Design Patterns

Asst. Prof. Reginald Neil C. Recario
rcrecario@up.edu.ph
rncrecario@gmail.com
Institute of Computer Science
University of the Philippines Los Baños

Definitions

- What is a pattern?
  - A *pattern* is a **recurring solution** to a standard **problem**, in a **context**.

Patterns in engineering

- How do other engineers find and use patterns?
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Designers (e.g. Automobile) instead, reuse standard designs with successful track records, learning from experience

Patterns in engineering

- Developing software from scratch is also expensive
  - Patterns support **reuse** of software architecture and design

The “gang of four” (GoF)

- **Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides** (Addison-Wesley, 1995)
  - *Design Patterns* book catalogs 23 different patterns as solutions to different classes of problems, in C++ & Smalltalk
  - The problems and solutions are broadly applicable, used by many people over many years

The “gang of four” (GoF)

- GOF presents each pattern in a structured format

Elements of Design Patterns

- Design patterns have 4 essential elements:
  - **Pattern name**: increases vocabulary of designers
  - **Problem**: intent, context, when to apply
  - **Solution**: UML-like structure, abstract code
  - **Consequences**: results and tradeoffs

Three Types of Patterns

- Creational patterns
- Structural patterns
- Behavioral patterns

Three Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects

Three Types of Patterns

- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects

Three Types of Patterns

- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Design Patterns are NOT

- Data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)
- Complex domain-specific designs
  (for an entire application or subsystem)

Design Patterns are NOT

They are:

- “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

Command pattern (Behavioral)

- **Synopsis** or **Intent**: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Command pattern

- **Context**: You want to model the time evolution of a program:
  - What needs to be done, e.g. queued requests, alarms, conditions for action
  - What is being done, e.g. which parts of a composite or distributed action have been completed
  - What has been done, e.g. a log of undoable operations

Command pattern

- Solution: represent units of work as Command objects
  - Interface of a Command object can be a simple execute() method
  - Extra methods can support undo and redo
  - Commands can be persistent and globally accessible, just like normal objects

Command pattern

- *What are some applications that need to support undo?*
  - Editor, calculator, database with transactions
  - Perform an execute at one time, undo at a different time

Command pattern

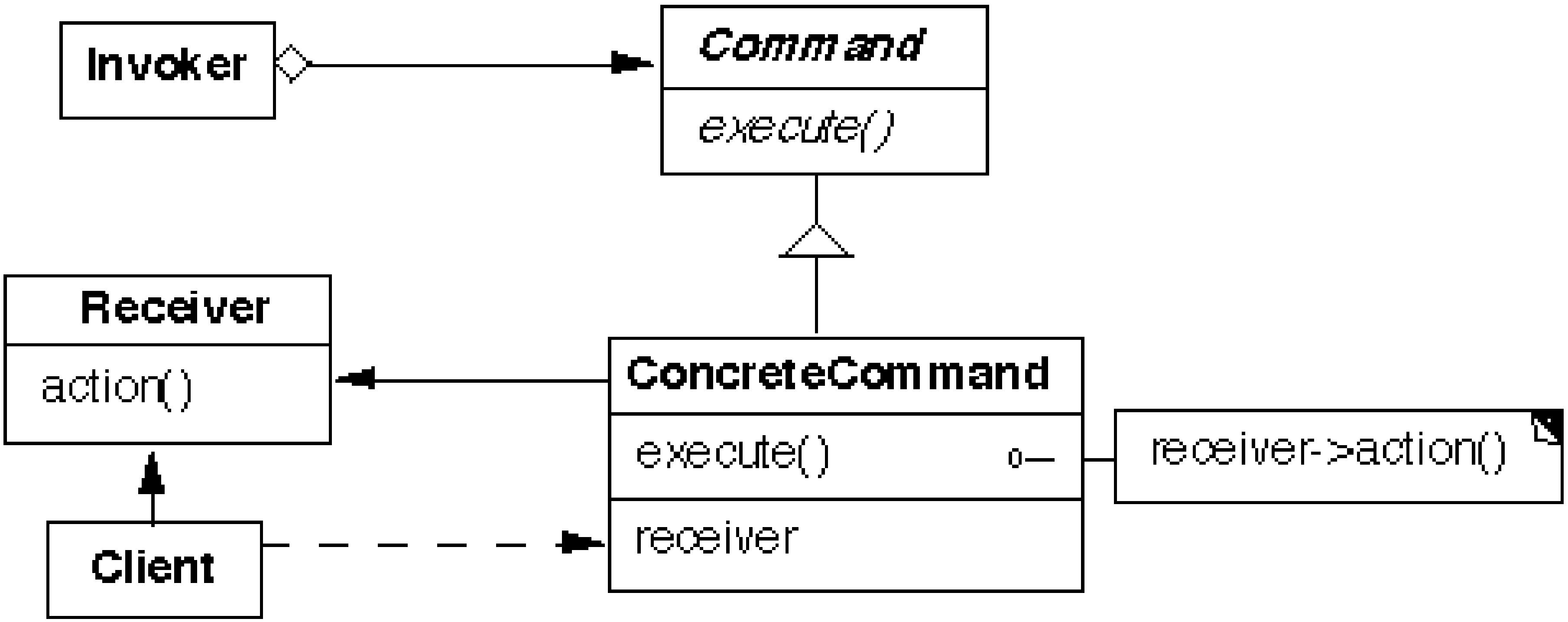
- **Participants** (the classes and/or objects participating in this pattern):
  - **Command (Command)** declares an interface for executing an operation
  - **ConcreteCommand** defines a binding between a Receiver object and an action
    - implements Execute by invoking the corresponding operation(s) on Receiver

Command pattern

- **Participants** (the classes and/or objects participating in this pattern):
  - **Invoker** asks the command to carry out the request
  - **Receiver** knows how to perform operations associated with carrying out the request
  - **Client** creates a ConcreteCommand object and sets its receiver

Command pattern

- **Structure:**



Command pattern

- **Consequences:**
  - You can undo/redo any Command
    - Each Command stores what it needs to restore state
  - You can store Commands in a stack or queue
    - Command processor pattern maintains a history

Command pattern

- **Consequences:**
  - It is easy to add new Commands, because you do not have to change existing classes
    - Command is an abstract class, from which you derive new classes
    - execute(), undo() and redo() are polymorphic functions

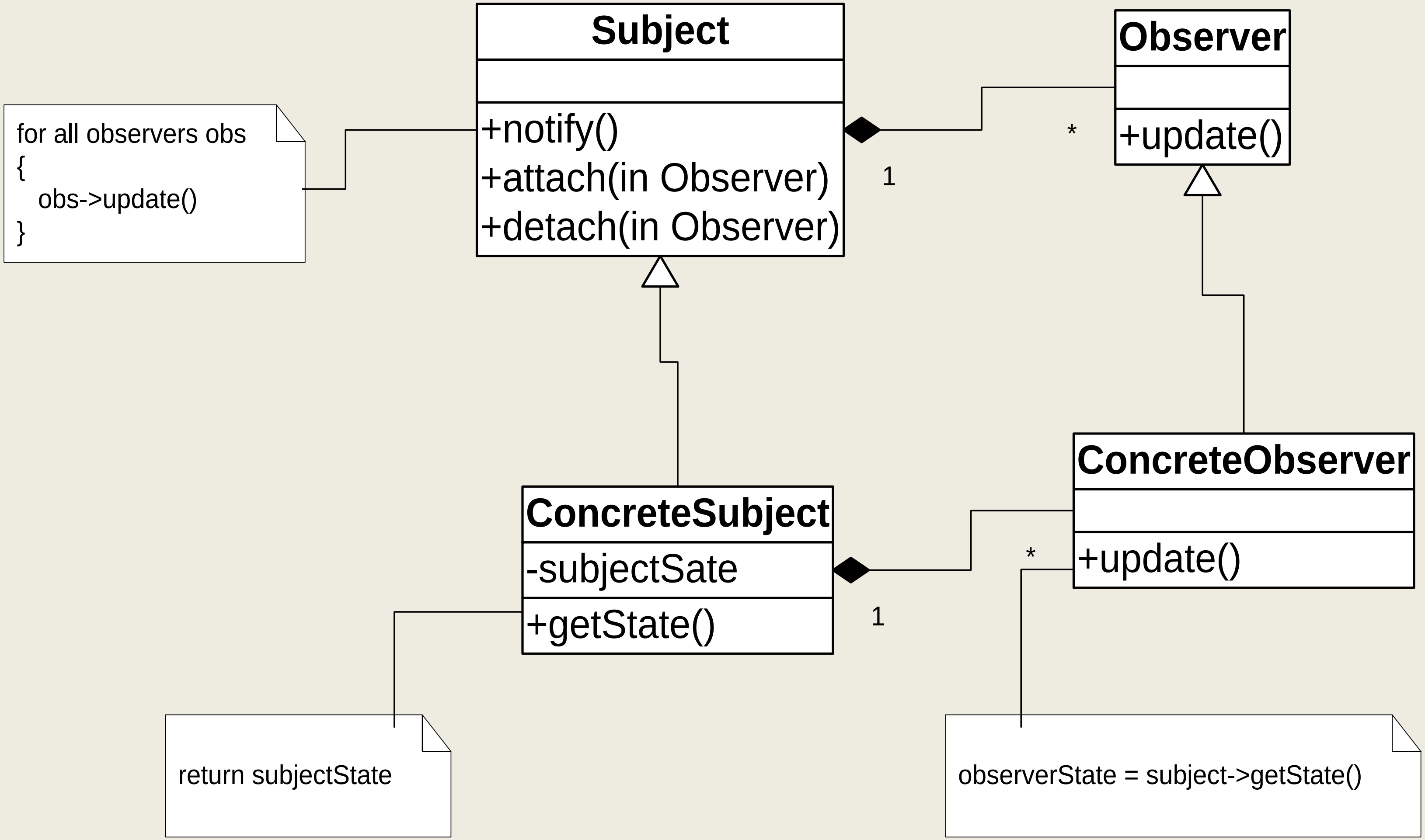
Observer pattern (Behavioral)

- Intent:
  - Define a one-to-many dependency between objects
    so that when one object changes state, all its dependents are notified and updated automatically
- Used in Model-View-Controller framework
  - Model is problem domain
  - View is windowing system
  - Controller is mouse/keyboard control

Observer pattern

- *How can Observer pattern be used in other applications?*
  - JDK's Abstract Window Toolkit (listeners)
  - Java's Thread monitors, notify(), etc.

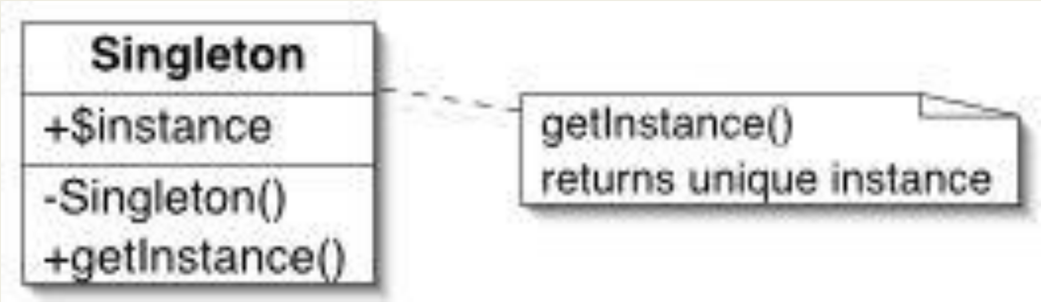
Structure of Observer Pattern



Singleton pattern (Creational)

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

Singleton pattern

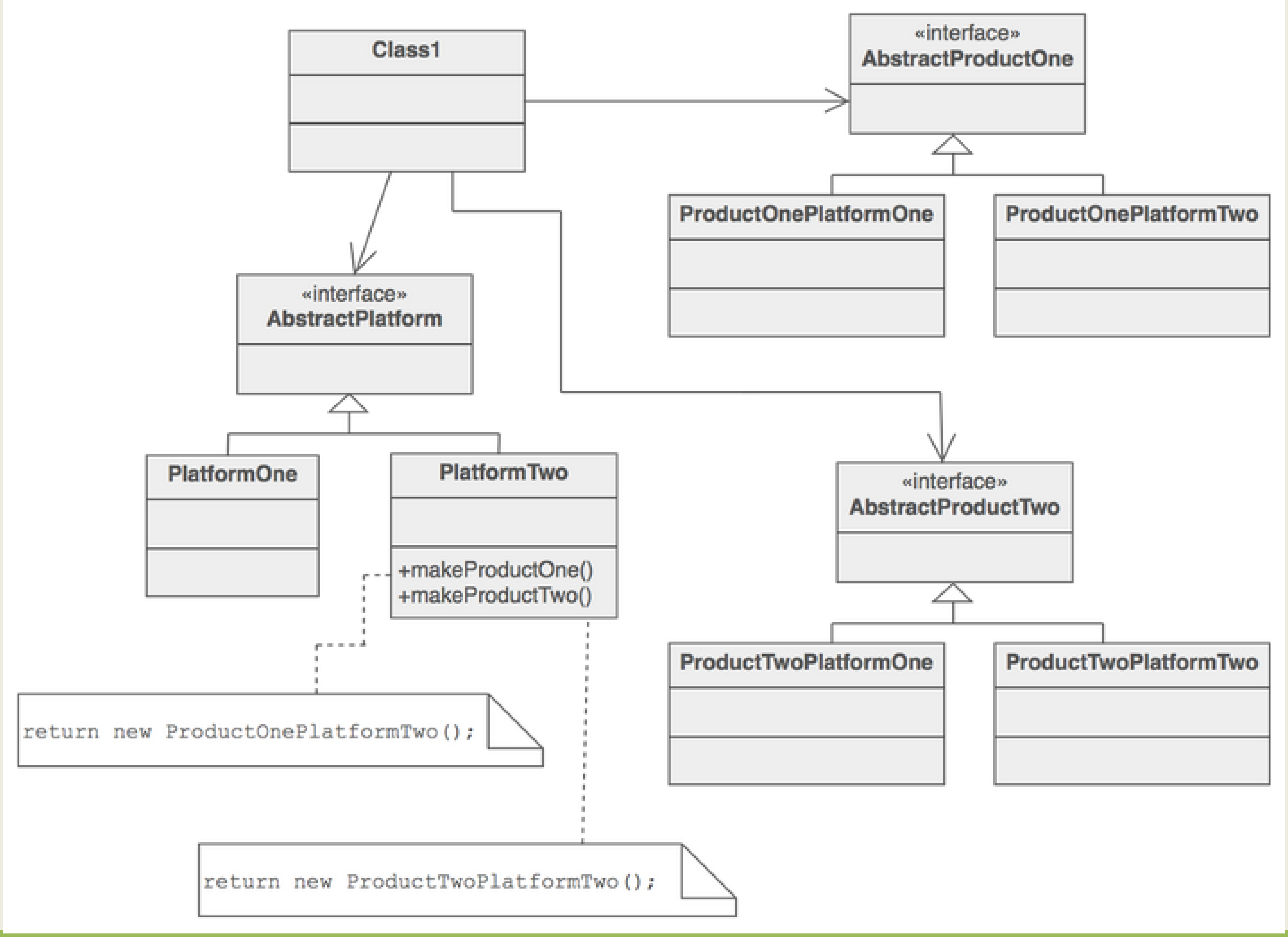



```
class Singleton { public:
      static Singleton* getInstance();
  protected: //Why are the following protected?
      Singleton();
      Singleton(const Singleton&);
      Singleton& operator= (const Singleton&);
 private: static Singleton* instance;
};
Singleton *p2 = p1->getInstance();
```

Abstract Factory (Creational)

- **Synopsis** or **Intent**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".

Abstract Factory

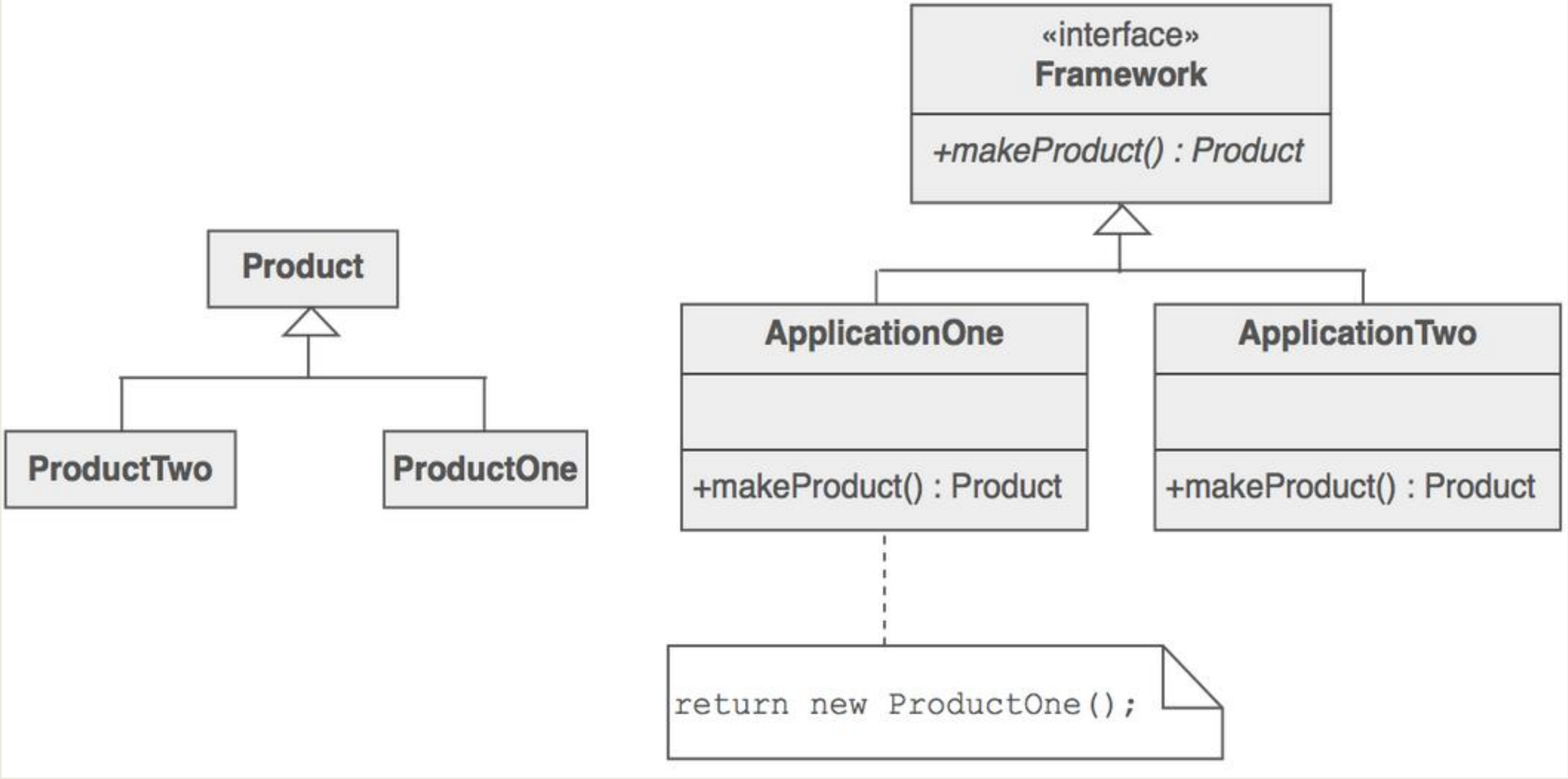


Reference: http://sourcemaking.com/files/v2/content/patterns/Abstract_Factory-2x.png

Factory Method (Creational)

- **Synopsis or Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.

Factory Method



Reference: http://sourcemaking.com/files/v2/content/patterns/Factory_Method-2x.png

Creational Patterns

- **Abstract Factory**:
  - Factory for building related objects
- **Builder**:
  - Factory for building complex objects incrementally
- **Factory Method**:
  - Method in a derived class creates associates
- **Prototype**:
  - Factory for cloning new instances from a prototype
- **Singleton**:
  - Factory for a singular (sole) instance

Structural patterns

- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition

Structural patterns

- Example: Proxy
  - ***Proxy***: acts as convenient surrogate or placeholder for another object.
    - Remote Proxy: local representative for object in a different address space
    - Virtual Proxy: represent large object that should be loaded on demand
    - Protected Proxy: protect access to the original object

Structural Patterns

- **Adapter:**
  - Translator adapts a server interface for a client
- **Bridge**:
  - Abstraction for binding one of many implementations
- **Composite**:
  - Structure for building recursive aggregations
- **Decorator**:
  - Decorator extends an object transparently

Structural Patterns

- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Many fine-grained objects shared efficiently.
- **Proxy**:
  - One object approximates another

Behavioral Patterns

- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Aggregate and access elements sequentially

Behavioral Patterns

- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Behavioral Patterns

- **Observer**:
  - Dependents update automatically when subject changes
- **State**:
  - Object whose behavior depends on its state
- **Strategy**:
  - Abstraction for selecting one of many algorithms

Behavioral Patterns

- **Template Method**:
  - Algorithm with some steps supplied by a derived class
- **Visitor**:
  - Operations applied to elements of a heterogeneous object structure

Patterns in software libraries

- AWT and Swing use Observer pattern
- Iterator pattern in C++ template library & JDK
- Façade pattern used in many student-oriented libraries to simplify more complicated libraries!
- Bridge and other patterns recurs in middleware for distributed computing frameworks
- …

More software patterns

- Design patterns
  - idioms **(low level, C++): Jim Coplein, Scott Meyers**
    - **I.e., when should you define a virtual destructor?**
  - design **(micro-architectures) [Gamma-GoF]**
  - architectural **(systems design): layers, reflection, broker**
    - Reflection makes classes self-aware, their structure and behavior accessible for adaptation and change:
      Meta-level provides self-representation, base level defines the application logic

More software patterns

- *Java Enterprise Design Patterns* (distributed transactions and databases)
  - E.g., ACID Transaction: *A*tomicity (restoring an object after a failed transaction), *C*onsistency, *I*solation, and *D*urability
- **Analysis patterns** (recurring & reusable analysis models, from various domains, i.e., accounting, financial trading, health care)
- **Process patterns** (software process & organization)

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Activity #1

- Form a group of 3 or four and discuss which pattern is applicable in the two situations:
  - #1: A time provider implementation that gives the correct time (say PST) but there should only be one time provider.
  - #2: An implementation of a program that determines whether or not you are running Libre Office applications.

Sample Code: Singleton

```
public class MySun {
    private static MySun instance = null;
    protected MySun() { // Exists only to defeat instantiation. }
    public static MySun getInstance() {
        if(instance == null) { instance = new
            MySun(); }
        return instance;
    }
}
```

Reference: http://www.javaworld.com/article/2073352/core-java/simply-singleton.html

Sample Code: Factory Method

```
interface Dog
{
  public void speak ();
}
```

Reference: http://alvinalexander.com/java/java-factory-pattern-example

Sample Code: Factory Method

```
class Poodle implements Dog{
  public void speak(){ System.out.println("The poodle says
    \"arf\""); }
}

class Rottweiler implements Dog{
  public void speak(){System.out.println("The Rottweiler says (in a
    very deep voice) \"WOOF!\"");}
}

class SiberianHusky implements Dog{
  public void speak(){System.out.println("The husky says \"Dude,
    what's up?\"");}
}
```

Sample Code: Factory Method

```
class DogFactory
{
  public static Dog getDog(String criteria)
  {
    if ( criteria.equals("small") )
      return new Poodle();
    else if ( criteria.equals("big") )
      return new Rottweiler();
    else if ( criteria.equals("working") )
      return new SiberianHusky();

    return null;
  }
}
```

Reference: http://alvinalexander.com/java/java-factory-pattern-example

Sample Code: Factory Method

```
public class JavaFactoryPatternExample{
  public static void main(String[] args){
    Dog dog = DogFactory.getDog("small");
    dog.speak();
    dog = DogFactory.getDog("big");
    dog.speak();
    dog = DogFactory.getDog("working");
    dog.speak();
  }
}
```

Reference: http://alvinalexander.com/java/java-factory-pattern-example

Sample Code: Abstract Factory

```
/*AbstractFactory.java*/
package com.cakes;
public class AbstractFactory {
public SpeciesFactory getSpeciesFactory(String type)
   { if ("mammal".equals(type)) { return new
   MammalFactory(); }
 else { return new ReptileFactory(); } } }
```

Sample Code: Abstract Factory

```
/* SpeciesFactory.java*/
package com.cakes;
import com.cakes.animals.Animal;
 public abstract class SpeciesFactory { public
   abstract Animal getAnimal(String type); }
```

Reference: http://www.avajava.com/tutorials/lessons/abstract-factory-pattern.html

Sample Code: Abstract Factory

```
/* SpeciesFactory.java*/
package com.cakes;
import com.cakes.animals.Animal;
import com.cakes.animals.Cat;
import com.cakes.animals.Dog;
 public class MammalFactory extends
   SpeciesFactory {
@Override public Animal getAnimal(String type) {
    if ("dog".equals(type)) { return new Dog(); }
   else { return new Cat(); } } }
```

Reference: http://www.avajava.com/tutorials/lessons/abstract-factory-pattern.html

Sample Code: Abstract Factory

```
/* SpeciesFactory.java*/
package com.cakes;
import com.cakes.animals.Animal;
import com.cakes.animals.Snake;
import com.cakes.animals. Tyrannosaurus;
 public class MammalFactory extends
   SpeciesFactory {
@Override public Animal getAnimal(String type) {
    if (“snake".equals(type)) { return new Snake(); }
   else { return new Tyrannosaurus(); } } }
```

Activity #2

- Form a group of 3 or four and discuss which pattern is applicable in the two situations:
  - #1: Creating one or more points in a 2 dimensional space.
  - #2: Implementing a program that can perform operations and represent one (x) up to five (x,y,z,a,b) dimensions.
  - #3 Creating a program that counts how many computer is connected to the network

Assignment (to be submitted)

- On a clean sheet of paper, provide one simple example of how ONE of the following (to be decided by the last number of your STUDENT NUMBER) is used:
- **Even number**: Iterator
- **Odd number**: Proxy
- **Submission**: Next meeting

References

- *_____. Blank, Glenn D. Design Patterns. Powerpoint Presentation.*
- *http://home.earthlink.net/~huston2/dp/*
- *http://www.dofactory.com/*
- *http://hillside.net/patterns/*
- *http://sourcemaking.com/design_patterns*
- *Java Enterprise Design Patterns*