Review of Design Pattern Concept

the Use of the same Coding Framework (Blueprint) in similar applications (situations)

Communication among Developers

Improved Understandability

Efficient Code Structure

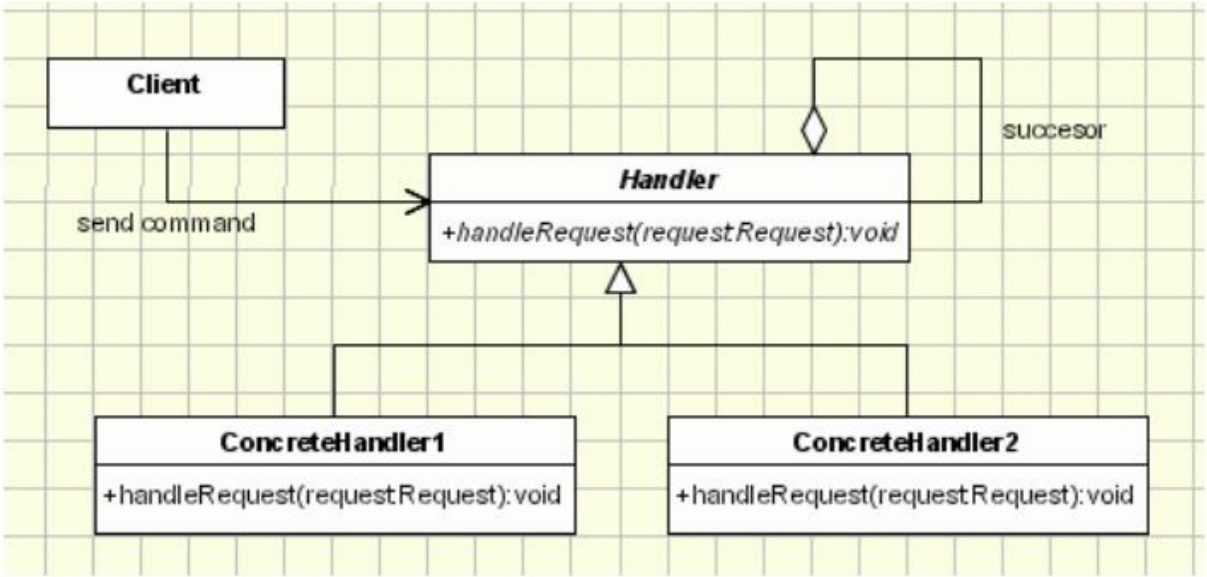
Mainly from Jia Book

Review of Lecture 1
Chain of Responsibility Pattern

Goals:

1. to avoid coupling the sender of a request to its receiver
2. to give more than one object a chance to handle the request.
3. to isolate the clients from knowledge of how requests are assigned.

Loose Connection between Requester and Server



What are two main important benefits with using chain of responsibility pattern?

```
public class Main {
    public static void main(String[] args) {
        Support alice     = new NoSupport("Alice");
        Support bob      = new LimitSupport("Bob", 100);
        Support charlie   = new SpecialSupport("Charlie", 429);
        Support diana     = new LimitSupport("Diana", 200);
        Support elmo     = new OddSupport("Elmo");
        Support fred      = new LimitSupport("Fred", 300);

        alice.setNext(bob).setNext(charlie).setNext(diana).
           setNext(elmo).setNext(fred);

        for (int i = 0; i < 500; i += 33) {
            alice.support(new Trouble(i));
        }
    }
}
```

Refactoring by Inheritance and Its Problem

Template Method Concept

(from page 260 of Jia Book)

Generic Components
(**Reusable** Component)

Program component that can be ***extended***, ***adapted***, and ***reused*** in many different contexts **without** having to modify the source code (or a **minor change)**

- **Refactoring**

  Changing of **recurring code** segments into a **generic component**

- **Generalizing**

  Restructuring of a specific solution to a category of **similar problems**

Refactoring

- **Refactoring Process**

①identify **duplicate code segments** in many different places

②capturing **duplicate logic into a generic component**

③every occurrence of the recurring code segment is replaced with a **reference to the generic component**

- **Effect of Refactoring**

①enhancing **maintainability**

②eliminating (or localizing) the cause of bug

Refactoring by Inheritance

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

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

```
class Common {
 void computeAll(…) {
  computeStep1();
  computeStep2();
computeStep3();
} }
class ComputationB
extends Common { //…
void method2(…) { //…
computeAll(..); //…
 } //… }
```

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

Refactoring by Delegation

```
class Helper {
    void computeAll(…) {
      computeStep1();
      computeStep1();
      computeStep1();
      }
    }

    Helper helper;
```

```
class ComputationA {
 void compute (…) { …
helper.computeAll(); …
  } …
}
class ComputationB {
 void compute (…) { …
helper.computeAll(); …
  } …
}
```

Problems with Refactoring by Inheritance

```
class Common {
 void commonCode1() {
  <common code segment 1>
   }
 void commonCode2() {
<common code segment 2>
   }
}
```

<u>What if common parts are related closely?</u>

Solution with Abstract Class

```
abstract class Common {
 void method(…) {
//template method
  <common code segment 1>
  contextSpecificCode();//a hook
  <common code segment2>
 }
abstract void contextSpecificCode();
}
```

How to make reusable component? Use abstract method!

Abstract Method with abstract keyword

No implementation for an abstract method

Solution with Abstract Class

```
abstract class Common {
 void method(…) {
//template method
  <common code segment 1>
  contextSpecificCode();
  <common code segment2>
 }
abstract void
      contextSpecificCode();
}
```

```
class ContextB extends Common {
 void contextSpecificCode() {
 <context-specific code B>
}..}
```

```
class ContextA extends Common {
 void contextSpecificCode() {
 <context-specific code A>
}..}
```

Problems with Refactoring by Inheritance

class ContextA {
void method(…) {
<common code segment 1>
<context-specific code A>
<common code segment 2>
	class ContextB {
	void method(…) {
	<common code segment 1>
	<context-specific code B>
	<common code segment 2>

class Common {
void **commonCode1()** {
<common code segment 1>
}
void **commonCode2()** {
<common code segment 2>
}
}
	class ContextA (or ContextB)
	extends Common {
	void method(…) {
	commonCode1()
	<context-specific **code A or B**>
	commonCode2()
	} //
	}

What problem if two separate common code segments are <u>closely related?</u>

```
abstract class Common {
 void method(…) {
   //template method
  <common code segment 1>
  contextSpecificCode(); // a hook
  <common code segment2>
 }
abstract void
 contextSpecificCode();
}
```

frozen part
(fixed behavior)

Changing part
(changeable behavior)

Solution with Abstract Class

```
abstract class Common {
 void method(…) {
//template method
  <common code segment 1>
  contextSpecificCode();//a hook
  <common code segment2>
 }
abstract void
  contextSpecificCode();
}
```

refactor
the context-sensitive code
as a placeholder,
which is intended to be
overridden and customized
in each subclass

```
class ContextA extends Common {
 void contextSpecificCode() {
 <context-specific code A>
}..}
```

```
class ContextB extends Common {
 void contextSpecificCode() {
 <context-specific code B>
}..}
```