {
  public override void AlgorithmInterface()
  {
   Console.WriteLine(
    "Called ConcreteStrategyC.AlgorithmInterface()");
  }
 }
 // "Context"
 class Context
 {
  Strategy strategy;
  // Constructor
  public Context(Strategy strategy)
  {
   this.strategy = strategy;
  }
  public void ContextInterface()
  {
   strategy.AlgorithmInterface();
  }
 }
}
```


About the GoF Design Patterns

- ***Gang-of-Four:*** Gamma, Helm, Johnson, and Vlissides
- Three categories
  - ***Behavioural*** How to build powerful behavior
  - ***Creational*** **How to build complex objects**
  - ***Structural*** How to build flexible structures .

Creational Pattern Example

Abstract Factory

Problem

- Sometimes constructing an object is very complicated
  - The object is made of many different kinds of complexly interconnected parts
  - The object is made from two or more sets of compatible parts, but the sets are incompatible to each other .

Creational Pattern Example

Abstract Factory

Application: Graphical User Interface

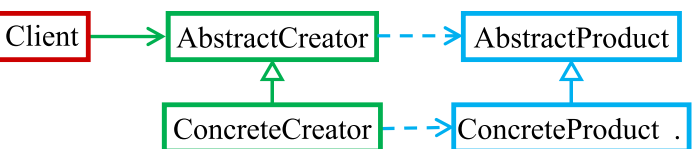
- Constructing user interface ***widgets***
  - Windows, scrollbars, radio buttons, menus, …
- Various ***platforms*** to support
  - Microsoft Windows, Apple macOS, Apple iOS, Unix-like X Window system .

Creational Pattern Example

Abstract Factory

General Recommendation

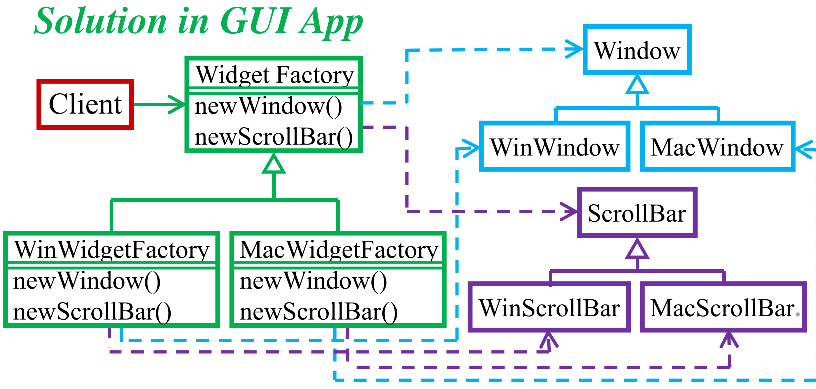
- Use an abstract class (called ***pure fabrication***) to do the construction



Creational Pattern Example

Abstract Factory

Solution in GUI App



Creational Pattern Example: Abstract Factory

Potential Implementation in C++

```
#include <memory>
using std::auto_ptr;
class Control { };
class PushControl : public Control { };
class Factory {
 public:
   // Returns Factory subclass based on classKey.
   // Each subclass has its own getControl() implementation.
   // This will be implemented after the subclasses have been declared.
   static auto_ptr<Factory> getFactory(int classKey);
   virtual auto_ptr<Control> getControl() const = 0;
};
class ControlFactory : public Factory {
public:
   virtual auto_ptr<Control> getControl() const {
     return auto_ptr<Control>(new PushControl());
   }
};
auto_ptr<Factory> Factory::getFactory(int classKey) {
  // Insert conditional logic here.  Sample:
  switch(classKey) {
    default:
      return auto_ptr<Factory>(new ControlFactory());
  }
}
```

About the GoF Design Patterns

- *Gang-of-Four:* Gamma, Helm, Johnson, and Vlissides
- Three categories
  - *Behavioural* How to build powerful behaviour
  - *Creational* How to build problematic objects
  - ***Structural*** **How to build flexible structures** .

Structural Pattern Example

Proxy

Common Types of Proxies

- ***Virtual Proxy***
  - Lightweight object that creates heavyweight object on demand
- ***Device Proxy***
  - Logical device that manages physical device
- ***Remote Proxy***
  - Local representation of remote object
- ***Protection Proxy***
  - Sentry (soldier) that guards secure object

Printer

Printer

Search engine

Search engine .

Structural Pattern Example

Proxy

General Recommendation

- Create an abstract class to specify the logical interface to the object
- Create two subclasses
  - The ***proxy*** and the ***actual object class***
- The proxy forwards messages to actual object
- Actual object returns messages through the proxy after actual operation .

Structural Pattern Example

Proxy

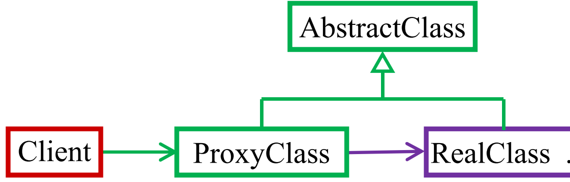
Problem

- We do not want to directly access an object, but through an ***intermediary*** or ***agent***
  - This may arise from reliability or complexity concerns .

Structural Pattern Example

Proxy

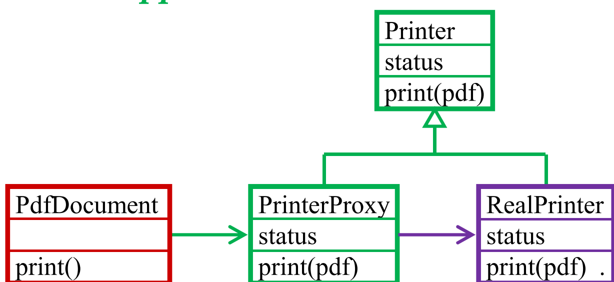
General Recommendation



Structural Pattern Example

Proxy

Printer Application



Structural Pattern Example: Proxy

Potential Implementation in Java

```
import java.util.*;
interface Image {
  public void displayImage( );
}
class RealImage implements Image {
  private String filename;
  public RealImage(String filename) {
     this.filename = filename;
     System.out.println("Loading   "+filename);
  }
  public void displayImage( ) { System.out.println("Displaying "+filename); }
}
class ProxyImage implements Image {
  private String filename;
  private RealImage image;
  public ProxyImage(String filename) { this.filename = filename; }
  public void displayImage( ) {
     if (image == null) image = new RealImage(filename);
     // load only on demand
     image.displayImage( );
  }
}
```

```
class ProxyExample {
  public static void main(String[ ] args) {
     ArrayList<Image> images = new ArrayList<Image>( );
     images.add( new ProxyImage("HiRes_10MB_Photo1") );
     images.add( new ProxyImage("HiRes_10MB_Photo2") );
     images.add( new ProxyImage("HiRes_10MB_Photo3") );
     images.get(0).displayImage(); // loading necessary
     images.get(1).displayImage(); // loading necessary
     images.get(0).displayImage();
     // no loading necessary; already done
     // the third image will never be loaded - time saved!
  }
```