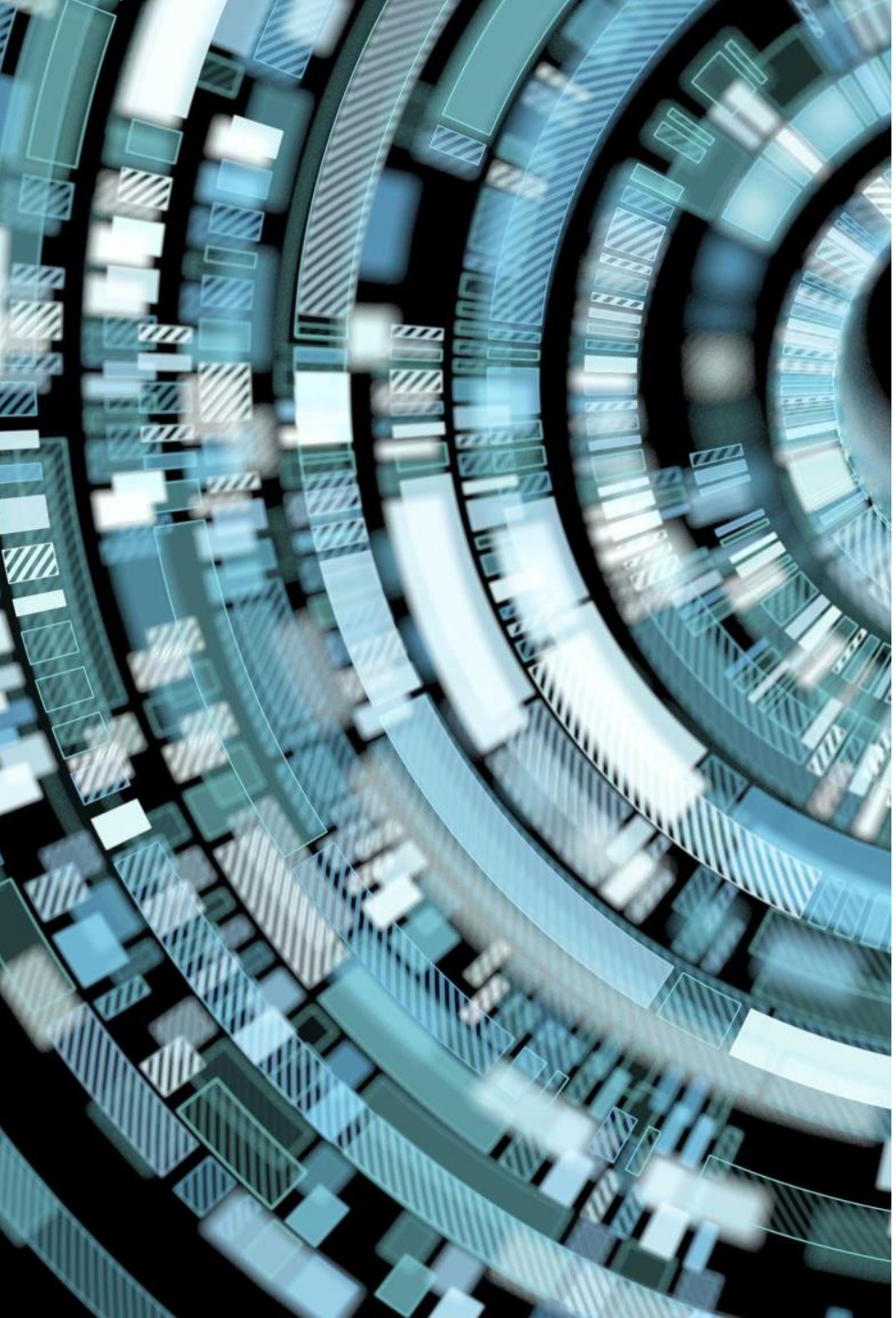
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 14

- Design Patterns
  - Overview
  - Classification
  - Intro to 6 design patterns

Design Patterns

The concept of **patterns** was first described by Christopher Alexander, an architect and design theorist, who describes a language for designing the urban environment.

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice“

Software Design Patterns

- The idea was later picked up Erich Gamma et al., who published **Design Patterns: Elements of Reusable Object-Oriented Software** in 1994
- The book featured 23 patterns solving various problems of object-oriented and software design, and quickly became a best-seller
- Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

https://refactoring.guru/design-patterns

Patterns vs Algorithms

- Patterns are often confused with algorithms
  - Both concepts describe typical solutions to some known problems
- Algorithm always defines a clear set of actions that can achieve some goal
- A pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

Essential Elements of a Pattern

The Pattern Name

A handle we can use to describe a design problem, its solutions, and consequences in a word or two

The Problem

Explains the problem and the context, and when to apply the pattern

The Solution

Describes how a general arrangement of elements solves the problem (e.g., the relationships, responsibilities, and collaborations between classes and objects)

The Consequences

Describes the results and trade-offs of applying the pattern

Design Patterns: Elements of Reusable Object-Oriented Software. Gamma et al.

Classification of Design Patterns

https://refactoring.guru/design-patterns

Creational Patterns

- Provide various object creation mechanisms, which increase flexibility and reuse of existing code
- E.g., **Factory Method, Singleton**

Structural Patterns

- Explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient
- E.g., **Decorator, Composite**

Behavioral Patterns

- Handle algorithms and the assignment of responsibilities between objects
- E.g., **Strategy, Command**

Factory Method Pattern

The Problem

Context

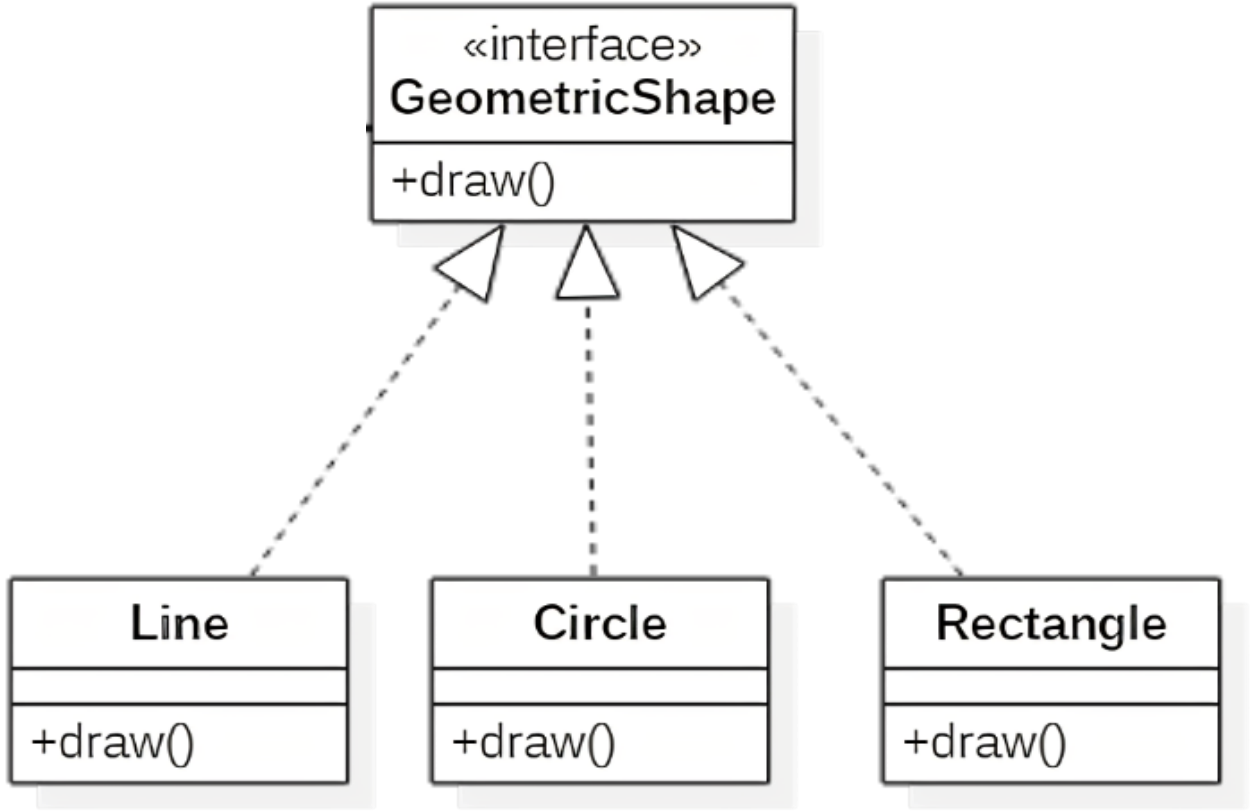
- Suppose that your system needs to create a bunch of similar objects
- The logics of creating these objects are similar, but could be complicated

Problems

- Adding new objects means that duplicate object creation code will be added in many places
- If the object creation logics is revised, the system also needs to update all the object creation code accordingly

Solution I – Simple Factory Pattern

First, adding a level of abstraction (interface, superclass) to similar objects, or products, that are being created

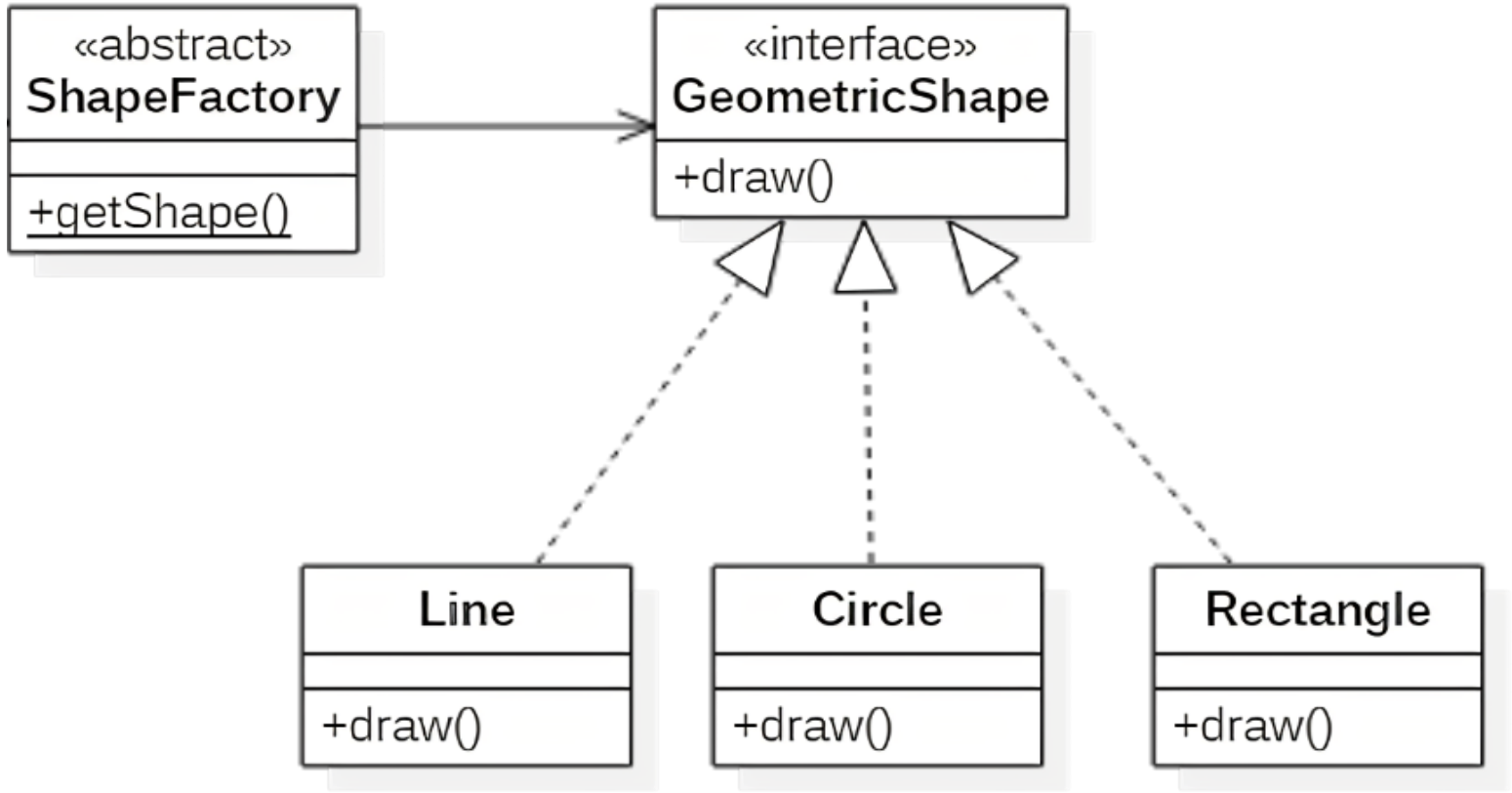


Full example: https://dzone.com/articles/factory-method-design-pattern

Solution I – Simple Factory Pattern

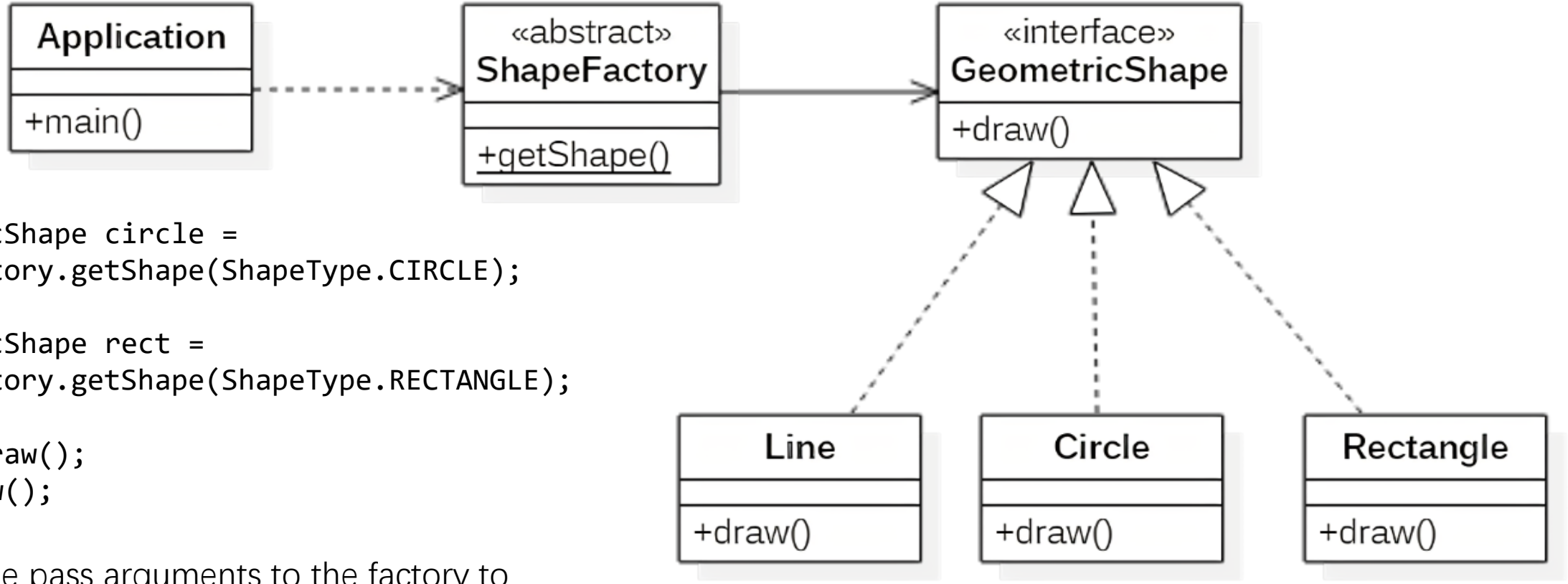
Then, define a factory class, which implements a method that produces (returns) the general object being created

```
public abstract class ShapeFactory {
    public static GeometricShape getShape(ShapeType name) {
        GeometricShape shape = null;
        switch (name) {
            case LINE:
                shape = new Line();
                break;
            case CIRCLE:
                shape = new Circle();
                break;
            case RECTANGLE:
                shape = new Rectangle();
                break;
        }
        return shape;
    }
}
```



Full example: https://dzone.com/articles/factory-method-design-pattern

Solution I – Simple Factory Pattern



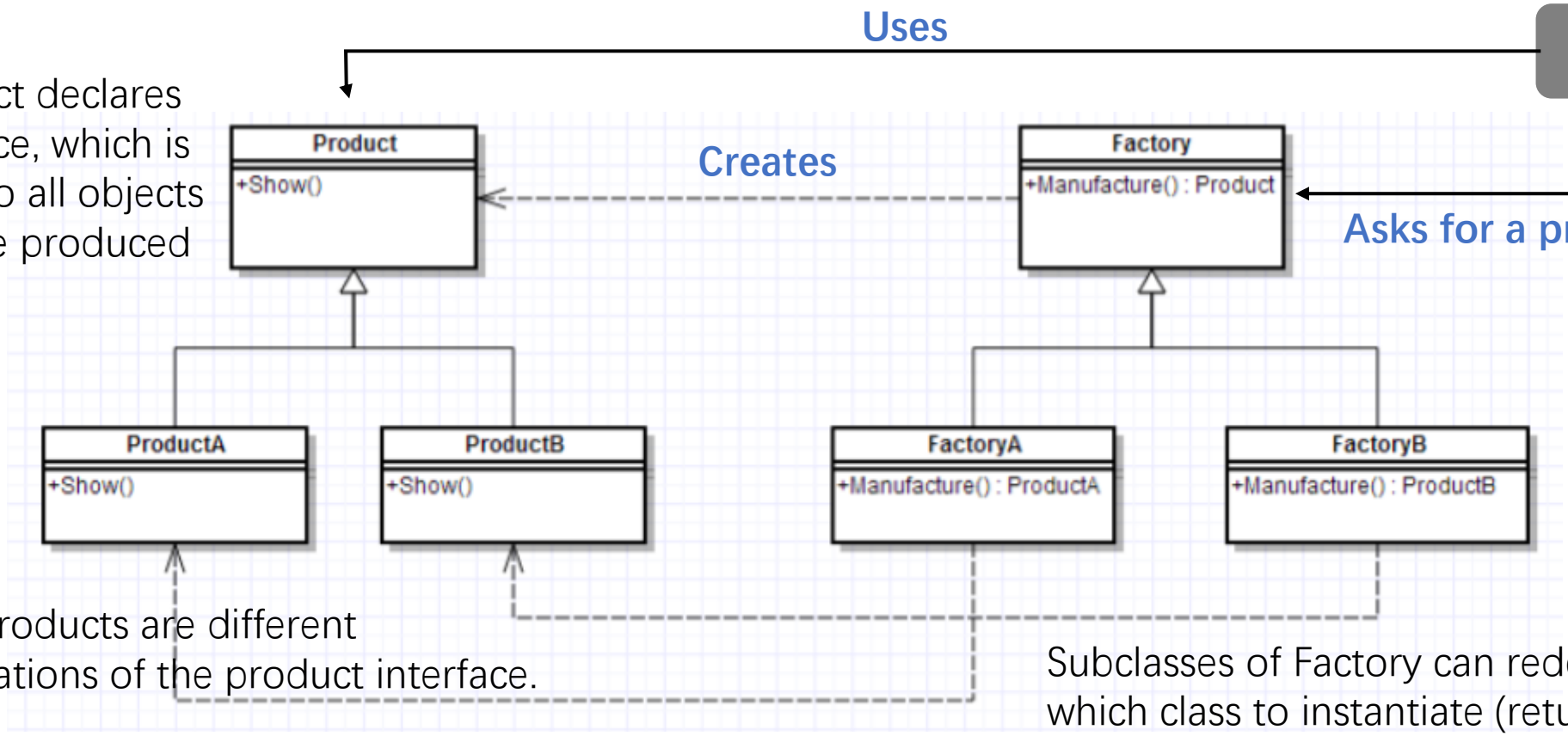

```
GeometricShape circle =
ShapeFactory.getShape(ShapeType.CIRCLE);

GeometricShape rect =
ShapeFactory.getShape(ShapeType.RECTANGLE);

circle.draw();
rect.draw();
```

Client code pass arguments to the factory to create objects, instead of new object directly. Polymorphism is enabled.

Solution II – Factory Method

Objects might be created in different ways

Replace direct object construction calls (using the new operator) with calls to a special factory method



The Product declares the interface, which is common to all objects that can be produced

The Factory refers to a separate factoryMethod() to create a Product object

Concrete Products are different implementations of the product interface.

Subclasses of Factory can redefine which class to instantiate (return type)

Factory Method – Example I

Collection Interface

public interface **Collection<E>**
extends Iterable<E>

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

    Object[] toArray();
    T[] toArray(T a[]);

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> c); // Optional
    boolean removeAll(Collection<?> c);  // Optional
    boolean retainAll(Collection<?> c);  // Optional
    void clear();                        // Optional
}
```

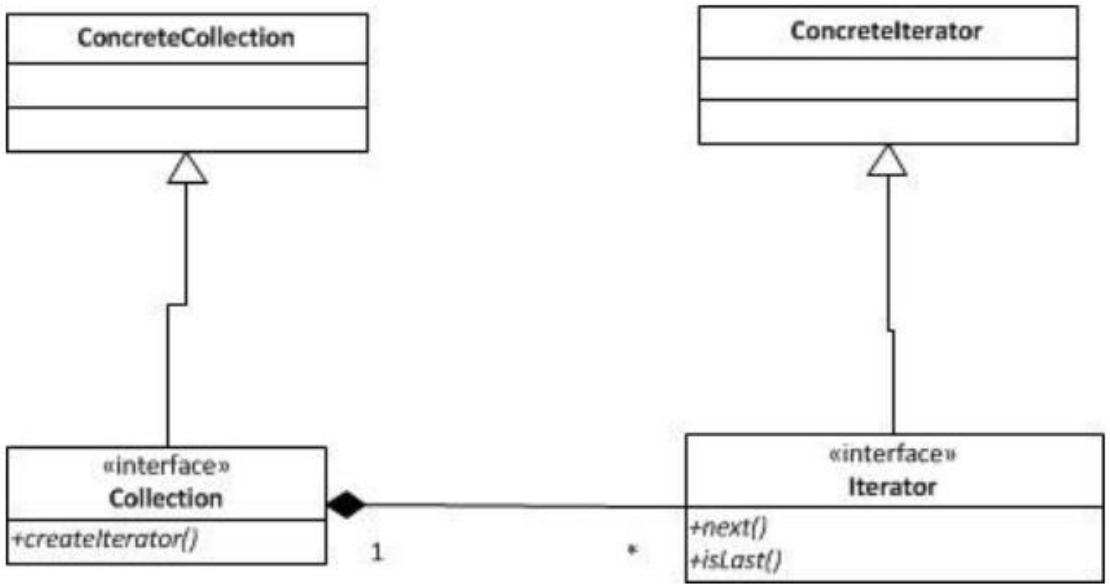
"Optional" means that classes implementing this interface does not necessarily have to implement that method (e.g., read-only collection)

Next slide

批量操作

15-214 12 35

`ArrayList`, `HashSet`, etc. each has its own concrete implementation of Iterator



`Collection` is the factory/creator, which creates the `Iterator`

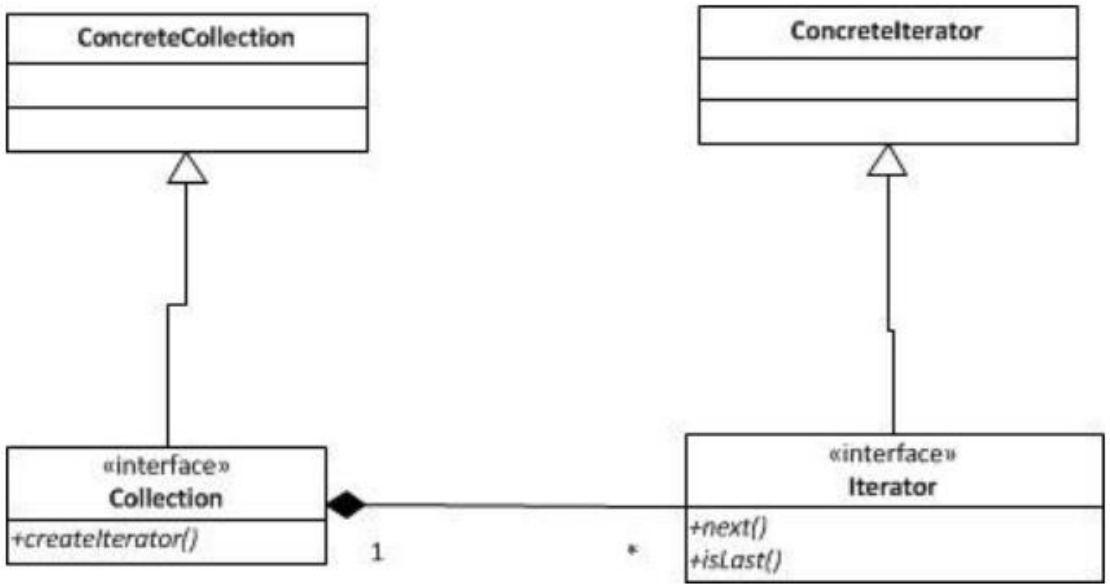
Image: https://javapapers.com/design-patterns/iterator-design-pattern/

Factory Method – Example I

You don't need to know what type of collections you're using, each collection implementation will provide an Iterator through the factory method iterator().

```
List<String> list = new ArrayList<String>( );
list.add( "dog" );
...
// create an iterator
Iterator<String> iter = list.iterator( );
while ( iter.hasNext() ) System.out.println( iter.next() );
```

`ArrayList`, `HashSet`, etc. each has its own concrete implementation of Iterator



`Collection` is the factory/creator, which creates the `Iterator`

Image: https://javapapers.com/design-patterns/iterator-design-pattern/

Factory Method – Example II

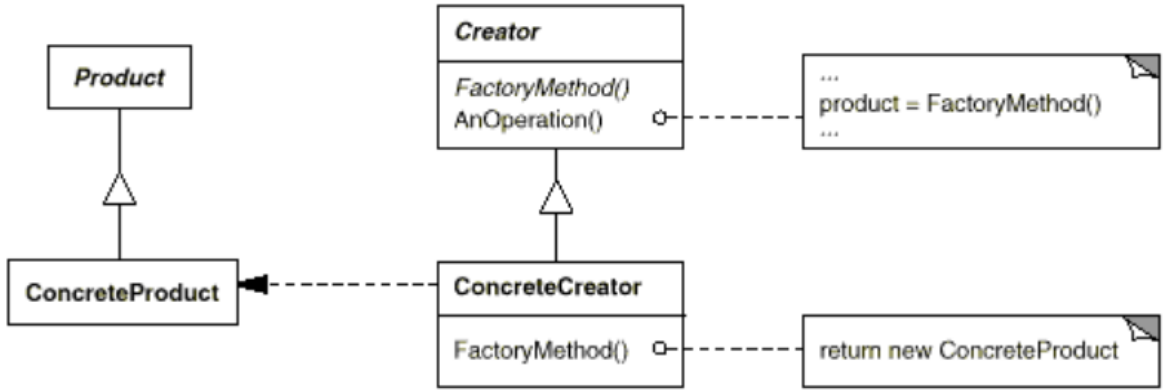
- There are different kinds of enemies with similar behaviors (products to be created)
- The game has different levels, which have different ways to create enemies (object creation logics)

- Products and concrete products: enemies
- Factory and concrete factories: level1Factory, level2Factory, etc.……

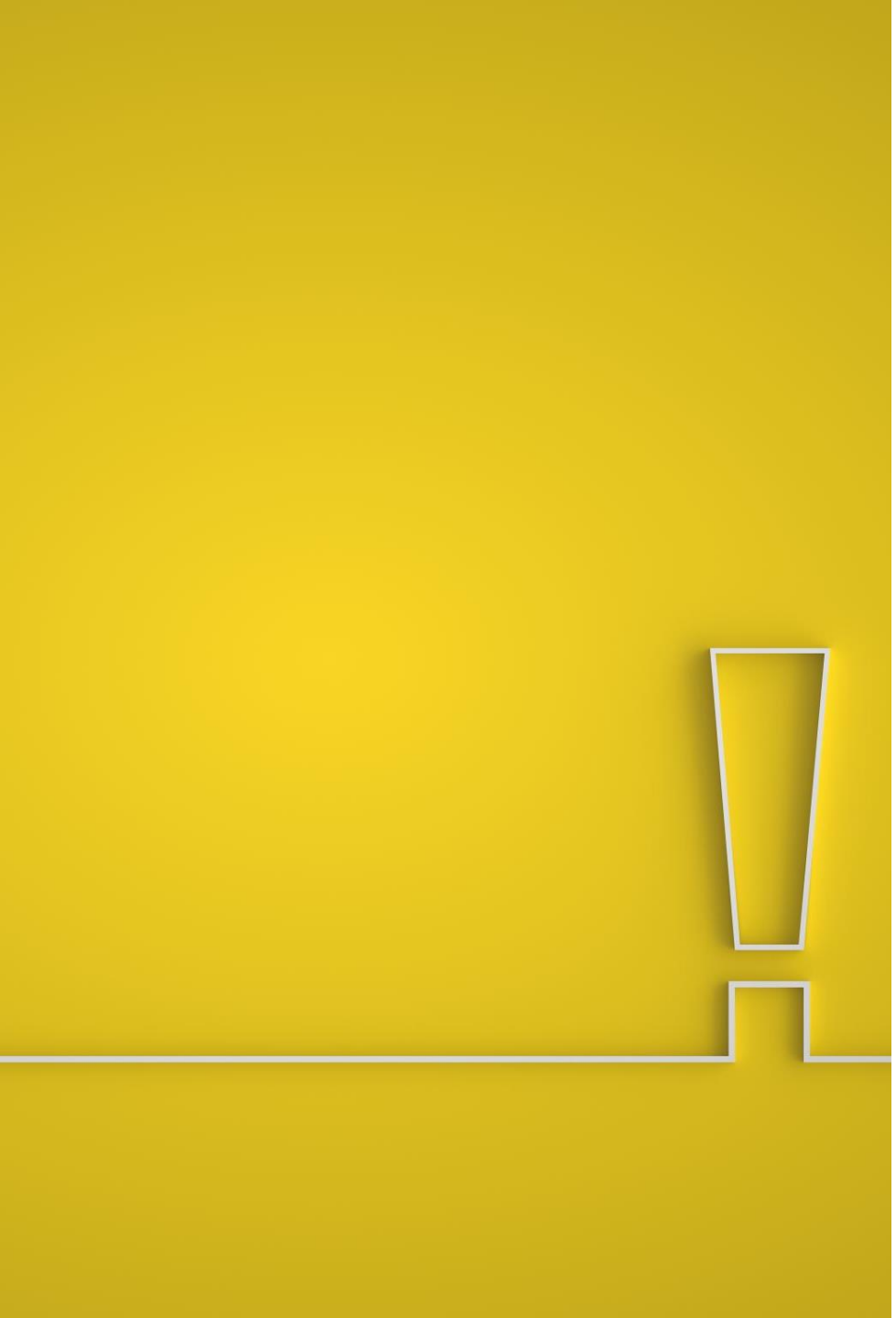
Factory Method - Summary

- A creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses

•**Factory/Creator** - declares the factory method and may provide a default implementation.
•**Concrete Factory/Creator** - implements or overrides the factory method to return a Concrete Product.
•**Product** - defines the interface for objects created by the factory method.
•**Concrete Product** - implements the Product interface.



Singleton Design Pattern



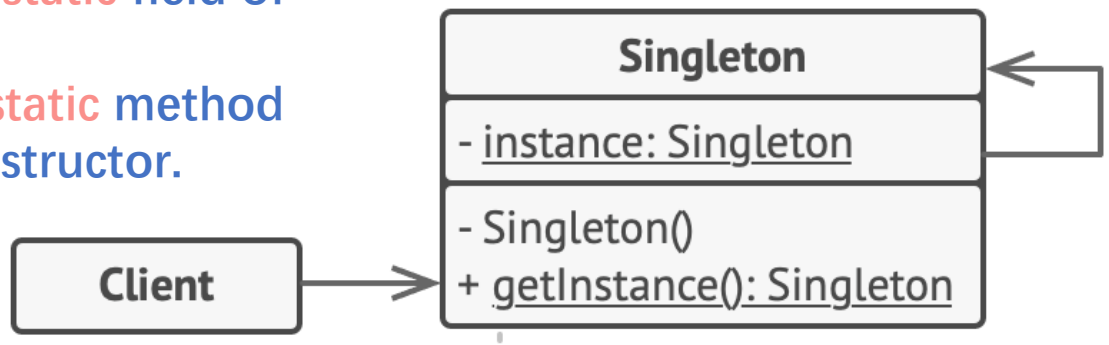
Singleton Design Pattern

A creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

- A class has only one instance: it is impossible to create a second instance of that class
- A global access point to this instance: everytime you acquire an instance of a singleton class, you always get the same instance

Singleton Design Pattern

- Declaring a private static field of the class type
- Declaring a public static method that acts as the constructor.



The default constructor is private, to prevent other objects from using the new operator with the Singleton class

- Under the hood, this method calls the private constructor to create an object and saves it in a static field.

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Clients can only invoke the public method to get the class instance, which is the same instance all the time

https://refactoring.guru/design-patterns/singleton

Singleton Design Pattern: Example

Singleton can be recognized by a static creation method, which returns the same cached object.

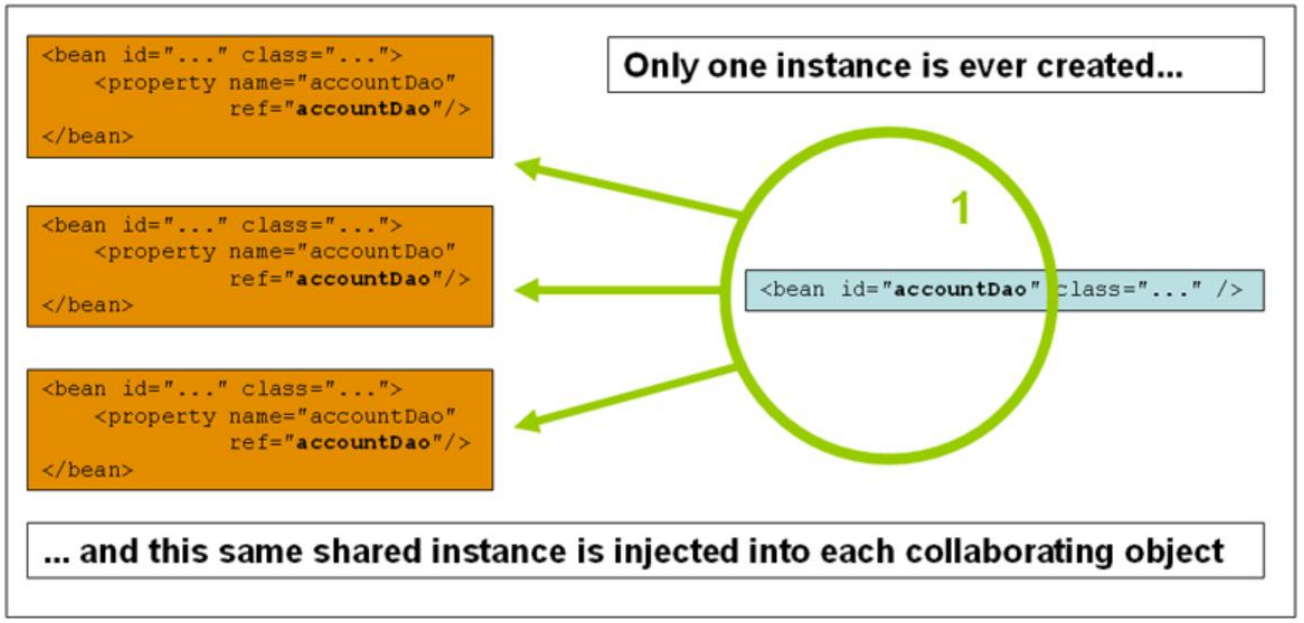
`java.lang.Runtime#getRuntime()`

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
```

The Singleton Bean Scope in Spring

Spring restricts a singleton to one bean object per Spring IoC container.

This single instance will be stored in a cache of such singleton beans, and all subsequent requests and references for that named bean will result in the cached object being returned.

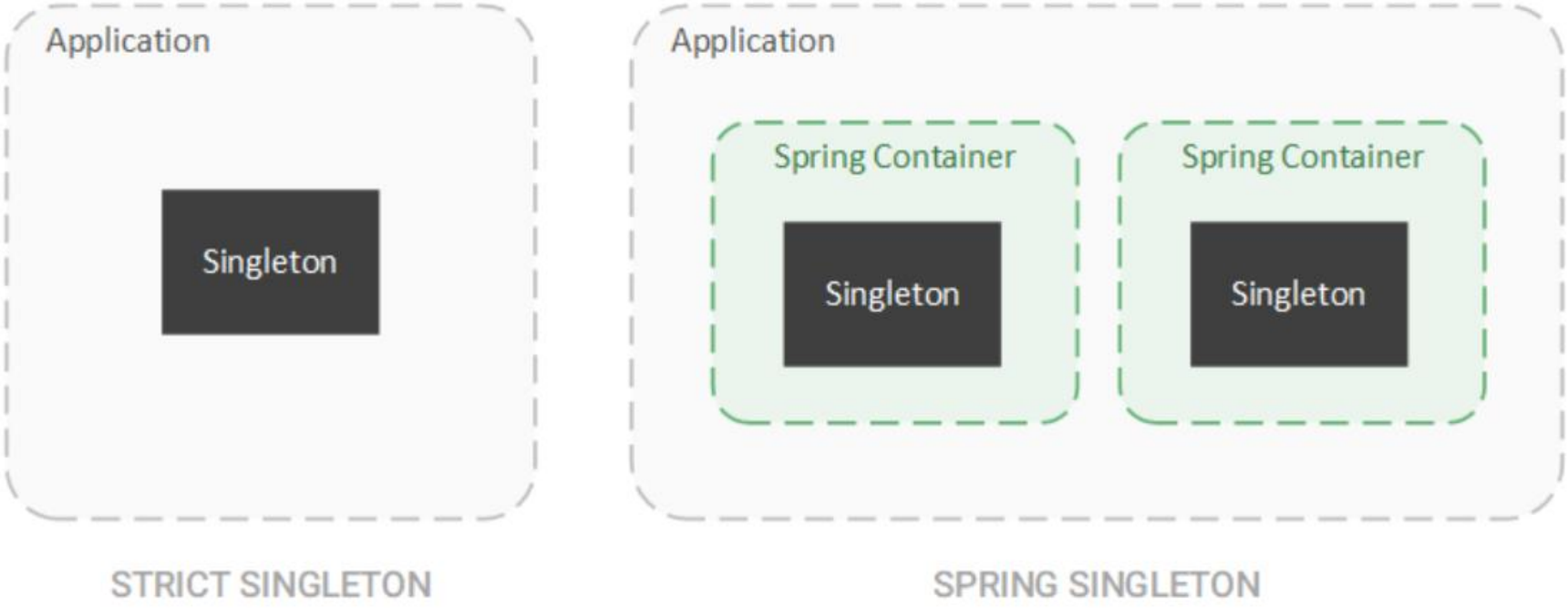


when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition

https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html

The Singleton Bean Scope in Spring

- Spring's approach differs from the strict definition of a singleton since an application can have more than one Spring container. Therefore, multiple objects of the same class can exist in a single application if we have multiple containers.



https://www.baeldung.com/spring-framework-design-patterns

Classification of Design Patterns

https://refactoring.guru/design-patterns

Creational Patterns

- Provide various object creation mechanisms, which increase flexibility and reuse of existing code
- E.g., Factory Method, Singleton

Structural Patterns

- Explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient
- E.g., **Decorator, Composite**

Behavioral Patterns

- Handle algorithms and the assignment of responsibilities between objects
- E.g., **Strategy, Command**

TAO Yida@SUSTECH

Decorator Pattern

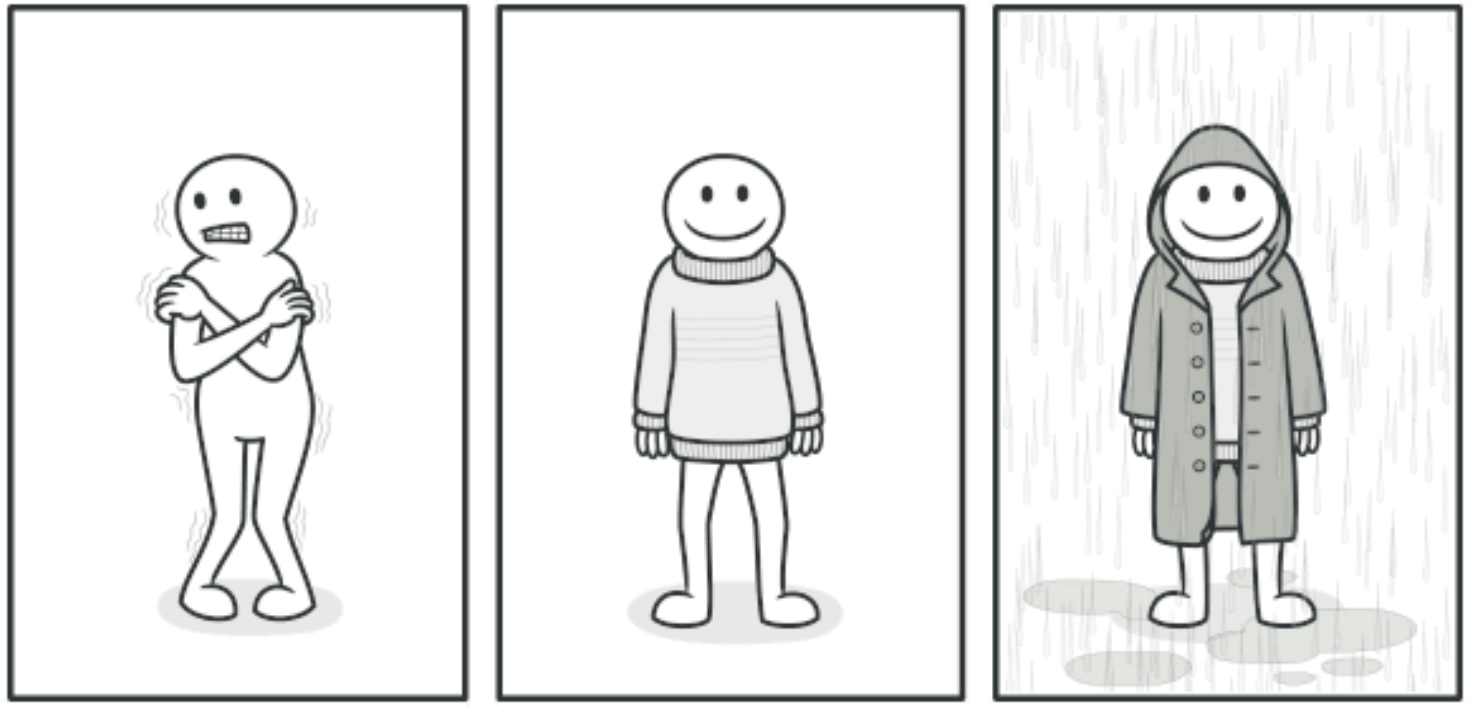
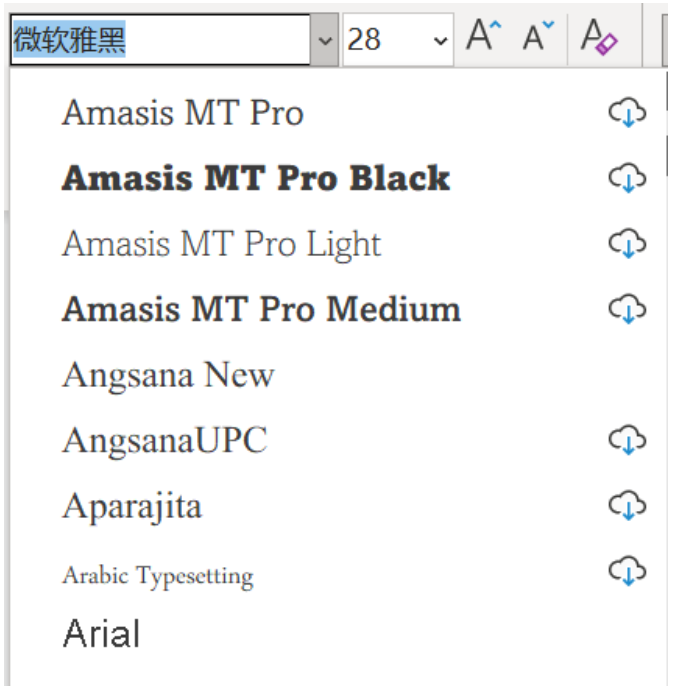


Image source: https://refactoring.guru/design-patterns/decorator

The Problem

- Imagine a text editor that support different styles for differ fonts (字体).
- How would you design the text editor software?



Bold (黑体), Italian (斜体), Underline (下划线), Superscript (上标)……

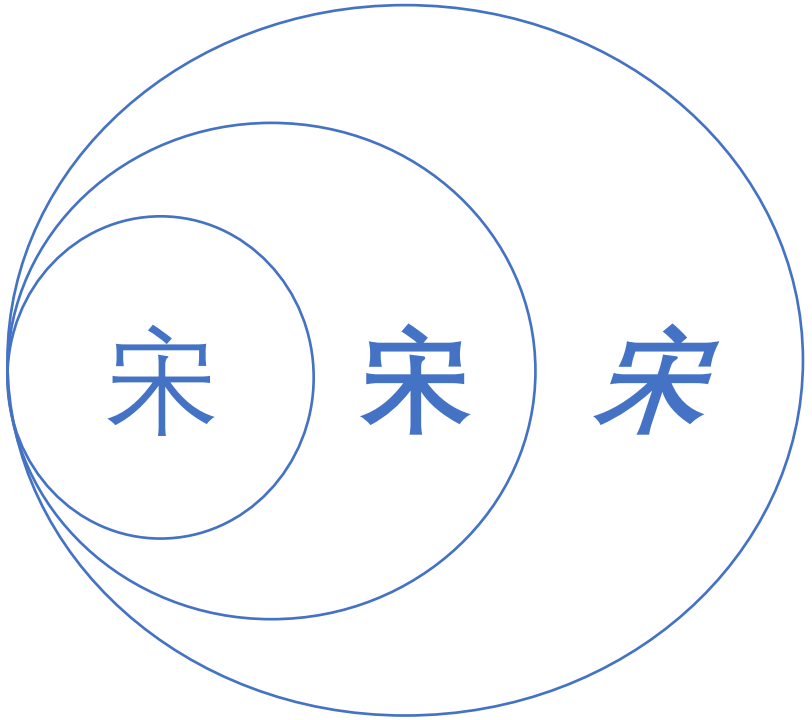
The Problem

- Can we design one class for each possibility?
- Class explosion problem: n styles, 2^n-1 combinations
- Inefficient code reuse

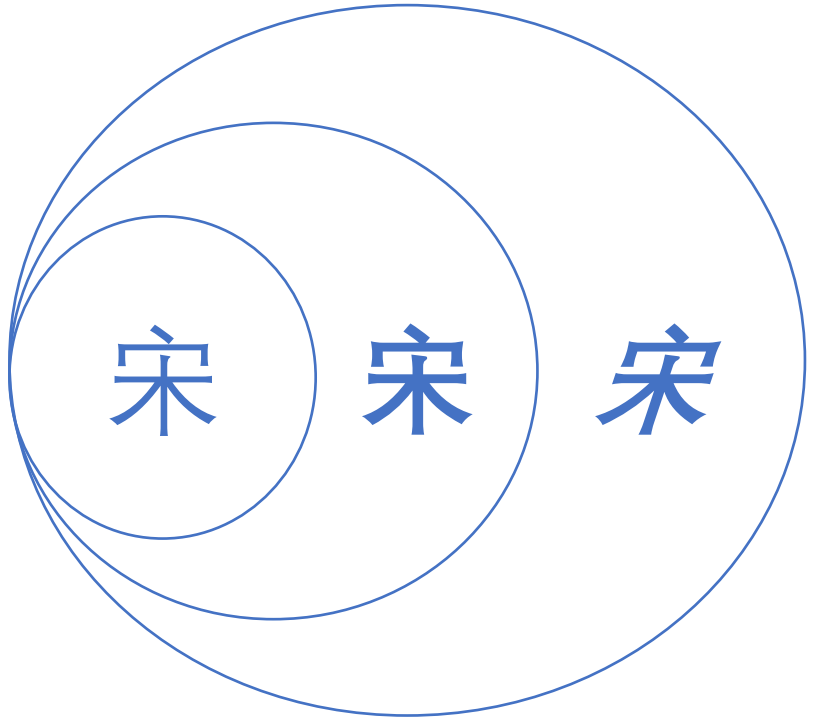
class SongBold: 宋体 Song + 黑体 (Bold)
class SongItalian: 宋体 Song + 斜体 (Italian)
class SongUnderline: 宋体 Song + 下划线 (Underline)
class SongBoldItalian: 宋体 Song +黑体 (Bold) + 斜体 (Italian)
class SongBoldUnderline: 宋体 Song +黑体 (Bold) +下划线 (Underline)
class SongItalianUnderline: 宋体 Song +斜体 (Italian) +下划线 (Underline)
……

The Solution – Decorator Pattern

- Decorator pattern is a structural design pattern that allows behavior to be added to an individual object, dynamically
  - New behaviors are added at runtime
  - No need to revise the code for the original class



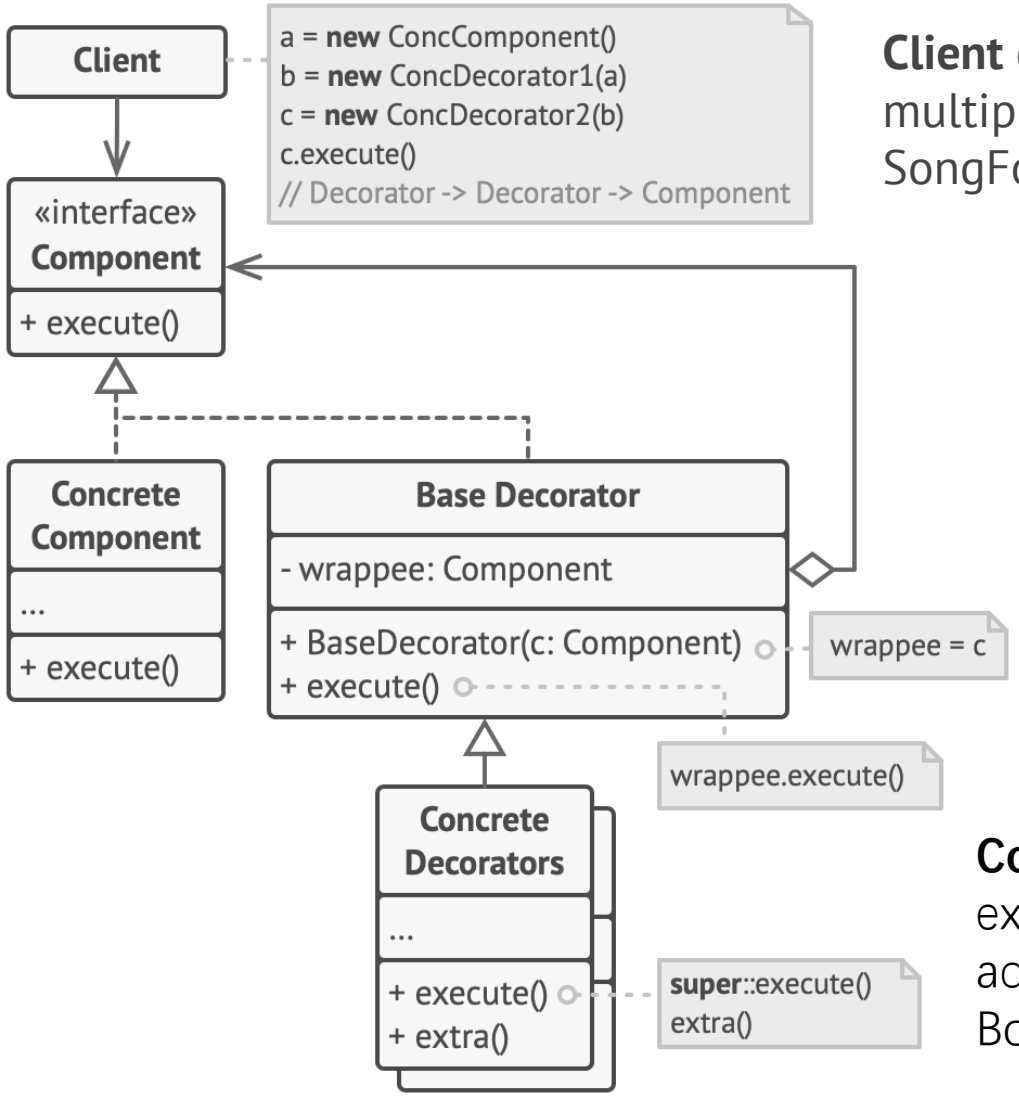
The Solution – Decorator Pattern



Decorator – The UML

The **Component** declares the common interface (e.g., `TextComponent`) for both decorators (e.g., `Styles`) and decorated objects (e.g., `Fonts`).

Concrete Component is a class of objects being decorated. It defines the **basic behavior**, which can be altered by decorators. (e.g., `SongFont.show()`)



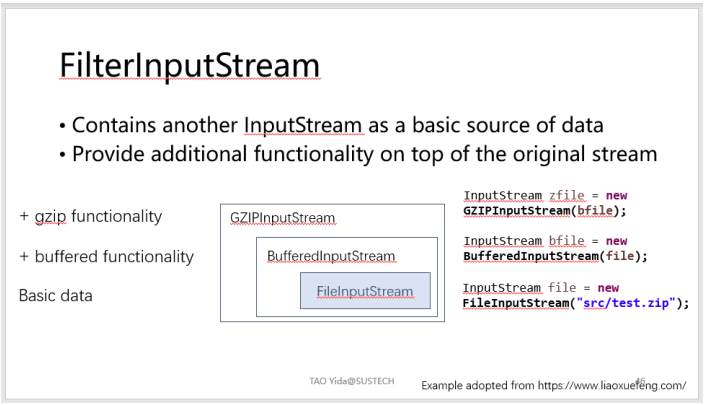
Client can wrap components in multiple layers of decorators (e.g., SongFont + Bold + Italian)

Base Decorator class **is a** Component, and **has a** field with the type **Component**, which could be constructed with either a **Component or a Decorator** (e.g., Song + Bold)

Concrete Decorators define extra behaviors that can be added to components. (e.g., Bold Song Font, show()+bold())

Image: https://refactoring.guru/design-patterns/decorator

Decorator - Example



Lecture 2

Component

read()

InputStream

File InputStream

Piped InputStream

ByteArray InputStream

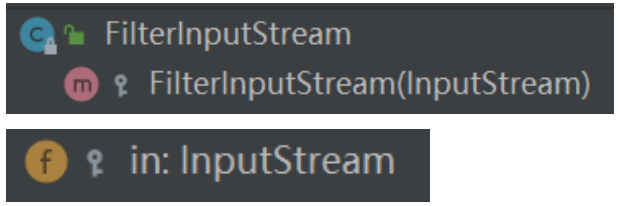
Filter InputStream

Data InputStream

Buffered InputStream

Pushback InputStream

Concrete Component



Base Decorator

Concrete Decorator

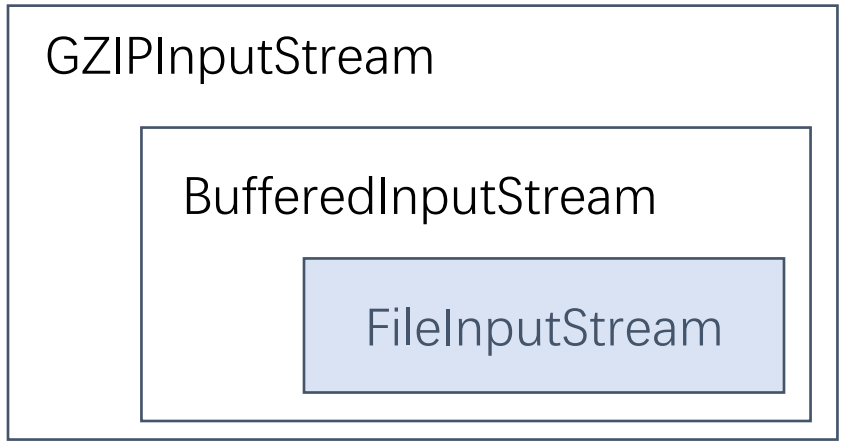
Recall: FilterInputStream

- Contains another InputStream as a basic source of data
- Provide additional functionality on top of the original stream

+ gzip functionality

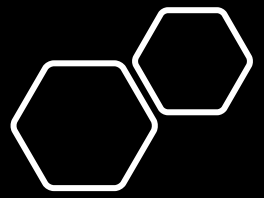
+ buffered functionality

Basic data




```
InputStream zfile = new
GZIPInputStream(bfile);

InputStream bfile = new
BufferedInputStream(file);

InputStream file = new
FileInputStream("src/test.zip");
```

Example adopted from https://www.liaoxuefeng.com/



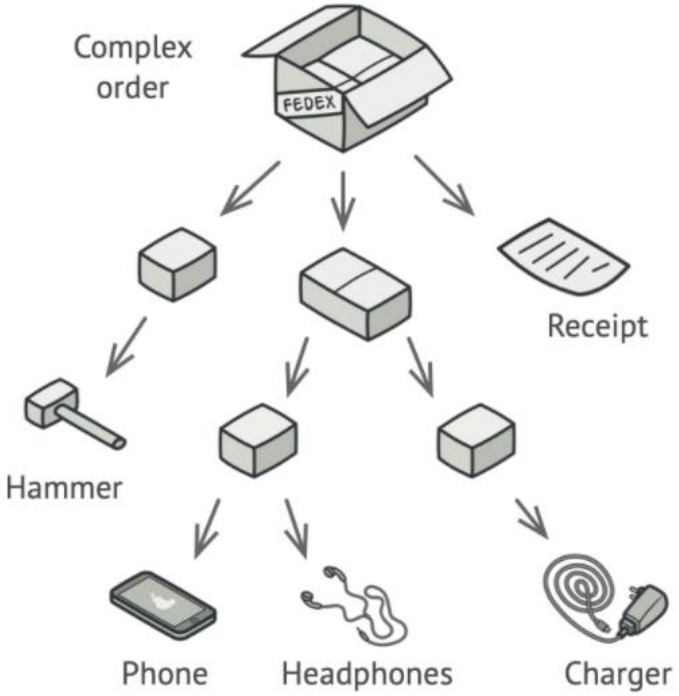
Decorator Summary

Decorator Pattern allows you to attach new behaviors to objects at runtime by placing these objects inside special wrapper objects that contain the new behaviors.

Composite Design Pattern

The Problem

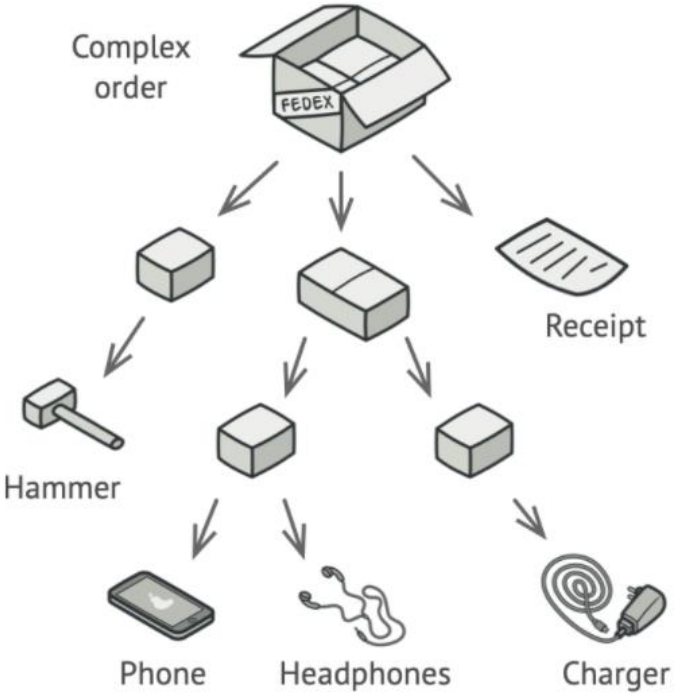
- Imagine that you have two types of objects: Products and Boxes
  - A Box can contain several Products and/or smaller Boxes
  - Smaller Boxes can also hold Products and/or even smaller Boxes….
- You decide to create an ordering system such that orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes.
- How would you determine the total price of such an order?



https://refactoring.guru/design-patterns/composite

The Solution – Composite Pattern

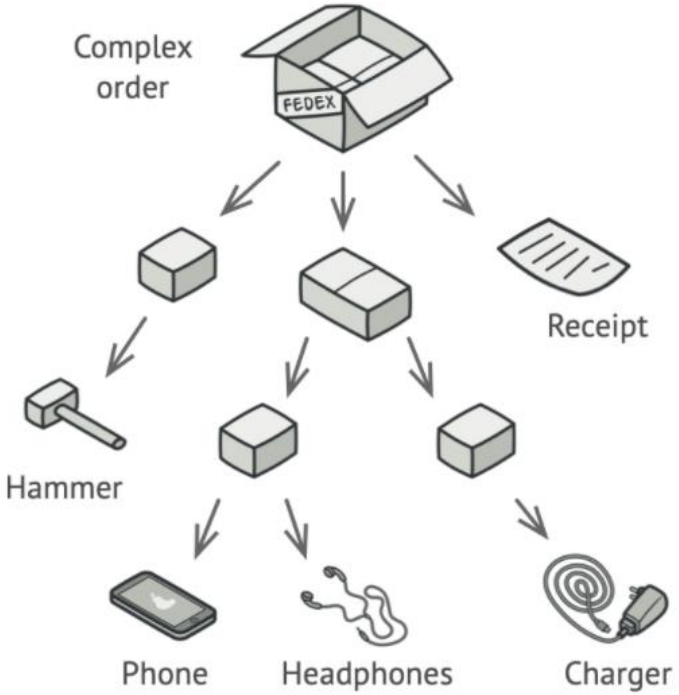
- A structural design pattern that lets you compose objects into tree structures to represent part-whole hierarchies.
- Composite lets client treat individual objects and compositions of objects uniformly
- The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them



https://refactoring.guru/design-patterns/composite

The Solution – Composite Pattern

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.
  - For a product, it would simply return the product’ s price.
  - For a box, it would go over each item the box contains, ask its price and then return a total for this box



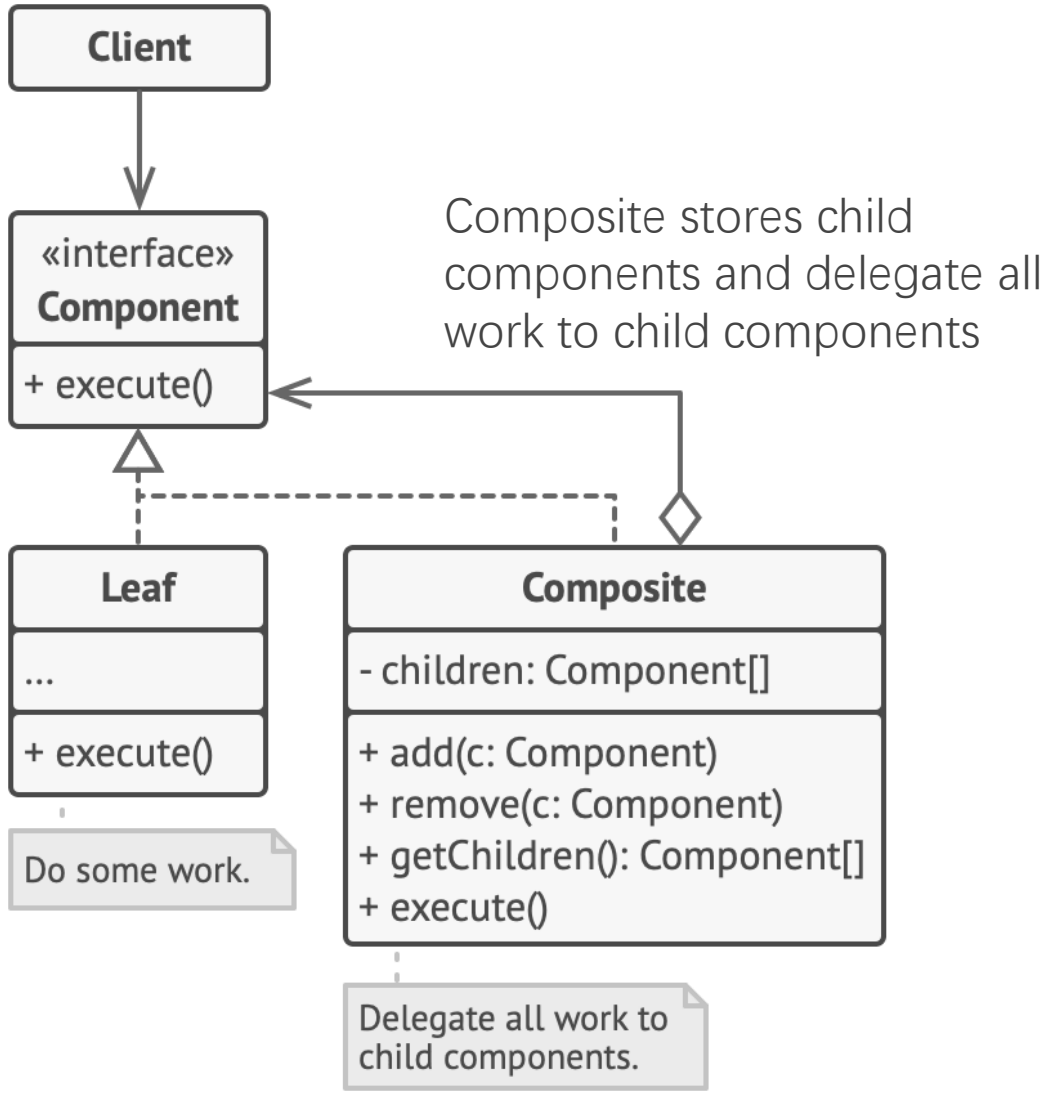
https://refactoring.guru/design-patterns/composite

The Client works with all elements through the component interface. As a result, the client can work in the same way with both leaf and composite of the tree.

Composite Design Pattern

The Component interface describes operations that are common to both simple and complex elements of the tree.

Leaf defines behavior for primitive objects in the composition



https://refactoring.guru/design-patterns/composite

Composite Design Pattern: Example

The AWT class hierarchy

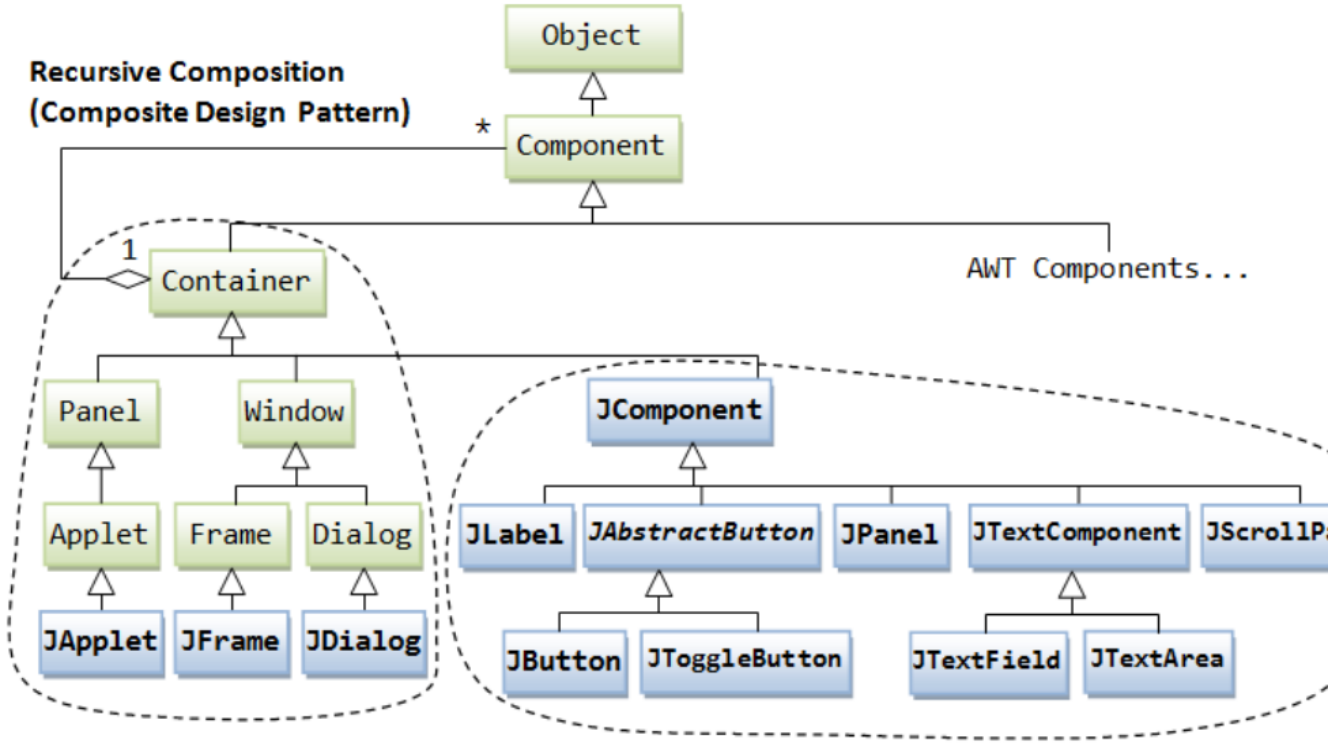


Image source: https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html

```
public class Container extends Component
{
  /**
   * Compatible with JDK 1.0+.
   */
  private static final long serialVersionUID = 4613797578919906343L;

  /* Serialized fields from the serialization spec. */
  int ncomponents;
  Component[] component;
```

add(Component comp)
Appends the specified component to the end of this container.

remove(Component comp)
Removes the specified component from this container.

public Component[] getComponents()

Gets all the components in this container.

setVisible(boolean b)
Shows or hides this component depending on the value of parameter b.

Composite vs. Decorator

- Composite pattern is used when the problem has an inherent tree structure
- Composite delegates work to basic components (leaves), while decorator adds new behaviors to basic components
- Composite could contain multiple components, while decorator typically decorates only one component (a decorator can be viewed as a degenerate composite with only one component)

Classification of Design Patterns

https://refactoring.guru/design-patterns

Creational Patterns

- Provide various object creation mechanisms, which increase flexibility and reuse of existing code
- E.g., Factory Method, Singleton

Structural Patterns

- Explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient
- E.g., Decorator, Composite

Behavioral Patterns

- Handle algorithms and the assignment of responsibilities between objects
- E.g., **Strategy, Command**

Strategy – The Problem

Imagine there is a TextValidator that could check the validity of a piece of text according to different strategies

- Strategy 1: text must be all lowercased
- Strategy 2: text must be all numbers
- Strategy 3: text must contain both numbers and letters
- ……

Putting all these strategies inside of one single TextValidator class might work, but

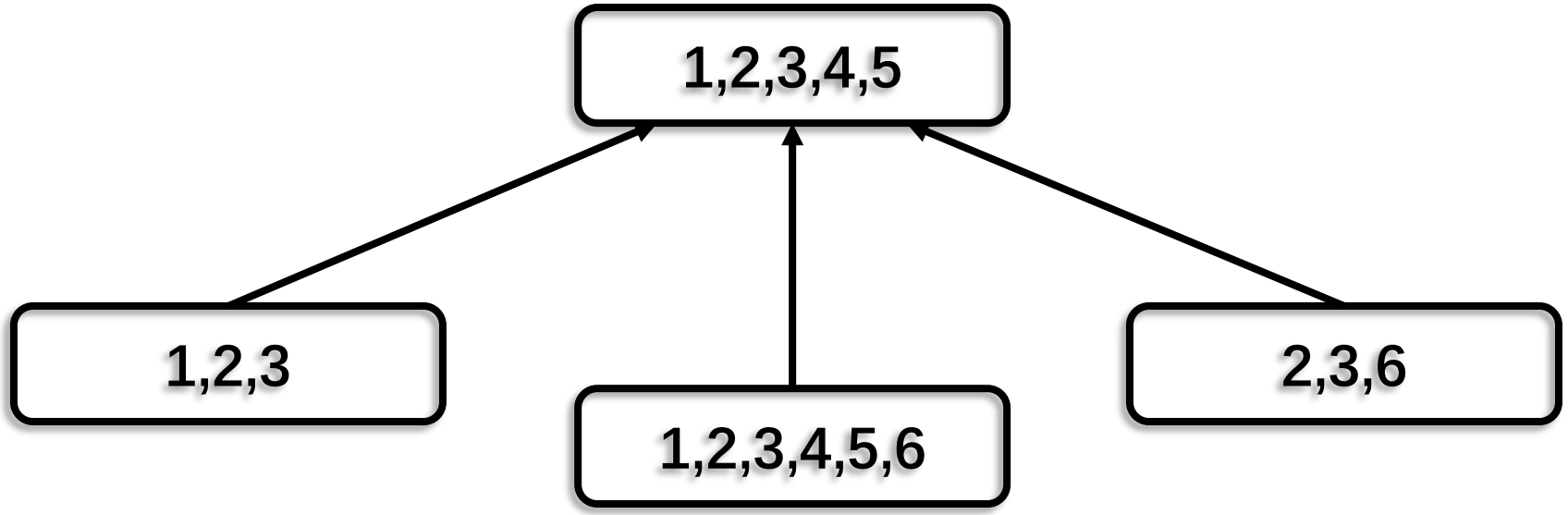
- The class becomes a beast if as more and more strategies are added
- The class is tangled, error-prone, hard to maintain
- What about code reuse?

Strategy – The Problem

We might use inheritance for reusing the strategies, but…

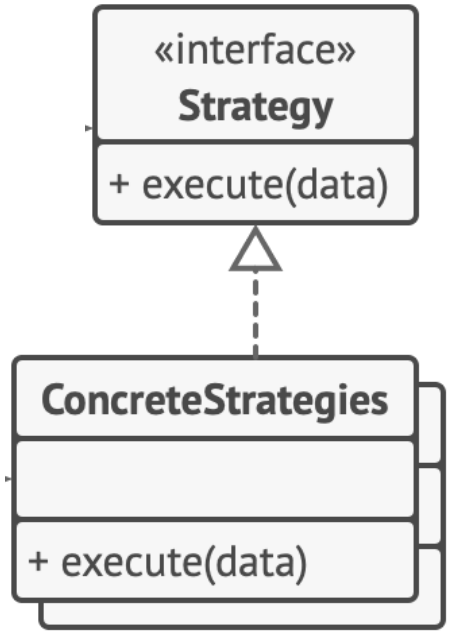
What if a subclass TextValidator does not need certain strategies defined in the superclass?

What if a subclass needs to reuse the strategies from another subclass?



Strategy – The Solution

- Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- This pattern enables selecting an algorithm at runtime, and lets the algorithm vary independently from clients that use it.



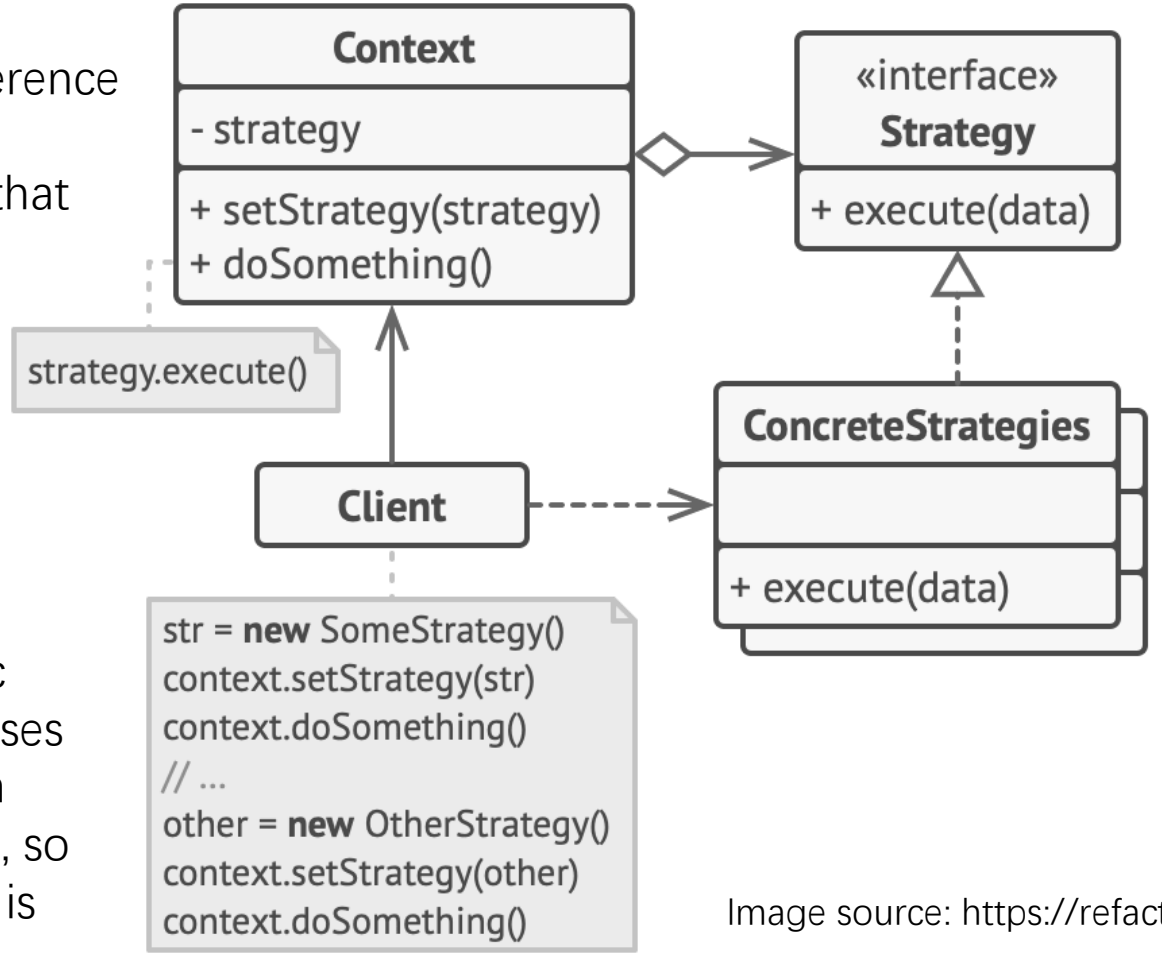
The **Strategy** interface declares a method used by the **Context** to execute a strategy/algorithm. (e.g., `validate`())

Concrete Strategies implement different variations of a strategy/algorithm the **Context** uses (e.g., `AllLowercaseStrategy`, `AllNumberStrategy`)

Image source: https://refactoring.guru/design-patterns/strategy

Strategy – The Solution

Context maintains a reference to one of the concrete strategies and executes that specific strategy (e.g., TextValidator)



The **Strategy** interface declares a method used by the **Context** to execute a strategy/algorithm. (e.g., `validate`())

Concrete Strategies implement different variations of a strategy/algorithm the **Context** uses (e.g., `AllLowercaseStrategy`, `AllNumberStrategy`)

Client creates a specific strategy object and passes it to the context using a **constructor** or a **setter**, so that the actual strategy is adapted at runtime

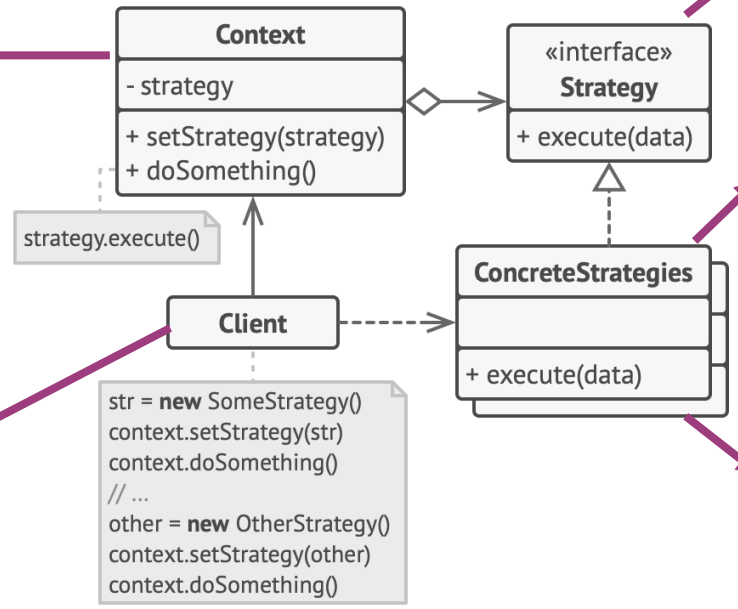
Image source: https://refactoring.guru/design-patterns/strategy

Strategy – The Solution

What is this interface?

```
interface ValidationStrategy {
    boolean execute(String s);
}
```

```
private static class Validator {
    private final ValidationStrategy validationStrategy;

    public Validator(ValidationStrategy validationStrategy) {
        this.validationStrategy = validationStrategy;
    }

    public boolean validate(String s) {
        return validationStrategy.execute(s);
    }
}
```




```
static class IsAllLowerCase implements ValidationStrategy {

    @Override
    public boolean execute(String s) {
        return s.matches("[a-z]+");
    }
}
```

```
Validator v1 = new Validator(new IsNumeric());
// false
System.out.println(v1.validate("aaaa"));
Validator v2 = new Validator(new IsAllLowerCase());
// true
System.out.println(v2.validate("bbbb"));
```

```
static class IsNumeric implements ValidationStrategy {

    @Override
    public boolean execute(String s) {
        return s.matches("\\d+");
    }
}
```

Code: https://blog.csdn.net/ryo1060732496/article/details/88831905

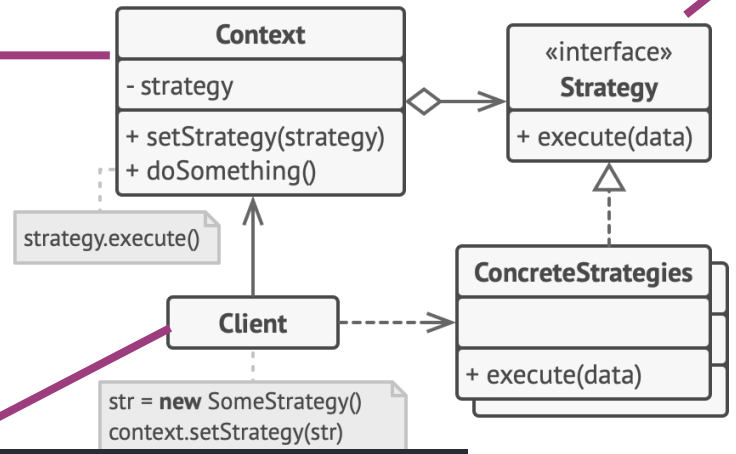
Image: https://refactoring.guru/design-patterns/strategy

Strategy – The Solution

What is this interface?

```
interface ValidationStrategy {
    boolean execute(String s);
}
```

```
private static class Validator {
    private final ValidationStrategy validationStrategy;

    public Validator(ValidationStrategy validationStrategy) {
        this.validationStrategy = validationStrategy;
    }

    public boolean validate(String s) {
        return validationStrategy.execute(s);
    }
}
```




```
Validator v3 = new Validator((String s) -> s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));
Validator v4 = new Validator((String s) -> s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```

- Since ValidationStrategy is a functional interface, we don't need to define new classes for each concrete strategy
- Could use lambda to implement new concrete strategies

Code: https://blog.csdn.net/ryo1060732496/article/details/88831905

Image: https://refactoring.guru/design-patterns/strategy

```
List<String> names = Arrays.asList("Anne", "Joe", "Harry");
Collections.sort(names, new Comparator<String>() {
  public int compare(String o1, String o2) {
    return o1.length() - o2.length();
  }
});
Assert.assertEquals(Arrays.asList("Joe", "Anne", "Harry"), names);
```

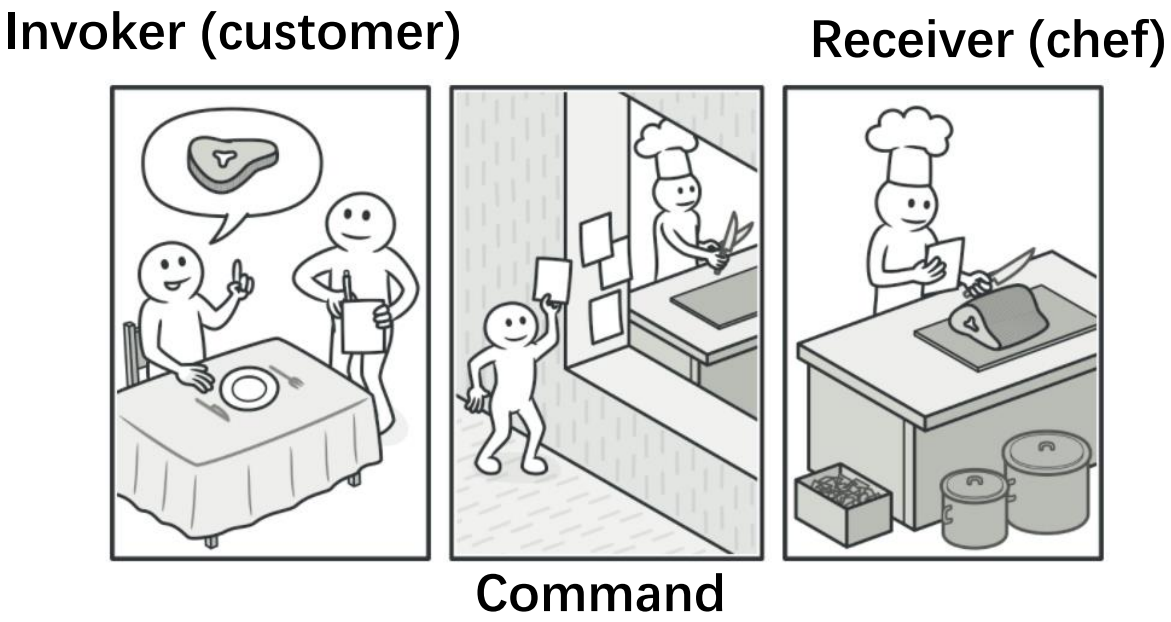
Strategy – An Example

One common usage of the strategy pattern is to define custom **sorting** strategies for Java Collections

Strategy Summary

- The strategy design pattern defines a group of algorithms. One of the algorithms is selected for use at runtime
- The functional interface specifies the general algorithm structure while the lambda function implements the details at runtime.

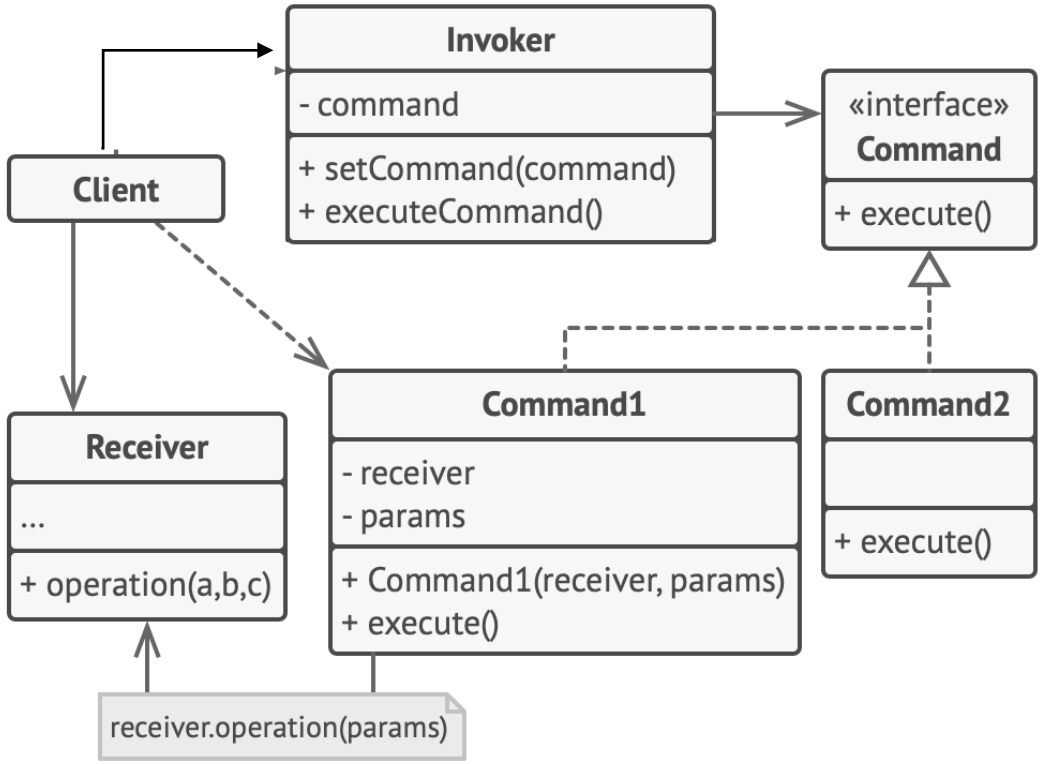
Command Design Pattern



https://refactoring.guru/design-patterns/command

- A behavioral design pattern that encapsulate a request into a stand-alone object that contains all information about the request
- Benefits: Decoupling Invokers (命令调用者) and Receivers (命令执行者)

https://refactoring.guru/design-patterns/command
https://dzone.com/articles/design-patterns-command




```
//Receiver
public class Light{
  private boolean on;
  public void switchOn(){
    on = true;
  }
  public void switchOff(){
    on = false;
  }
}
```

```
//Command
public interface Command{
  public void execute();
}
```

```
//Invoker
public class RemoteControl{
  private Command command;
  public void setCommand(Command command){
    this.command = command;
  }
  public void pressButton(){
    command.execute();
  }
}
```

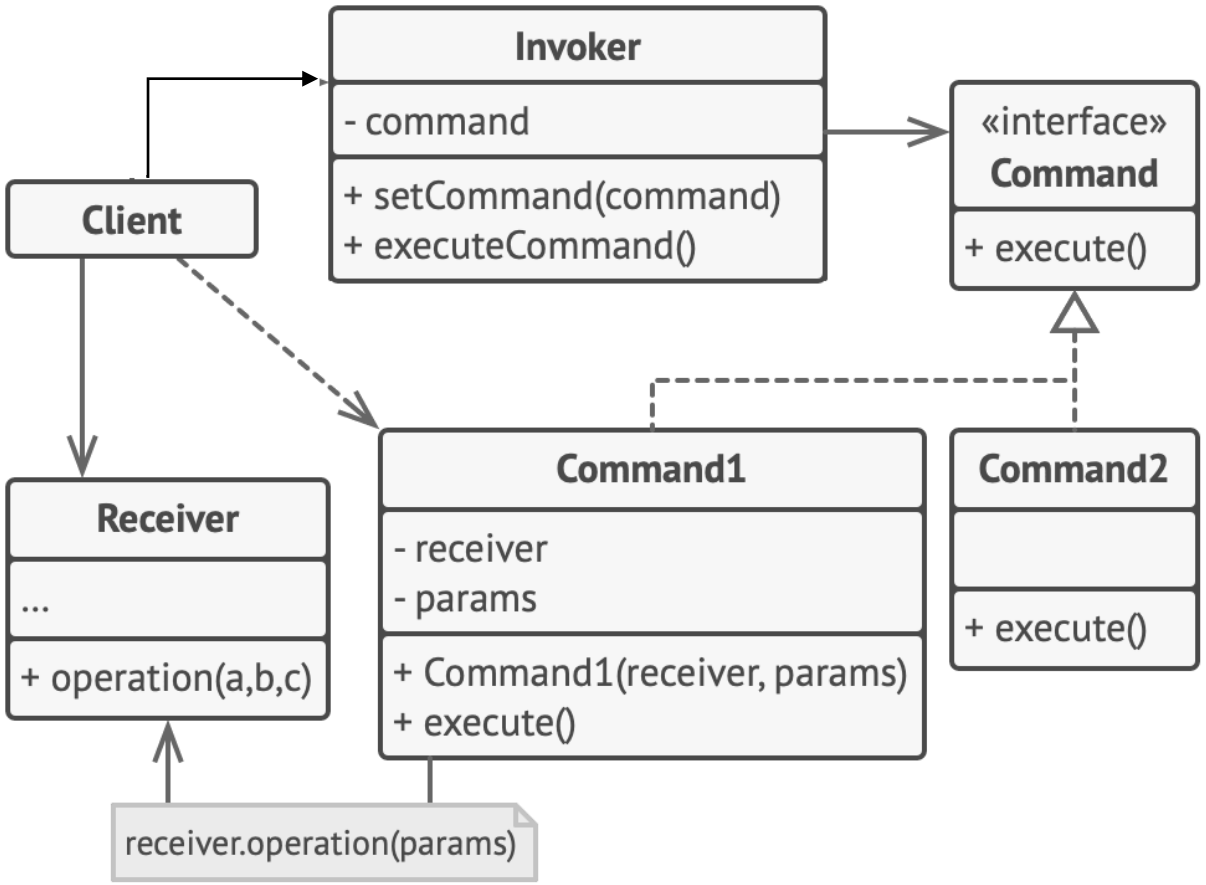
```
//Concrete Command
public class LightOnCommand implements Command{
  //reference to the light
  Light light;
  public LightOnCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOn();
  }
}
```

```
//Concrete Command
public class LightOffCommand implements Command{
  //reference to the light
  Light light;
  public LightOffCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOff();
  }
}
```

Command Design Pattern

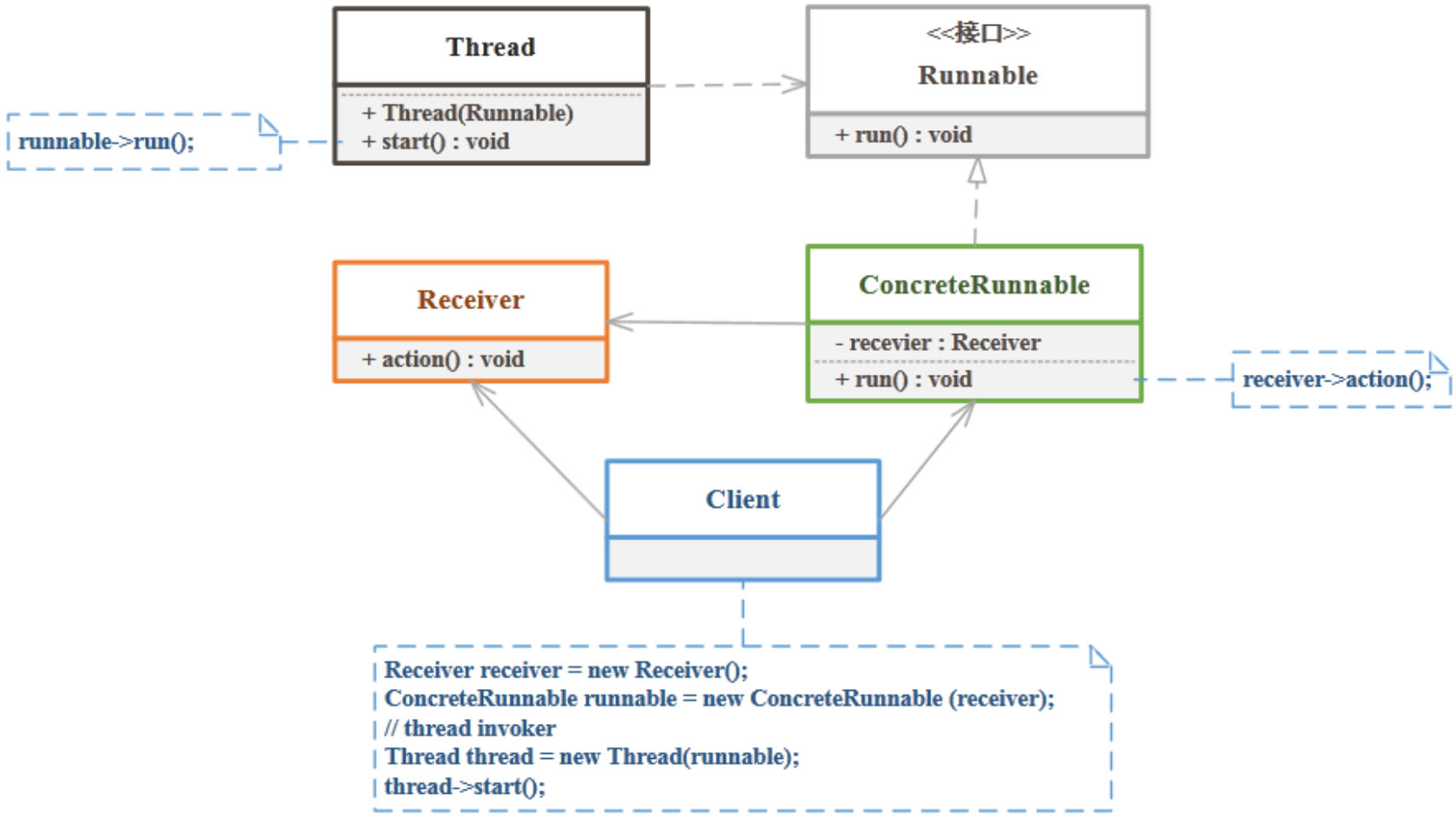
Command Design Pattern

https://refactoring.guru/design-patterns/command
https://dzone.com/articles/design-patterns-command




```
//Client
public class Client{
  public static void main(String[] args)     {
    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightsOnCommand(light);
    Command lightsOff = new LightsOffCommand(light);

    //switch on
    control.setCommand(lightsOn);
    control.pressButton();

    //switch off
    control.setCommand(lightsOff);
    control.pressButton();
  }
}
```

Command Design Pattern: Example



https://www.alicharles.com/article/design-pattern/jdk-command-pattern/

Next Lecture

- Course Review