

SWEN20003
Object Oriented Software Development

Design Patterns

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne
© University of Melbourne 2020

The Road So Far

- Java Foundations
 - ▶ A Quick Tour of Java
- Object Oriented Programming Foundations
 - ▶ Classes and Objects
 - ▶ Arrays and Strings
 - ▶ Input and Output
 - ▶ *Software Tools and Bagel*
 - ▶ Inheritance and Polymorphism
 - ▶ Interfaces and Polymorphism
- Advanced Object Oriented Programming and Software Design
 - ▶ Modelling Classes and Relationships
 - ▶ Generics
 - ▶ Collections and Maps

Lecture Objectives

After this lecture you will be able to:

- Describe what **design patterns** are, and why they are useful
- Analyse and understand **design pattern specifications**
- Make use of (some) design patterns

Overview

This topic will be delivered through two lectures.

Design Patterns 1: Slides 4-21

- Introduction and Motivation
- Singleton Pattern
- Factory Method Pattern

Design Patterns 2: Slides 22-51

- Template Method Pattern
- Strategy Pattern
- Observer Pattern
- Pattern Types

Software Design

Are you already a good software designer?

If not, how do you become a good software designer?

Follow three simple steps...

How to become a good software designer

Step 1: Learn the fundamentals of programming

- Programming Languages
- Algorithms and Data Structures

Step 2: Learn design paradigms and principles

- Structured Design
- Object Oriented Design

Step 3: Study and mimic the designs of experienced designers

- Good designers reuse solutions - they do not design everything from scratch
- **Design Patterns** systematically document re-occurring design solutions so that they can be reused

Design Patterns

A Software *Design Pattern* is a description of a solution to a recurring problem in software design.

The recurring nature of the problems makes the solution useful to software developers.

Publishing it in the form of a Pattern enables the solution to be used without reinventing the wheel.

Pioneering work on design patterns was done by Eric Gamma and the team who authored the classic book on design patterns (Gang of Four):

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Participants List of classes/objects and their roles in the pattern

Collaboration How the objects in the pattern interact

Consequences A description of the results, side effects, and tradeoffs when using the pattern

Implementation Example of “solving a problem” with the pattern

Known Uses Specific, real-world examples of using the pattern

Design Patterns

Thousands of design patterns exist today.

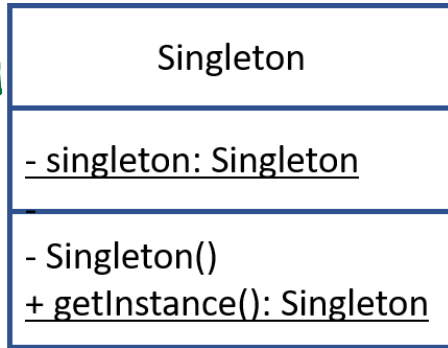
Let us take a look at some common design patterns...

Singleton Pattern

Ensure that a class has **only one instance** and **provide a global point of access to it.**

only single object created

constructor private ?



Singleton Pattern - Implementation

```
class Singleton {  
    private static Singleton _instance = null;  
    private Singleton() {  
        //fill in the blank  
    }  
  
    public static Singleton getInstance() {  
        if ( _instance == null )  
            _instance = new Singleton();  
        return _instance;  
    }  
  
    public void otherOperations() { }  
}
```

check the
instance is
null

Instantiate (rather than use constructor)

⇒ first time create a new object
then the next time will return
the existing one

Singleton Pattern - Collaboration

```
class TestSingleton {  
  
    public void method1(){  
        X = Singleton.getInstance();  
    }  
  
    public void method2(){  
        Y = Singleton.getInstance();  
    }  
}
```

⇒ X and Y point to the same object

Singleton Pattern Analysis

- Intent** Ensure that a class has only one instance and provide a global point of access to it. *make it static*
- Motivation** There are cases where only one instance of a class must be enforced with easy access to the object.
- Applicability** Use when a single instance of a class is required.
- Structure** See previous slides
- Participants** Singleton class.
- Collaboration** See previous slides.
- Consequences** Use it with caution because inappropriate use could result in a bad design. *⇒ violate the principle of information hiding*
- Implementation** See previous slides. *⇒ expose*
- Known Uses** e.g. CacheManager class, PrinterSpooler class. *THIS OBJECT to everyone*

Factory Method Pattern

Creating objects in the class that requires (uses) the objects is inflexible.

- It commits the class to a particular object
- Makes it impossible to change the instantiation without having to change the class.

Factory Method pattern solves this problem by:

- Defining a separate operation for creating an object.
- Creating an object by calling a factory method.

Factory Method Pattern

Keyword

Factory: A *general* technique for *manufacturing* (creating) objects.

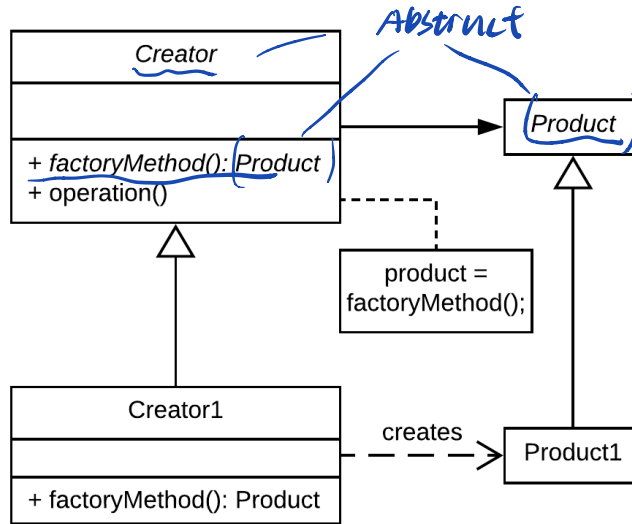
Keyword

Product: An abstract class that generalises the objects *being created/produced* by the factory.

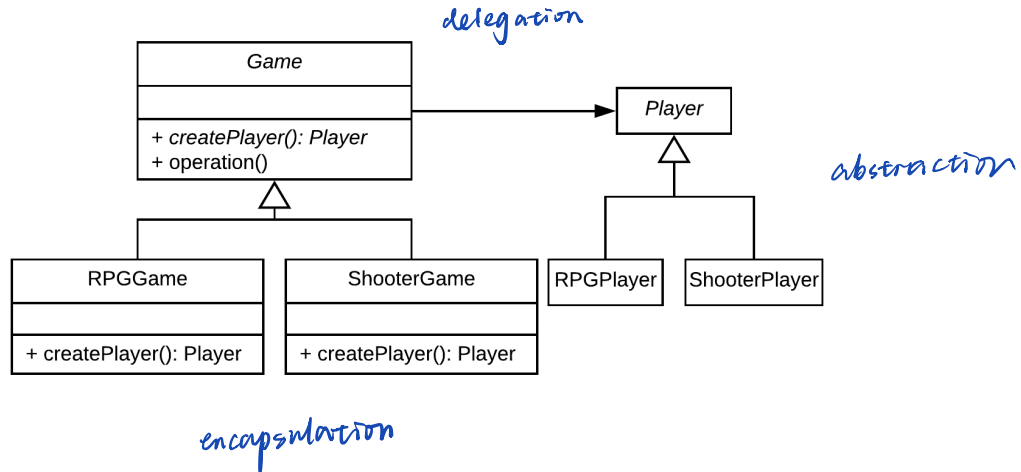
Keyword

Creator: An abstract class that generalises the objects that *will consume/produce* products; generally have some *operation* (e.g. the constructor) that will invoke the factory method.

Factory Method Pattern Structure



Factory Method Pattern Example



Factory Method Pattern Example

```
public abstract class Game {  
    private final List<Player> players = new ArrayList<>();  
  
    public Game(int nPlayers) {  
        for (int i = 0; i < nPlayers; i++) {  
            players.add(createPlayer());  
        }  
    }  
  
    public abstract Player createPlayer();  
}
```

Factory Method Patterns Example

```
public class RPGGame extends Game {
    public RPGGame(int nPlayers) {
        super(nPlayers);
    }
    @Override
    public Player createPlayer() {
        return new RPGPlayer();
    }
}

public class ShooterGame extends Game {
    public ShooterGame(int nPlayers) {
        super(nPlayers);
    }
    @Override
    public Player createPlayer() {
        return new ShooterPlayer();
    }
}
```

```
RPGGame shadowQuest = new RPGGame(4);
```

Factory Method Pattern Motivation

But... What's wrong with how we'd normally do this?

- Can cause significant *code duplication*
- May require *inaccessible* information
- Not very well *abstracted*
- “Figuring out” which object to instantiate is not a primary concern for most classes
- Can make classes very *rigid* and *fragile*

Advantages of the Factory Method Pattern:

- *Delegates* object creation (and the decision process) to subclasses
- *Abstracts* object creation by using a factory (object production) method
- *Encapsulates* objects by allowing *subclasses* to determine what they need

Factory Method Pattern Analysis

Intent To generalise object creation

Motivation Loading player objects when a game loads

Applicability When sister classes depend on (and create) similar objects

Structure See previous slides

Participants See previous slides

Collaboration Concrete creator objects invoke the factory method in order to produce their desired product

Consequences Object creation in the superclass is now *decoupled* from the specific object required

Implementation See previous slides

Known Uses See previous slides

Design Patterns - Recap

Previous Lecture:

Design Patterns 1: Slides 4-21

- Introduction and Motivation
- Singleton Pattern
- Factory Method Pattern

Today's Lecture:

Design Patterns 2: Slides 22-51

- Template Method Pattern
- Strategy Pattern
- Observer Pattern
- Pattern Types

Template Method and Strategy Patterns

Building generic components that can be *extended*, *adapted* and *re-used* is key to good design.

The two main techniques that are used for developing generic components are *refactoring* and *generalizing*.

- identifying recurring code and replacing with generic code

Inheritance and *delegation* are the two main OO design techniques that are used for building generic components.

Template Method and Strategy Patterns

Template Method and *Strategy* are two design patterns that solve the problem of separating a generic algorithm from a detailed design.

Template Method pattern uses *Inheritance*.

Strategy pattern uses *Delegation*.

Template Method Pattern Motivation

```
public class BubbleSorter {  
    static int operations = 0;  
    public static int sort(int[] array){  
        operations = 0;  
        if (array.length <= 1)  
            return operations;  
        for (int i = array.length - 2; i >=0; i --){  
            for (int j = 0; j <= i; j++){  
                compareAndSwap(array, j);  
            }  
        }  
        return operations;  
    }  
}
```

contd...

Template Method Pattern Motivation

```
public static void compareAndSwap(int[] array, int index){  
    if (array[index] > array[index+1])  
        swap(array,index);  
    operations++;  
}  
  
public static void swap(int[] array, int index){  
    int temp = array[index];  
    array[index] = array[index+1];  
    array[index+1] = temp;  
}  
}
```

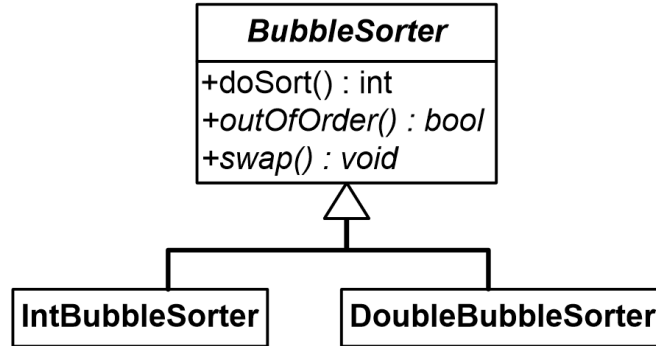
Template Method Pattern Example

write a sorting algorithm
doesn't depend on data

```
public abstract class AbstractBubbleSorter {
    private static int operations = 0;
    protected int length = 0;
    protected int doSort(){
        operations = 0;
        if (length <= 1)
            return operations;
        for (int i = length - 2; i >=0; i--){
            for (int j = 0; j <= i; j++){
                if (outOfOrder(j))
                    swap(j);
                operations++;
            }
        }
        return operations;
    }
    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

Template Method Pattern Example

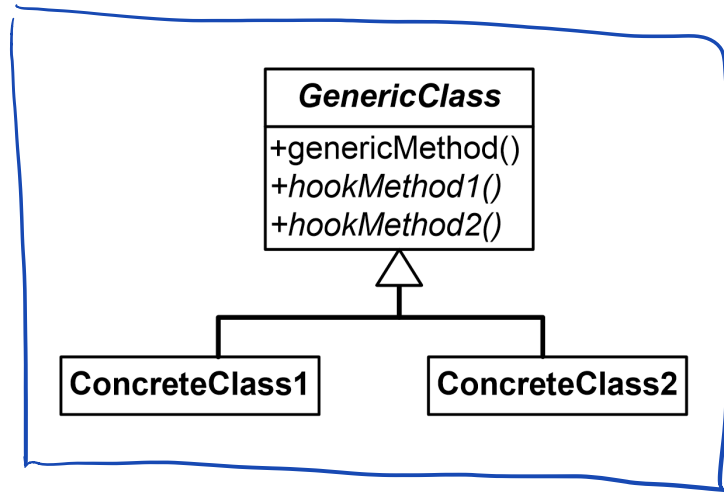
seperate the data from the algorithm



Template Method Pattern Example

```
public class IntBubbleSorter extends AbstractBubbleSorter {
    private int[] array = null;
    public int sort(int[] a){
        array = a;
        length = array.length;
        return doSort();
    }
    @Override
    protected boolean outOfOrder(int index) {
        return (array[index] > array[index+1]);
    }
    @Override
    protected void swap(int index) {
        int temp = array[index];
        array[index] = array[index+1];
        array[index+1] = temp;
    }
}
```

Template Method Pattern Structure



Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.
- Collaboration** See previous slides.
- Consequences** All algorithms must use the same interface.
- Implementation** See previous slides.
- Known Uses** See previous slides.

Strategy Pattern

*generic part in parent
specific part in subclass*
↑

Template method is an example of using inheritance as a mechanism for re-use.

- Generic algorithm is placed in the base class
- Specific implementation is deferred to the sub class

The design tradeoff of using inheritance is the strong dependency to the base class.

useful in any type of sorting algorithm

- Although the methods outOfOrder and swap are generic methods they cannot be re-used because they inherit the AbstractBubbleSorter class

The Strategy pattern is an alternative.

Strategy Pattern Example

```
public class BubbleSorterS {  
  
    static int operations = 0;  
    private int length = 0;  
    private SortHandle itsSortHandle = null;  
  
    public BubbleSorterS(SortHandle handle){  
        itsSortHandle = handle;  
    }  
}
```

→ outOfOrder()
swap() are in the interface

Contd..

Strategy Pattern Example

```
public int sort(Object array){  
    itsSortHandle.setArray(array);  
    length = itsSortHandle.length();  
    operations = 0;  
    if (length <= 1)  
        return operations;  
  
    for(int nextToLast = length - 2; nextToLast >=0;nextToLast--){  
        for (int index=0; index <= nextToLast; index++){  
            if (itsSortHandle.outOfOrder(index))  
                itsSortHandle.swap(index);  
            operations++;  
        }  
    }  
    return operations;  
}
```

→ implement the itsSortHandle interface

Strategy Pattern Example

```
public interface SortHandle {  
    public void swap(int index);  
    public boolean outOfOrder(int index);  
    public int length();  
    public void setArray(Object array);  
}
```

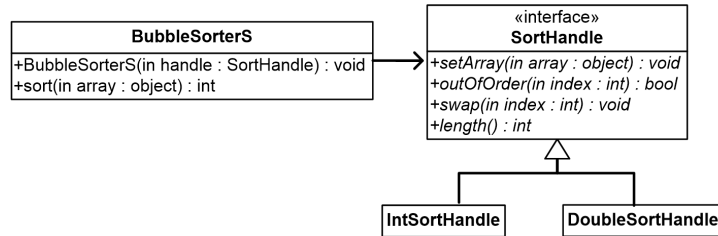
Strategy Pattern Example

```
public class IntSortHandle implements SortHandle {  
    private int[] array = null;  
  
    public int length() {  
        return array.length;  
    }  
  
    public boolean outOfOrder(int index) {  
        return (array[index] > array[index+1]);  
    }  
  
    public void setArray(Object array) {  
        this.array = (int [])array;  
    }  
  
    public void swap(int index) {  
        int temp = array[index];  
        array[index] = array[index + 1];  
        array[index + 1] = temp;  
    }  
}
```

↗ didn't make it a child class
of BubbleSorter

use it in any sorting algorithm

Strategy Pattern Example



Strategy Pattern Example

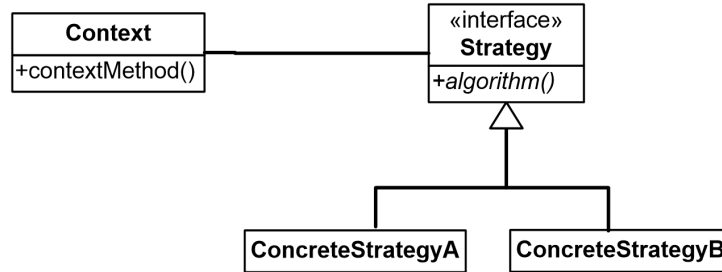
```
public class QuickBubbleSorterS {  
    static int operations = 0;  
    private int length = 0;  
    private SortHandle itsSortHandle = null;  
  
    public QuickBubbleSorterS(SortHandle handle){  
        itsSortHandle = handle;  
    }  
  
    public int sort(Object array){  
        itsSortHandle.setArray(array);  
        length = itsSortHandle.length();  
        operations = 0;  
    }  
}
```

Contd...

Strategy Pattern Analysis

```
    if (length <= 1)
        return operations;
    boolean thisPassInOrder = false;
    for(int nextToLast = length - 2; nextToLast >=0 &&
        !thisPassInOrder; nextToLast--){
        thisPassInOrder = true;
        for (int index=0; index <= nextToLast; index++){
            if (itsSortHandle.outOfOrder(index)){
                itsSortHandle.swap(index);
                thisPassInOrder = false;
            }
            operations++;
        }
    }
    return operations;
}
```

Strategy Pattern Structure



Observer Pattern

There are situations where many objects (observers) depend on the state of one object (subject).

A single object managing a one-to-many dependency between objects is inflexible.

- It commits (tightly couples) the subject to particular dependent objects.
- Such tightly coupled objects are hard to implement, test and maintain.

people "subscribe" to the data
publisher "push the data to you"

Observer pattern decouples the subject and observers using a publish-subscribe style communication pattern

Observer Pattern

Keyword

center object

Subject: An "important" object, whose state (or *change* in state) determines the actions of other classes.

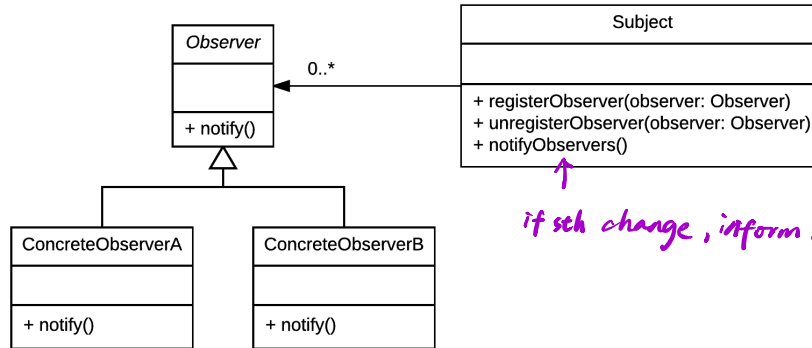
Keyword

Observer: An object that monitors the subject in order to respond to its state, and any changes made to it.

*my opinion
"reactor"*

Observer Pattern Structure

subject keep a list of Observer that are interested in its state



if sth change, inform all observers

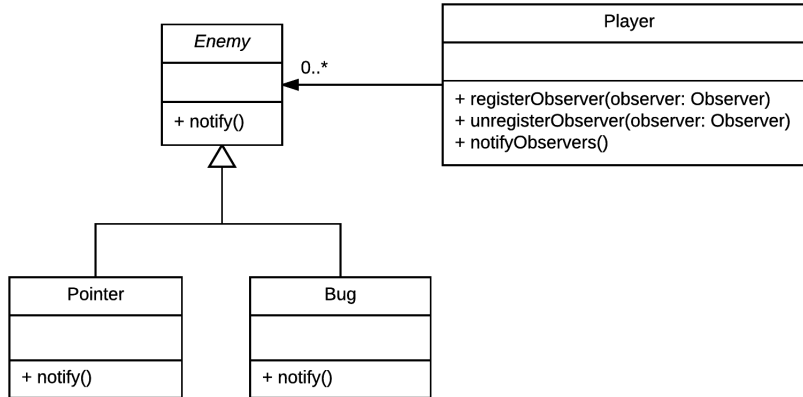
Observer Pattern

event driven programming → *observer respond to "event"*
central object changing

Why do we like this way better?

- Automatically notifies observers of *state changes*
- Decouples the subject and the observer → *minimum the dependency*
- With some extra tools, powers a great deal of *event-driven* programs
- Clear responsibilities: subjects (ideally) know nothing about the observers, except that they exist
- Observers can be added and removed from subjects at will, with zero effect (also known as the publish-subscribe model)
- Can be extended in various ways to improve messaging, decoupling, and event-handling

Observer Pattern Example



Note: this isn't actually a "Java-ready" design

Observer Pattern Example

```
import java.util.Observer;

public abstract class Enemy implements Observer {

    public abstract update(Observable o, Object arg);

}

public class Pointer extends Enemy {

    public update(Observable o, Object arg) {
        ...
    }

}

public class Bug extends Enemy {

    public update(Observable o, Object arg) {
        ...
    }

}
```

Observer Pattern Example

for subject

```
import java.util.Observable;

public class Player extends Observable {
    public Player(..., ArrayList<Observer> observers) {
        ...
        for (Observer o : observers) {
            this.addObserver(o);
        }
        notifyObservers("Player created");
    }
}
```

However, in most cases it makes more sense for Subject (or Observable) to be an interface, so we can write it ourself.

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software, a book written by the *Gang of Four* (or GoF) describing 23 common software design patterns.

All good developers have *seen* most of the patterns, and have an understanding of what they do, and what common problems have already been solved.

Don't reinvent the wheel... You'll learn more about patterns in SWEN30006.

Design Patterns

Commonly solved *classes* of problems:

Creational Solutions related to object creation,
e.g. Singleton, Factory Method

Structural Solutions dealing with the structure of classes and their
relationships,
e.g. Adapter, Bridge

Behavioural Solutions dealing with the interaction among classes,
e.g. Strategy, Template Method, Observer

Review

- ① Identify where the patterns covered in the lecture can be applied to the project
- ② Describe the advantages of using design patterns (in general)
- ③ Describe the core components (the analysis) of the patterns covered in the lecture

Lecture Objectives

You should be able to:

- Describe what **design patterns** are, and why they are useful
- Analyse and understand **design pattern specifications**
- Make use of (some) design patterns