Design Pattern Concept

with the example of Chain of Responsibility

OODP, Spring, 2022

Creational pattern

Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.

Builder pattern, which separates the construction of a complex object from its representation so that the **same construction process** can create different representation.

Factory method pattern, which allows a class to **defer instantiation to subclasses.**

Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by **cloning** this prototype.

Singleton pattern, which ensures that a class only has **one instance**, and provides a global point of access to it.

Behavioral Pattern

Chain of responsibility pattern: Services for client are to be passed on **a chain of server objects**…

Command pattern: invoker, receiver, and command example…

Iterator pattern: Iterators are used to access the **elements of an aggregate object** sequentially without exposing its underlying representation

Memento pattern: Provides the ability to restore an object to its previous state (rollback)

Mediator pattern: Provides a **unified interface** to a set of interfaces in a subsystem

Observer pattern: **distributed event handling**…

Strategy pattern: **Algorithms can be selected on the fly**

Visitor pattern: A way to **separate an algorithm from an object**

structural design patterns

Adapter pattern: allows you to make an existing class work with other existing class libraries **without changing the code of the existing class**.

Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently
Example: DrawingAPI.java; Shape.java

Composite pattern: a tree structure of objects where every object has the same interface

Decorator pattern: add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes

Facade pattern: create a simplified interface of an existing interface **to ease usage for common tasks**

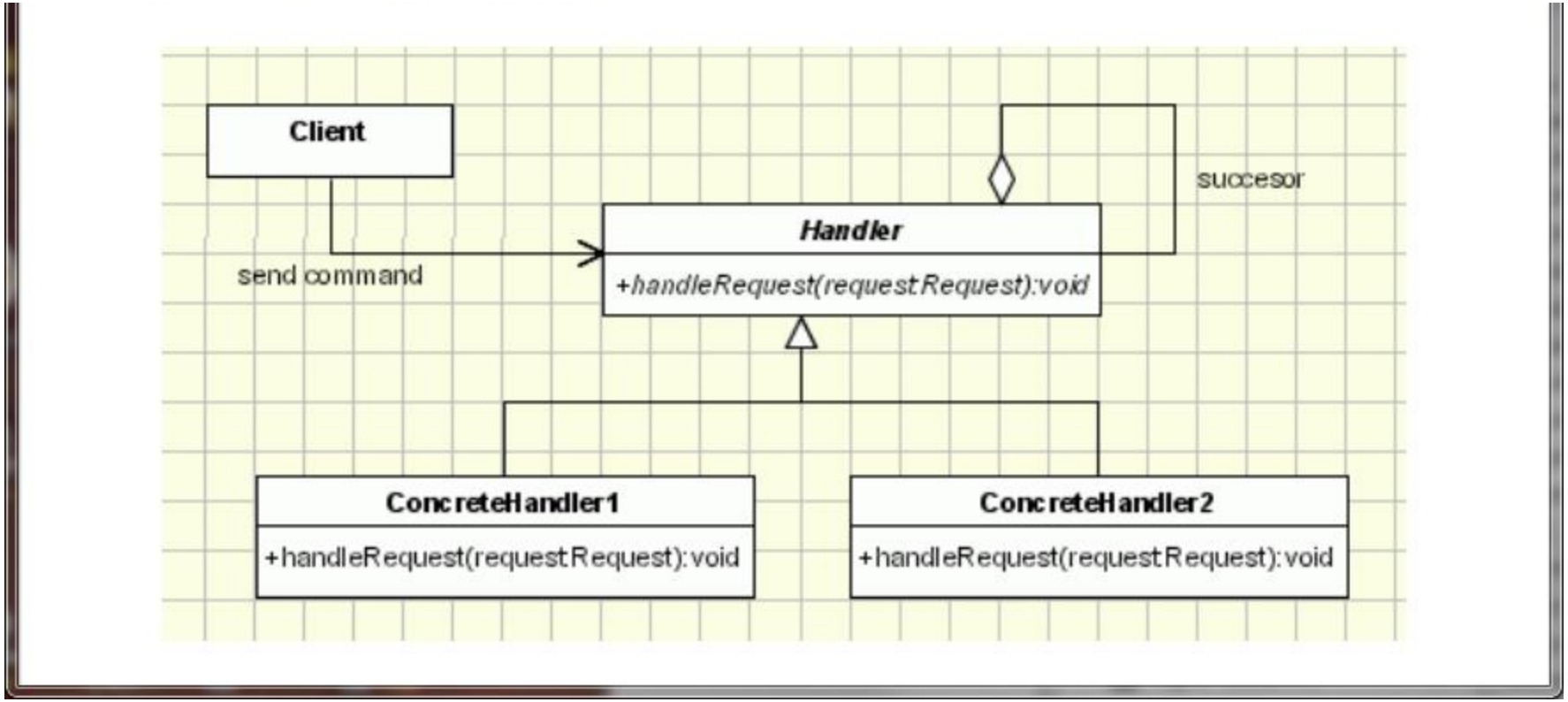
Flyweight pattern: a high quantity of objects share a common properties object to **save space**

Proxy pattern: a class functioning as an **interface to another thing**

Goals of Chain of Responsibility Pattern

1. to **avoid coupling the sender of a request to its receiver**, by giving more than one object a chance to handle the request.
2. **to isolate the clients from knowledge of how responsibilities are assigned.**
3. A request will be sent to the chain of objects. The request will be handled by a chain of servers until the request is resolved.

Loose Connection between Requester and Server



Make sure to understand the code logic of the given example.

- the role of abstract keyword;
- how to chain multiple objects; and
- the role of toString() method.

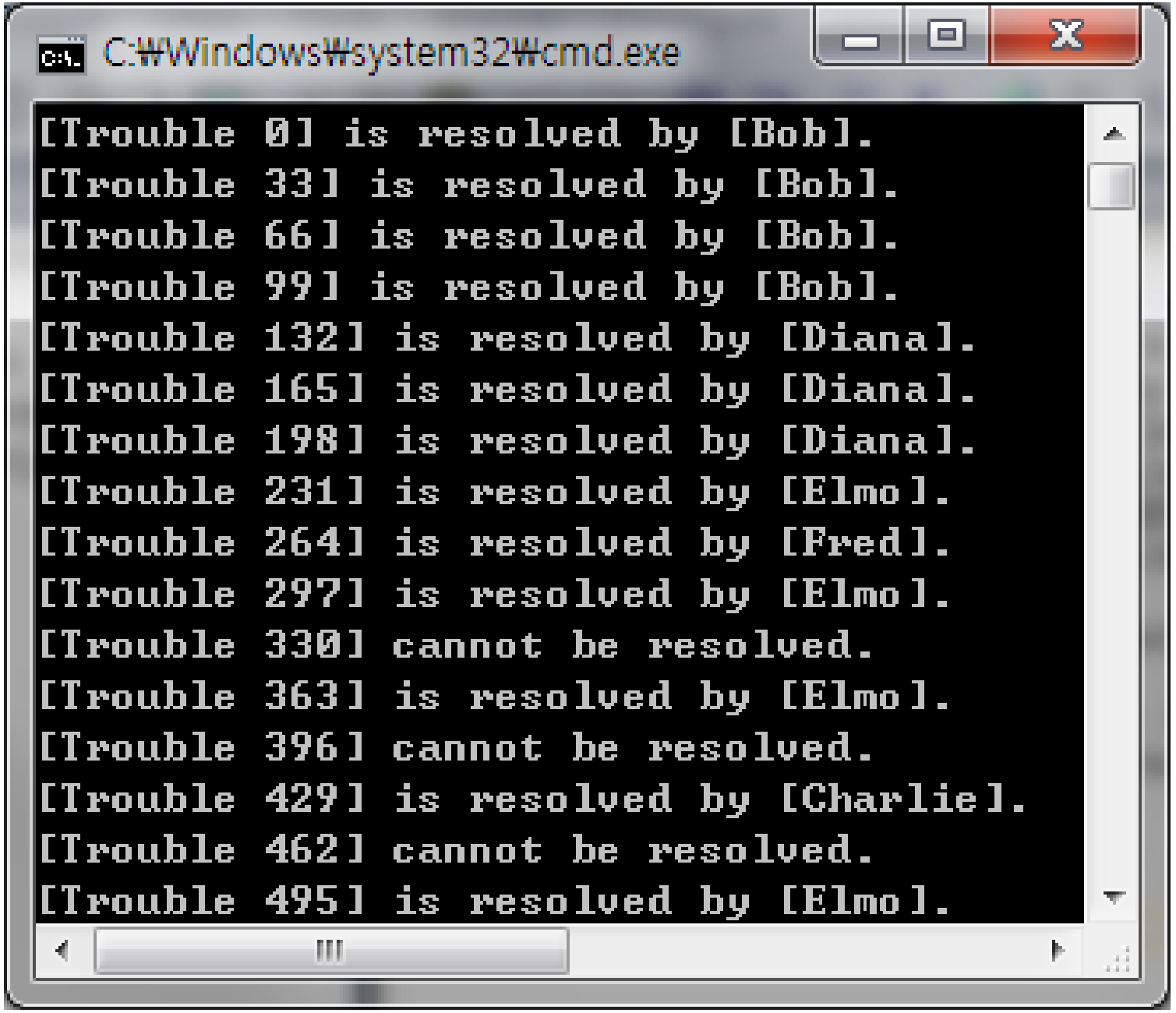
```
public class Main {
    public static void main(String[] args) {
        Support alice   = new NoSupport("Alice");
        Support bob     = new LimitSupport("Bob", 100);
        Support charlie = new SpecialSupport("Charlie", 429);
        Support diana   = new LimitSupport("Diana", 200);
     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

```
public abstract class Support {
    private String name;
    private Support next;
    public Support(String name) {
        this.name = name;
    }
    public Support setNext(Support next) {
        this.next = next;
        return next;
    }
    public final void support(Trouble trouble) {
        if (resolve(trouble)) {
            done(trouble);
        } else if (next != null) {
            next.support(trouble);
        } else {
            fail(trouble);
        }
    }
```

```
 public String toString() {
        return "[" + name + "]";
    }
    protected abstract boolean resolve(Trouble trouble);
    protected void done(Trouble trouble) {
        System.out.println(trouble + " is resolved by " + this + ".");
    }
    protected void fail(Trouble trouble) {
        System.out.println(trouble + " cannot be resolved.");
    }
}
```

```
public class OddSupport extends Support {
    public OddSupport(String name) {
        super(name);
    }
    protected boolean resolve(Trouble trouble) {
        if (trouble.getNumber() % 2 == 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

```
public class SpecialSupport extends Support {
    private int number;
    public SpecialSupport(String name, int number) {
        super(name);
        this.number = number;
    }
    protected boolean resolve(Trouble trouble) {
        if (trouble.getNumber() == number) {
            return true;
        } else {
            return false;
        }
    }
}
```


C:¥Windows¥system32¥cmd.exe
[Trouble 0] is resolved by [Bob].
[Trouble 33] is resolved by [Bob].
[Trouble 66] is resolved by [Bob].
[Trouble 99] is resolved by [Bob].
[Trouble 132] is resolved by [Diana].
[Trouble 165] is resolved by [Diana].
[Trouble 198] is resolved by [Diana].
[Trouble 231] is resolved by [Elmo].
[Trouble 264] is resolved by [Fred].
[Trouble 297] is resolved by [Elmo].
[Trouble 330] cannot be resolved.
[Trouble 363] is resolved by [Elmo].
[Trouble 396] cannot be resolved.
[Trouble 429] is resolved by [Charlie].
[Trouble 462] cannot be resolved.
[Trouble 495] is resolved by [Elmo].

```
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));
        }
    }
}
```

Software Design Pattern

Design Pattern in General Meaning

- A **generic or reusable** solution to a **recurring** problem
- Can be **adapted** and **combined** in many different ways to generate endless possibilities (or *to provide numerous solutions to various problems*)

Software Design Pattern: **reusable** description of ***solutions to recurring (many similar) problems*** in software design, which may be easily **adapted** to various applications

The Purpose of using Software design pattern

- document the experience in a relatively small number of design patterns
- support **reuse** in design that has been proven **effective**
- provide the **common vocabulary** for software designers to communicate about software design in variety
- Program component that can be ***extended***, ***adapted***, and ***reused*** in many different contexts **without** having to modify the source code (or a **minor change)**