Object –Oriented Software Development Using Java

Chap 7. Design by Abstraction

한동대학교 전자전산학부
김 기석 교수

- peterkim@handong.edu
- http://www.handong.edu

Chapter Overview

- overview
  - design patterns
  - designing reusable and flexible(extensible) components using abstract classes, interfaces, and design patterns
  - several design patterns for abstraction – the Template Methods, Strategy, Factory, Iterator
  - ~~Case Study : the animation of sorting algorithms~~

Chapter Overview

- Reusability
  - 1) reuse of implementation through inheritance
    - relative easy
  - 2) reuse of system designs and architecture
    - relative difficult
    - Ex) design patterns(chap 7)
    - Ex) frameworks (chap 8)

7.1 Design Patterns

- The Concept of patterns
  - originally articulated by Christopher Alexander to describe architectural designs
  - In Searching for the essence of great buildings, great towns, and beautiful places
  - Alexander , “The Timeless Way of Building and A Pattern Language – Towns, Buildings, Construction”
  - The Timeless Way of Building which “is thousands of years old, and the same today as it has always been”
  - 253 patterns
  - Each pattern describes a problem which occurs over and over
  - and describes the core of the solution to that problem ….

7.1 Design Patterns

- Each pattern represents a solution to a problem
- Similarity between architectural designs & Software Designs – it is natural to adapt the concept of architectural patterns to software design.

- Both are creative processes that unfold within a wide range (i.e., all possible designs).
- The resulting design must satisfy the customer's needs.
- The resulting design must be feasible to engineer.
- The designers must balance many competing constraints and requirements.
- The designers must seek certain intrinsic yet unquantifiable qualities, such as elegance and extensibility.

7.1 Design Patterns

- Software design patterns
  - Schematic descriptions of solutions to recurring problems in software design

- The Main purposes of using software design patterns
  - (a) to capture and document acquired in software design in a relatively small number of design patterns
  - (b) to support and boost in software systems that use established design patterns proven effective
  - (c) to provide a or software designers to communicate about software design.

- Gamma's Design Patterns [1995] (the pioneering work)
  - 23 design patterns

- Category of design patterns

1. [redacted] which deal with the process of object creation;
2. [redacted] which deal primarily with the static composition and structure of classes and objects; and
3. [redacted], which deal primarily with dynamic interaction among classes and objects.

7.1 Design Patterns

- The description of each design pattern
  - Pattern Name : The essence of the pattern
  - Category : Creational, structural, behavioral
  - Intent : A short description of addressed
  - Also known as : other well known names
  - Applicability : situations in which the pattern can be applied
  - Structure : that depicts the participants of the pattern and the relationships among them.
  - Participants : A list of classes and/or objects participating in the pattern

7.1.1 Design Pattern : Singleton

- Singleton : Example of Design Pattern

Design Pattern ***Singleton***

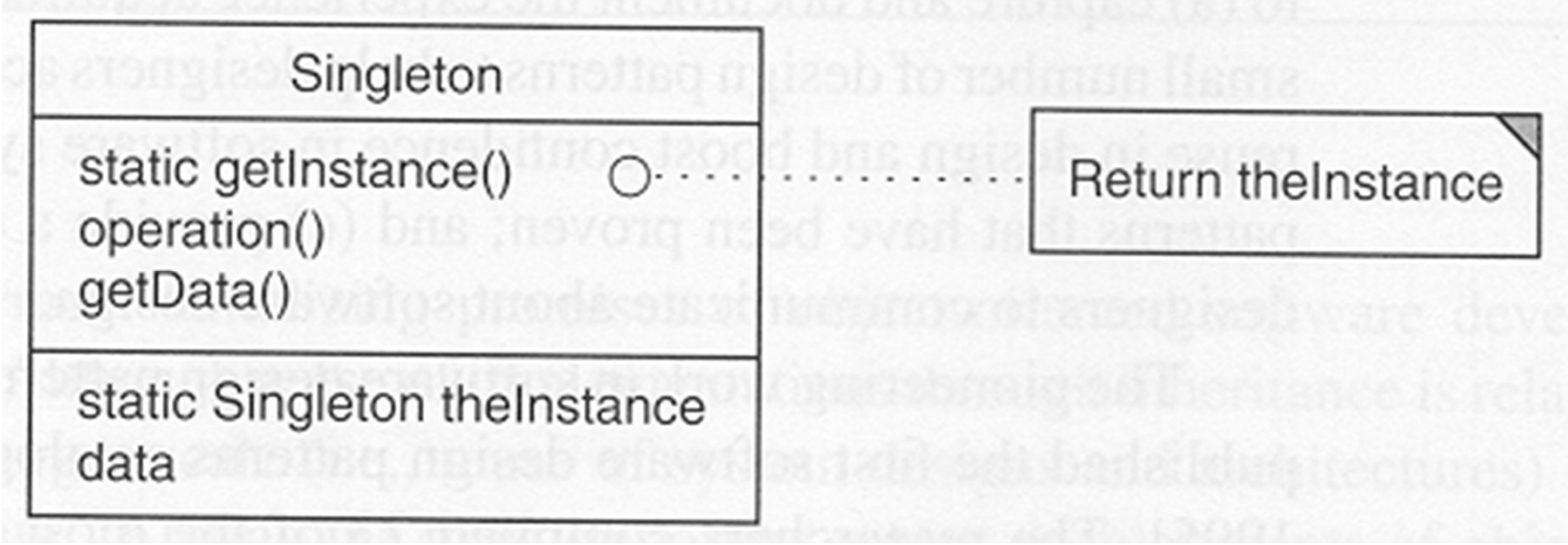
Category: design pattern.

Intent: Ensure that a class has and provide a global point of access to it.

Applicability: Use the Singleton pattern when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.

7.1.1 Design Pattern : Singleton

- The Structure of the singleton pattern



- Only one participant
  - Singleton : declares the unique instance of the class as a static variable, defines a static method getInstance() for clients to access the unique instance .

An implementation of the Singleton pattern

```
// p. 252.

Public class Singleton {
        public static Singleton getInstance() {
                return theInstance;
        }
        private Singleton() {
                //(initialize instance fields
        }
        //(other fields and methods
        private static Singleton theInstance = new Singleton();
}
```

- Idioms
  - patterns concerning techniques in a specific programming language

7.2 Designing Generic Components

- [redacted]c components(== reusable components)
  - in the form of classes or packages
  - can be extended, adapted, and reused in many different contexts without having to modify the source code
  - two basic technique of designing Generic Components
    - 1) [redacted] 2) [redacted]
  - two mechanism used to build generic components
    - 1) [redacted] 2) [redacted]
  - Abstract classes and interfaces play important roles in implementing generic components

7.2.1 Refactoring

- By identifying recurring code segments that are identical or nearly identical

  ex) start(), stop(), and run() in every animation applet
- Refactoring consists of the following tasks

- Identifying code segments in a program that implement the [redacted] often in the same exact code, in many different places.
- Capturing this logic in a [redacted] that is defined once.
- [redacted] the program so that every occurrence of the code segment is replaced with a reference to the generic component.

7.2.1 Refactoring

- Benefits of Refactoring
  - eliminates code duplication and the dangers for maintenance.
  - a bug fix or logic enhancement need be implemented only once.

Design Guideline *Refactoring Recurring Code Segments*

Recurring code segments based on the same logic are hazardous to maintenance. They should be refactored so that the code segment occurs only once. Other occurrences of the code segment should be replaced with references to the common code.

- Refactoring
  - 1) Refactoring by
  - 2) Refactoring by
  - 3) Refactoring by

7.2.1 Refactoring

- 1) Refactoring by Method invocation
  - The Simplest form of refactoring
  - Use of function or method invocation
  - ex) the highlighted code segment occurs in two different contexts
    - computeAll() that contains the recurring code segment.

```
class Computation {
    void method1(. . . ) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }
    void method2(. . . ) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }
    //...
}
```

∈

```
class RefactoredComputation {
    void computeAll() {
        computeStep1();
        computeStep2();
        computeStep3();
    }
    void method1(. . . ) {
        //...
        computeAll();
        //...
    }
    void method2(. . . ) {
        //...
        computeAll();
        //...
    }
    //...
}
```

7.2.1 Refactoring

- Refactoring by method invocation
  - effective only when
    - 1) each occurrence of the recurring code segment is contained within a single method
    - 2) all the methods that contain the recurring code segment belong to the same class.
- Refactoring by inheritance or delegation
  - 1) when the recurring code segment involves several methods
  - 2) recurring code segment occurs in several classes

7.2.1 Refactoring

- 2) Refactoring by Inheritance
  - recurring code segments in different classes
  - A Common superclass of ComputationA and ComputationB is introduced .

```
Class ComputationA {
        void method1(…) {
          //….
          computeStep1();
          computeStep2();
          computeStep3();
          //….
        }
        ////
}
```

```
Class ComputationB {
        void method2(…) {
          //….
          computeStep1();
          computeStep2();
          computeStep3();
          //….
        }
        ////
}
```

Refactoring by inheritence

```
Class Common {
        void computeAll(…) {
          computeStep1();
          computeStep2();
          computeStep3();
        }
}
```

```
Class ComputationA
        extends Common  {
          void method1(…) {
            //….
            computeAll();
            //…..
          }
          //…..
}
```

```
Class ComputationB
        extends Common  {
          void method2(…) {
            //….
            computeAll();
            //…..
          }
          //…..
}
```

7.2.1 Refactoring

- 3) Refactoring by Delegation
  - refactoring of recurring code segments in different classes (not parent class)
  - Introduce a helper class and place the refactored code sequence in the computeAll() method of the helper class.
  - Both ComputationA, ComputatonB need to Contain a reference to the helper class.

Refactoring by delegation

```
Class Helper {
        void computeAll(…) {
          computeStep1();
          computeStep2();
          computeStep3();
        }
}
```

```
Class ComputationA  {
        void compute(…) {
          //….
          helper.computeAll()
          //…..
        }
        //…..
        Helper helper;
}
```

```
Class ComputationB  {
        void compute(…) {
          //….
          helper.computeAll();
          //…..
        }
        //…..
        Helper helper;
}
```

7.2.1 Refactoring

- Refactoring by inheritance and by delegation
  - By inheritance
    - simpler than by delegation
    - 그러나 [redacted] 특성으로 말미암아 불가능한 경우가 존재..
    - 즉 computationA와 computationB 둘중 하나라도 Object가 아닌 클래스의 subclass이어야 한다면.. 불가능
  - By delegation
    - more flexible than by inheritance
    - Since a class that wants access to the refactored method does not need to be related via inheritance to the class of the method.
    - refactoring can always be achieved through d[redacted]

끝