Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Design Patterns (DPs) I

I. Salomie, C.Pop
2017

Introduction

OO design goals

- Design for reusability
- Design for flexibility
- Design for generality that allows for (with minimum of effort)
  - Future developments
  - Future updates
  - Extensibility

Granularity in knowledge capturing

- High level – Frameworks
- Medium level – Design Patterns (DP)
- Low level – Classes

Granularity in reuse

- Solution level – Frameworks
- Design level – Design Patterns (DP)
- Code level – Class Inheritance

Definitions and History

- DP - recurring solutions to design problems you see over and over
- DP - set of rules describing how to accomplish certain tasks in the realm of software development (W. Pree, 1994)
- A DP addresses a recurring design problem that arises in specific design situations and presents a solution to it (Buschmann, 1996)
- DPs - identify and specify abstractions that are above the level of single classes and instances, or of components (Gamma, 1993)
- OOPSLA 1991 – Pattern Workshop moderated by Bruce Andersen
- 1993 – first meeting of Hillside Group leaded by Kent Beck and Grady Booch
- 1994 First PLoP (Pattern Language and Program Design) Conference organized by Coplien and Schmidt
- 1995 Design Patterns book published by GoF
- Design Patterns Main References
  - [GoF] Gamma, Helm, Johnson, Vlissides - Design Patterns. Elements of Reusable OO Software, Addison Wesley, 1995
  - [Grand] Mark Grand – Patterns in Java, John Wiley, 1998
- http://hillside.net

Types of Patterns

- Design Patterns must be discovered - pattern mining
- GoF - Gamma, Helm, Johnson, Vlissides (1995)
  - discovered, classified and catalogued a set of 23 patterns
  - Design Patterns. Elements of Reusable OO Software, Addison Wesley, 1995
- Now - hundreds of patterns
- Pattern Spaces
  - A set of domain specific patterns
- The Pattern Space presented by GoF in their book
  - Creational Patterns
    - Help in object creation using indirect creation levels (subclasses or other objects)
  - Structural Patterns
    - Help in composing groups of objects into larger structures
  - Behavioral Patterns
    - Help in defining the communication between objects and also help in flow control of complex programs
- Other Type of Patterns

GoF Patterns
Creational patterns

- Abstract Factory
  - Generation of families of related or dependent objects (product objects) without specifying their concrete class
- Builder
  - Used to generate composite objects. Separates the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory Method (or Virtual Constructor)
  - Defines an interface for creating an object but let subclasses to decide which class to instantiate. Allows a class to defer instantiation process to subclasses.
- Prototype
  - Specify the objects to create using a prototypical instance and create new objects by copying this prototype.
- Singleton
  - Creation of unique objects of a certain class

GoF Patterns
Structural patterns

- Adapter

  Wrapping objects into a new interface. Converts the interface of a class into another interface clients expect. Adapter allows classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge

  Decouples an abstraction from its implementation so that the two can vary independently
- Composite

  Composing and processing of complex objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator

  Attaches additional responsibilities to an object in a dynamic way. Provides a flexible alternative to subclassing for extending functionality.

GoF Patterns
Structural patterns (cont.)

- Façade

  Provide a unified interface to a set of interfaces in a sub-system. Façade defines a higher level interface that makes the subsystem easier to use.
- Flyweight

  Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy

  Provides a surrogate or placeholder for another object to control access to it. Used for accessing remote or expensive objects

GoF Patterns
Behavioral patterns

- Command
  - Encapsulate a request as an object. Uses objects to represent operations.
- Chain of Responsibility
  - Avoids coupling the sender of a request to receiver. Defines a chain of objects that are entitled to know and process the request until an object handles it. The object that will process the request is not known at design time.
- Iterator
  - Encapsulates the way one traverses and accesses the components of an aggregate object
- Interpreter
  - For a given language defines a representation of its grammar and an interpreter for using the grammar to interpret the sentences of the language
- Mediator
  - Encapsulates the protocol between a set of peer cooperating objects (instead of letting peers to define references between them - strong coupling). Mediator object provides the indirection needed for loose coupling

GoF Patterns

Behavioral patterns (cont.)

- Memento
  - Captures and externalizes an object internal state. This way the object can be restored to this state latter on
- Observer
  - Defines a one to many dependency between objects. When one object changes the state, all its dependents are notified and updated automatically
- State
  - Representing and processing states as objects
- Visitor
  - Represent an operation to be performed on the elements of an object structure
- Template Method
  - Defines a skeleton of an algorithm in an operation, postponing some steps to subclasses. The invariant part of the algorithm should be implemented by the template method and the behavior that vary is implemented by subclasses
- Strategy
  - A strategy object encapsulates an algorithm

Pattern Description

GoF proposal

Essential description elements

- Name
  - A good pattern name is essential
- Problem
  - Describes typical pattern application problems
- Solution
  - Shows how the problem could be solved
  - Describes the design pattern elements, their relationships and responsibilities
- Consequences
  - When & how the pattern should be used
  - Best cases scenarios to use the pattern
  - Benefits and cost

Extra elements

- Intention
  - A short explanation of why the pattern is useful
- Applicability
  - Pattern application constraints
- Examples
  - Real problems where the pattern can be used, code excerpts
- Sample code
  - A real implementation of the pattern
- Related Patterns

How to discover patterns

- Reflect on your programming experience
- Identify similarities and situations that happens again and again

How to use patterns

- Much as a personal approach
- Top down using patterns
- Bottom-up using patterns
- Mixed way of using patterns

Adapter (Wrapper) [GoF, Grand] Structural pattern

Adapters

- Allow classes work together that couldn't otherwise due to incompatible interface
- Real world examples

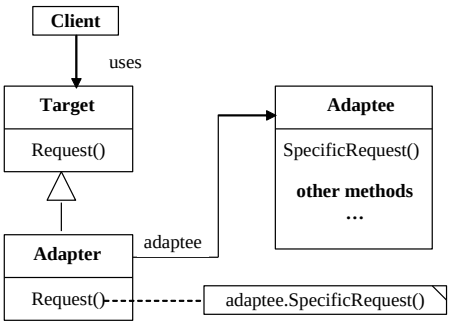
Intention

- Converts interface of a class into an interface that can be used by another class
- You want to use a class but its interface doesn't match the one you need
- You want a client to call a method of an object instance of a class that doesn't implement the interface
- You can arrange for client to make the call through an **adapter class** that either
  - implements the required interface or
  - contains an object instance of class *adaptee*
- Ensures the integration of legacy software into new OO applications

Adapter (Wrapper) [GoF, Grand] Structural pattern

Participants

- Client
  - Class that calls a method of another class through an interface
- Target
  - Defines the domain-specific interface that Client uses (calls)
- Adapter
  - Adapts the interface of *Adaptee* to the *Target* interface
  - Its methods call one ore more methods that class *Adaptee* defines
- Adaptee
  - This class doesn't implement the *Target* methods but has some functionalities you want the *Client* class to call or,
  - Defines an existing interface that needs adapting



Pattern Description
GoF proposal

Consequences
- *Adapter* working with many *Adaptees*
  - eventually with the subclasses of a class *Adaptee*
- The *Adapter* can add functionality to the *Adaptee* classes (and also to all its subclasses)
- What is the complexity of the Adapter?
- What is the adaptation level?
  - it depends of how similar is the Target interface to *Adaptee's*
  - simple interface conversion
    - operation name conversion
  - different set of operations

Examples
- Adapter pattern
  - Provides a new interface to an object Obj1 which helps to adapt to the interface of an object Obj2
  - This allows Obj1 and Obj2 to cooperate
- Java adapter classes
  - Java library defines classes that implement the Adapter design pattern
  - Adapters of Java primitive types
  - Adapters in event handling
    - adapters between objects generating events and objects handling the events
    - Examples (MouseAdapter, WindowAdapter, etc.)

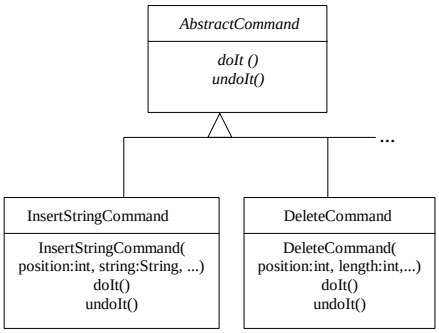
Command [GoF, Grand]
Behavioral pattern

- Intention
  - Uses objects to represent operations (commands)
- Main consequences
  - Commands can be treated and manipulated like objects
  - A sequence of commands could be represented as queues, stacks or using other data structures
    - Allows command undo, repeat iterate, etc.

Command
Behavioral pattern

Motivation [Grand]

- Design of a word processing program
  - features *do* and *undo* commands
- Each command represented as an object with *do* and *undo* methods
- See Figure



How

- Ask word processor to do something
- Create an object, instance of a subclass of AbstractCommand, that corresponds to the command
  - Pass all necessary information to the instance's constructors
- Call *doIt()* to execute the command
- Places the command in a data structure to keep a command history

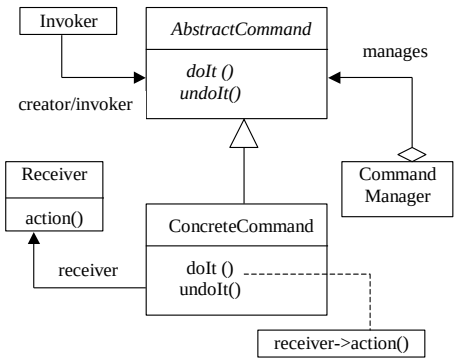
Command
Structure and Participants

AbstractCommand

- The super-class of the classes that encapsulate commands
- Minimally defines a *doIt* method which is overridden (specialized) in the subclasses
- Other methods (undo) can also be defined by the *AbstractCommand*

ConcreteCommand

- Concrete classes that encapsulate a specific command
- Defines the binding between a receiver object and an action
- Implements *doIt* by invoking the corresponding operation(s) on *Receiver*
- Other classes invoke the command through a call to class *doIt* method
- The object's constructor normally supplies all necessary parameters to the command



Command
Structure and Participants

Invoker

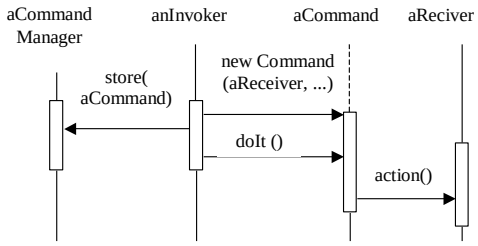
- A class in this role creates concrete command objects if it needs to invoke a command
- It may call those objects' *doIt* method or leave this responsibility to the *CommandManager*

Receiver

- Knows how to perform the operation
- For example, it can be a document, an application, etc.

ComandManager

- Responsible for managing a collection of *Command* objects created by the *Invoker* object(s)
- Can be application independent (reusable)



Participants collaboration

Command
Consequences

- Decouples the object that invokes the operation from the one that knows how to perform it
  - The invoker and executer are different objects
  - This separation allows for flexibility in timing and sequencing the commands
    - Commands can be manipulated like any other objects
- Commands are first-class objects
- Commands - assembled into composite commands
- New commands can be easily added without changing the existing ones

Composite [GoF, Grand]

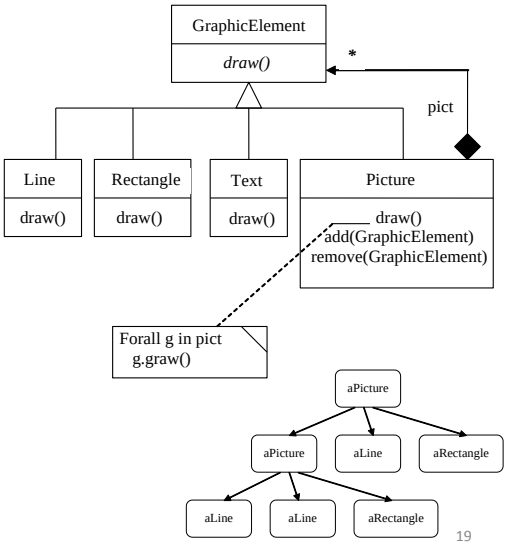
Structural pattern

Intention

- Represents part-whole hierarchies as a result of object composition
- Also known as recursive composition pattern

Motivation

- Atomic elements and containers
- Treating atomic elements and containers
  - in the same way
  - in a different way
- Composite pattern main feature
  - abstract class that represents both primitives and their containers



Composite

Structure and Participants

Component

- Declares the interface for the objects in Composition

Leaf

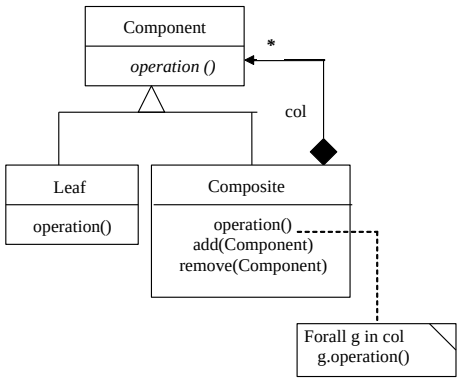
- Represents leaf (atomic) objects in the composition
- Defines the behavior for the primitive objects in the composition

Composite

- Defines behavior for components having children
- Stores child components

Client

- Manipulates objects in the composition through the Component interface



Composite
Consequences

- Defines class hierarchies consisting of primitive objects and composite objects
  - Primitive objects can be composed into more complex objects which in turn can be composed, and so on recursively
  - Wherever client code expects a primitive object, it can also take a composite object
- Make the clients simple
  - Clients can treat composite structures and individual objects uniformly
  - Clients normally don't know (and shouldn't care) whether they are dealing with a leaf or a composite component – this simplifies client's code
  - Makes it easier to add new kinds of components. Newly and defined Composites or leafs subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes

Factory Method [GoF, Grand]
Creational Pattern

Intention

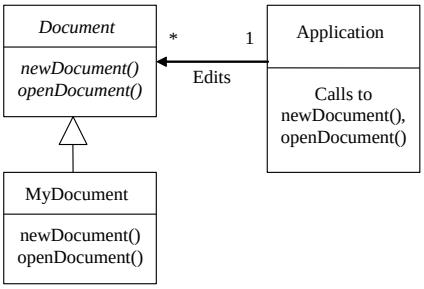
- Class C should instantiate other classes without being dependent on them
- Class C remains independent of the classes it instantiates
  - It delegating the choice of which class to instantiate to another object
  - The newly created object is referred through a common interface

Objective

- To allow the *Application* class to call the programmer provided classes without having any dependencies on it

How

- The framework provides an abstract class (interface) that the programmer-provided class would have to subclass (implement)



A document processing system

-The objective is to decouple the Application of the documents
-The Application should not be hardwired to the documents

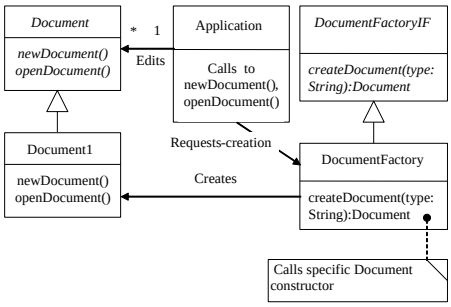
Factory Method
Creational Pattern

Details

- An *Application* object calls *createDocument* method of an object instance of a subclass of *DocumentFactoryIF*
- It passes a *String* to the *createDocument* method that tells that method which subclass of the *Document* class to instantiate
- *Application* class does not need to know the actual class of the object whose method it calls or which subclass of the *Document* class it instantiates

Conclusion

- The **Factory method** pattern provides an application-independent object with an application-specific object to which it can delegate the creation of other application-specific objects



Application framework with Document factory

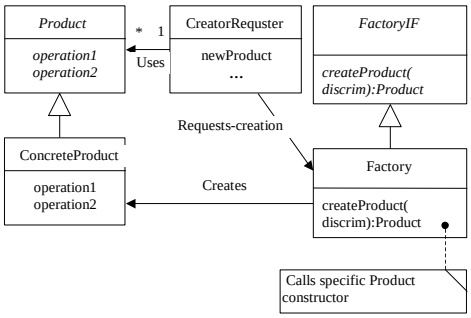
Factory Method
Creational Pattern – The solution and participants

Product

- The abstract super-class of objects produced by the **Factory Method** pattern
- An actual class in this role is usually not called *Product*, but has names like *Document*, *Image*, etc.

Concrete Product

- Any concrete class instantiated by the objects that participate in the **Factory Method** pattern
- An actual class in this role is usually not called *ConcreteProduct*, but has a name like *RTFDocument* or *JPEGImage*



Factory Method

Creational Pattern – The solution and participants

FactoryIF

- Application independent abstract class
- Classes of the objects that create *Product* objects on behalf of *CreateRequester* are subclasses of this abstract class
- Declares a method that is called by *CreateRequester* object to create concrete product objects.
  - The method typically has a name like *createDocument* or *createImage*
- Method arguments
  - Used to infer the class to instantiate
- Typically names of FactoryIF classes *DocumentFactoryIF* or *ImageFactoryIF*

CreationRequester

- Application-independent class
- Needs to create application specific objects. It does so indirectly through an instance of a factory class

Factory class

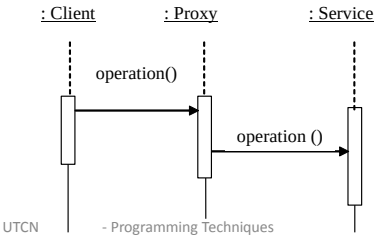
- Application specific class
- Subclass of the abstract class and has a method to create *ConcreteProduct* objects
- Classes that typically fill this role have names like *DocumentFactory* or *ImageFactory*

Proxy [GoF, Grand]

Structural Pattern

Intention

- Creation of surrogate objects that replace other objects
  - Usually expensive or that need not an immediate instantiation
- Proxy clients are not aware they are dealing with a surrogate instead of the real objects they need



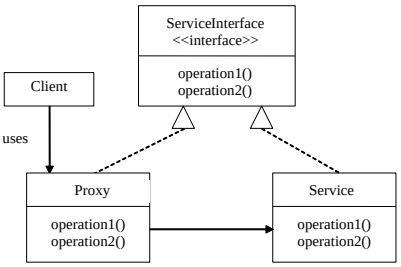
Introduction

- Proxy objects are representing real objects
- Proxy objects define the same operations as real objects
  - Proxy class and the class they represent (class Service) have the same interface or the same superclass
- Proxy operations implement the management aspect of real operation
  - If necessary, delegates the operation execution to the real object

Proxy
Structural Pattern

Proxy types

- Representative of an expensive operation
  - Takes a long execution time
  - Proxy returns a quick partial answer
- Remote proxy
  - Representative of an operation implemented on a remote machine
  - Used in Java RMI, CORBA, RPC
- Access proxy
  - Access control to shared resources
- Virtual Proxy
  - Representative of an expensive object
  - The expensive object is created only when necessary
  - Postpones the instantiation of expensive objects until it is required (lazy instantiation)



Note. A *abstract* class can be used instead of the Interface.
In this case, the classes Proxy and Service are subclasses of the abstract class

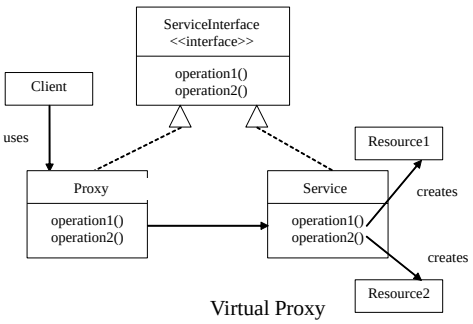
Proxy
Structural Pattern – Virtual Proxy and Participants

Service

- Provides the full service
- For Virtual Proxy, it is responsible to create its own resources (Resource1, Resource2, …)

Client

- Uses the services provided by class Service
- It cannot directly access Service operations and resources, but by means of Proxy class



Virtual Proxy

Proxy
Structural Pattern

Participants (cont.)

Proxy

- An indirect resource between client and service
- Postpones the instantiation of Service objects until they are needed (Virtual Proxy)
- By this indirection, the Client object is not aware that the Service object could not be created when the operation is invoked and processed by the proxy object (Virtual Proxy)

Consequences

- The clients are not aware of Proxy arrangements
- Classes that are accessed through proxy are dynamically loaded
  - Only when necessary (virtual proxy)
  - Their instances are created only when they are needed

Singleton [GoF, Grand]

- Intention
  - The creation of only one instance object of a class (the singleton class)
  - Examples
  - Main requirements
    - Ensure only one instance object for a class
    - Provide a uniform, global access point to the singleton object
  - The class will be responsible for the management of its sole instance
- Singleton class hides the one only instance creation operation behind a static member method
  - Traditionally named instance()
  - Method instance() returns a reference (or pointer) to the sole instance
- Implementation issue
  - The implementation requires static data
  - Programming languages that have no static data cannot guarantee (within the class itself) the sole instance object

Singleton

Implementation using static methods in Java

```
public final class Singleton {
  private static Singleton oneObject;
  private Singleton() { }
  public static Singleton instance() {
    if(oneObject == null)
      oneObject = new Singleton();
    return oneObject;
  }
}
…
Singleton s1 = Singleton.instance();
Singleton s2 = Singleton.instance();
// s1 and s2 are the same object
// (same reference)
```

- If the oneObject already exist
  - The programmer always obtains the same reference
- No exception handling necessary
- What happens if

Singleton s = new Singleton();

- Answer: …
- Alternative
  - instance() returns null if the singleton object is already created
  - an exception is thrown
- How about creating only two or three instances of the same class?

Singleton

Implementation using exceptions in Java

```
// class SingletonException
public class SingletonException extends
      RuntimeException {
  public SingletonException() { super(); }
  public SingletonException(String str) { super(str);}
}
// class Singleton
public class Singleton {
  static boolean oneInstance = false;
  // … other instance variables
  public Singleton(… ) throws  SingletonException {
      if(oneInstance == true) throw
                SingletonException("singleton is created");
       else {oneInstance = true;}
      // … other code
    }
  }
}
```

```
public class Test {
  static public void main (String argv[]) {
     Singleton s1, s2;
     try {s1 = new Singleton(…);} // ok
     catch (SingletonException e) {
        System.err.println(e.getMessage());
     }
     try {s2 = new Singleton(…);} // ex
     catch (SingletonException e){
        System.err.println(e.getMessage());
     }
   }
}
```