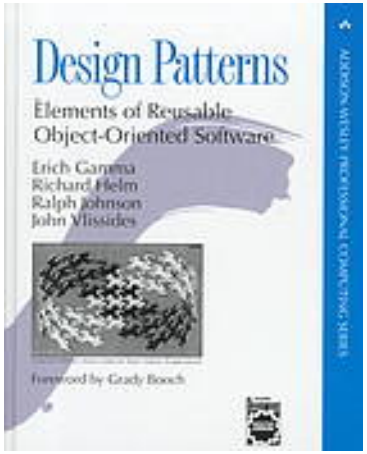
Design Patterns

Classic design patterns, State pattern, Singleton pattern, Adapter pattern

Classic design patterns (GoF)

Classic design patterns usually refer to the original GoF patterns.

- GoF book: Design Patterns. Elements of Reusable Object-Oriented Software
- Authors: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- There are 23 GoF patterns.



Pattern groupings:

- Creational Patterns – for creating objects
- Structural Patterns – composing classes and objects to form larger structures
- Behavioral Patterns – for communication among objects

Creational patterns

- **Singleton**
- Prototype
- Factory/Factory Method
- Abstract Factory
- Builder

Structural patterns

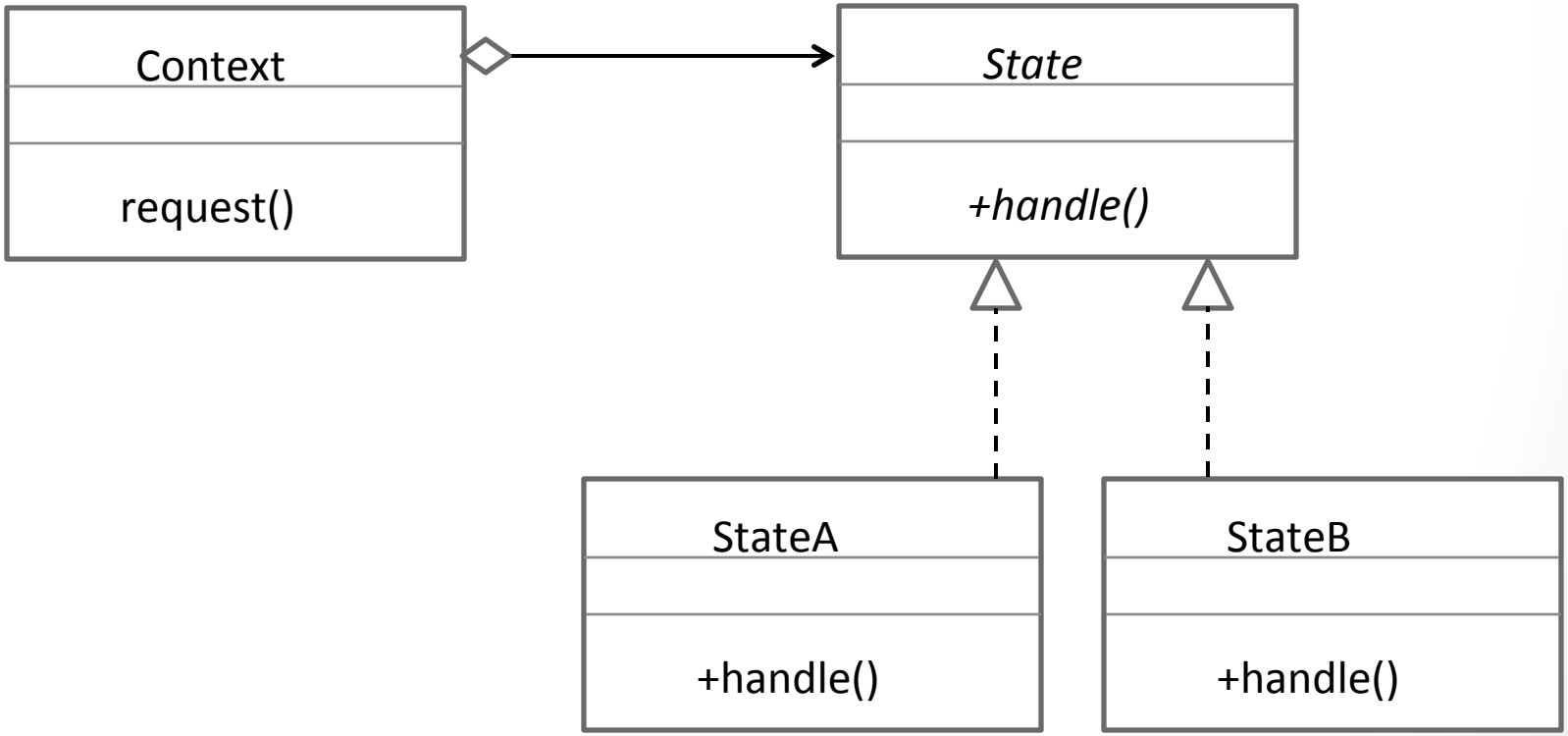
- **Adapter**
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral patterns

- Behavioral Patterns
  - **State**
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - Strategy
  - Template Method
  - Visitor

State pattern

GoF: *"State [pattern] allows an object to alter its behavior when its internal state changes. The object will appear to change its class."*



Singleton pattern

GoF: "Ensure a class has only one instance, and provide a global point of access to it."

Highlights:

- There must be only one instance of the class.
- The class must instantiate itself.
- The instance must be accessible to clients.
- The class must maintain all of its own data.

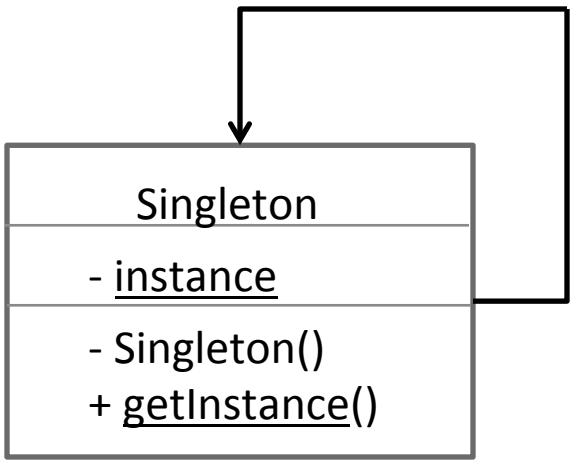
Some real-life uses:

- Window managers
- File systems
- Print spoolers
- Loggers
- Configurations
- Factories

Singleton: UML and code features

Code features:

- Instance variable representing the instance is static.
- Getter for the instance variable is static.
- Constructor is private.



Issues:

- Multiple execution threads.
- Early or lazy instantiation?
  - Early – instance created at outset of execution.
  - Lazy – instance created as soon as one is needed.

Singleton code

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton() { }

   public static Singleton getSingleton() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
}
```

Variations:

- Multithreading:
  public static **synchronized** Singleton getSingleton() { …
- Early instantiation:
  private static Singleton **instance = new Singleton()** ;

Adapter (Wrapper) pattern

GoF: "Convert the interface of one class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Highlights:

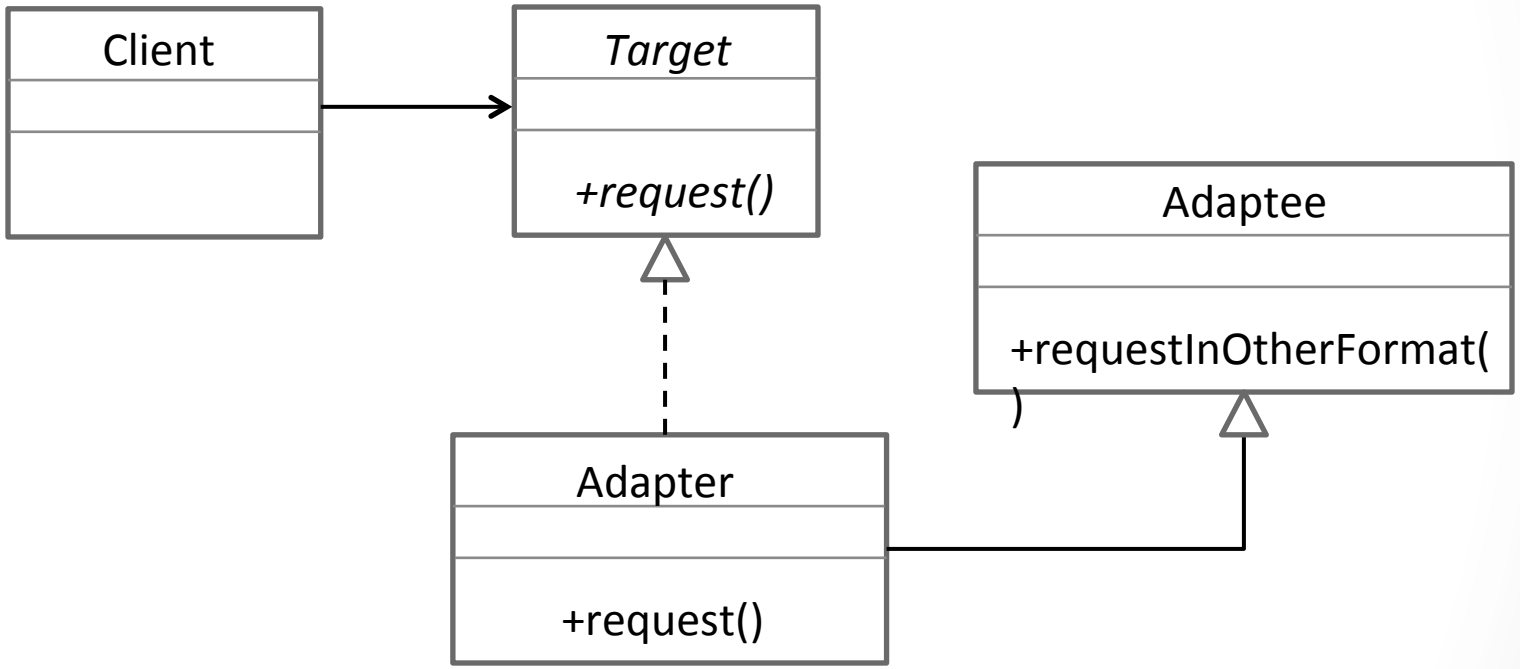
- A **target** interface describes the requests the **client** needs to make.
- An **adaptee** describes operations that could be used if only they were in the correct format (signature, and so on).
- An **adapter** implements the target and either extends or has an instance variable of the adaptee type. Requests are delegated to the adaptee.

Some real-life uses:

- Eclipse plugins
- Library or toolkits that require different interfaces to work with specific domains.
- XML parsers that generate tree structures

Adapter: UML diagram

This is a “class adapter” that uses inheritance. An “object adapter” is similar, except the Adapter would use composition instead of inheritance.



The Adapter pattern is useful when you want to adapt to/use code that has already been written (the Adaptee code).

Adapter example

This example consists of:

- NameList – the Adaptee: a simple list class in which the list elements are names (each consisting of a last name and a first name).
- NamePrinter – Target: an interface that specifies the request messages:
  - to add a new name to the list
  - For printing the list with last names coming first, then first names: <last_name>, <first_name>
  - For printing the list with first names coming before last names: <first_name> <last_name>
- ListAdapter – Adapter: Does the work. ListAdapter implements the Target and extends the Adaptee
- Client: illustrates how to use the adaptor. NamePrinter (the Target) lists the “requests” that the client needs fulfilled.

NameList – the Adaptee

```
import java.util.ArrayList;

public class NameList {
   private ArrayList<String> list = new ArrayList<String>();

   public void add (String first, String last) {
      list.add(first + " " + last);
   }

   public String printNames() {
      String result = "";
      for (String s: list) {
         result += s + "\n";
      }
      return result;
   }
}
```

NamePrinter – the Target

```
public interface NamePrinter {
    /**
     * Adds a new name/element to the list. Each element is
     * a pair of single names.
     * @param one first name in the element pair
     * @param two last name in the element pair
     */
    void add(String one, String two);
    /**
     * A string representation of the list in which elements
     * are represented as <last_name>, <first_name>
     * @return the string representation
     */
    String lastFirst();
    /**
     * A string representation of the list in which elements
     * are represented as <first_name> <last_name>
     * @return the string representation
     */
    String firstFirst();
}
```

ListAdapter – the Adapter

```
import java.util.Scanner;

public class ListAdapter extends NameList
                         implements NamePrinter{
    @Override
    public String firstFirst() {
        return super.printNames();
    }

    @Override
    public String lastFirst() {
        String result = "";
        Scanner scan = new Scanner(super.printNames());
        while (scan.hasNextLine()) {
            String first = scan.next();
            result += scan.next() + ", " + first + "\n";
            scan.nextLine();
        }
        return result;
    }
}
```

Client – the Client

```
public class Client {
   public static void main(String[] args) {
      NamePrinter x = new ListAdapter();
      x.add("John", "Lennon");
      x.add("Paul", "McCartney");
      x.add("George", "Harrison");
      x.add("Ringo", "Starr");

      System.out.println("First name first:");
      System.out.println(x.firstFirst());
      System.out.println("Last name first:");
      System.out.println(x.lastFirst());
   }
}
```

First name first:
John Lennon
Paul McCartney
George Harrison
Ringo Starr

Last name first:
Lennon, John
McCartney, Paul
Harrison, George
Starr, Ringo

References

- Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley, 1995.