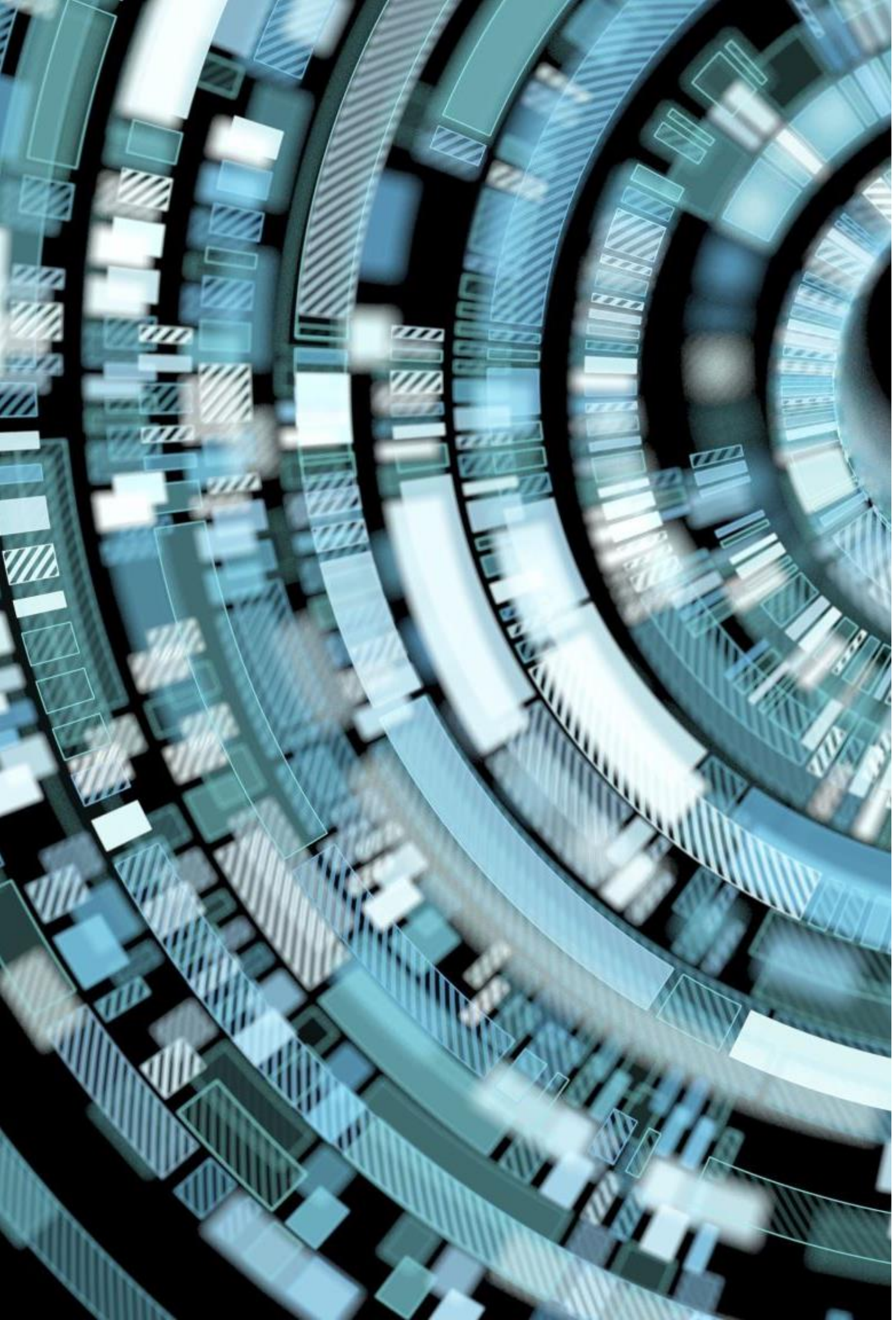
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 6

- Design Patterns
  - Overview
  - Classification
  - Intro to 3 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**

Structural Patterns

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

Behavioral Patterns

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

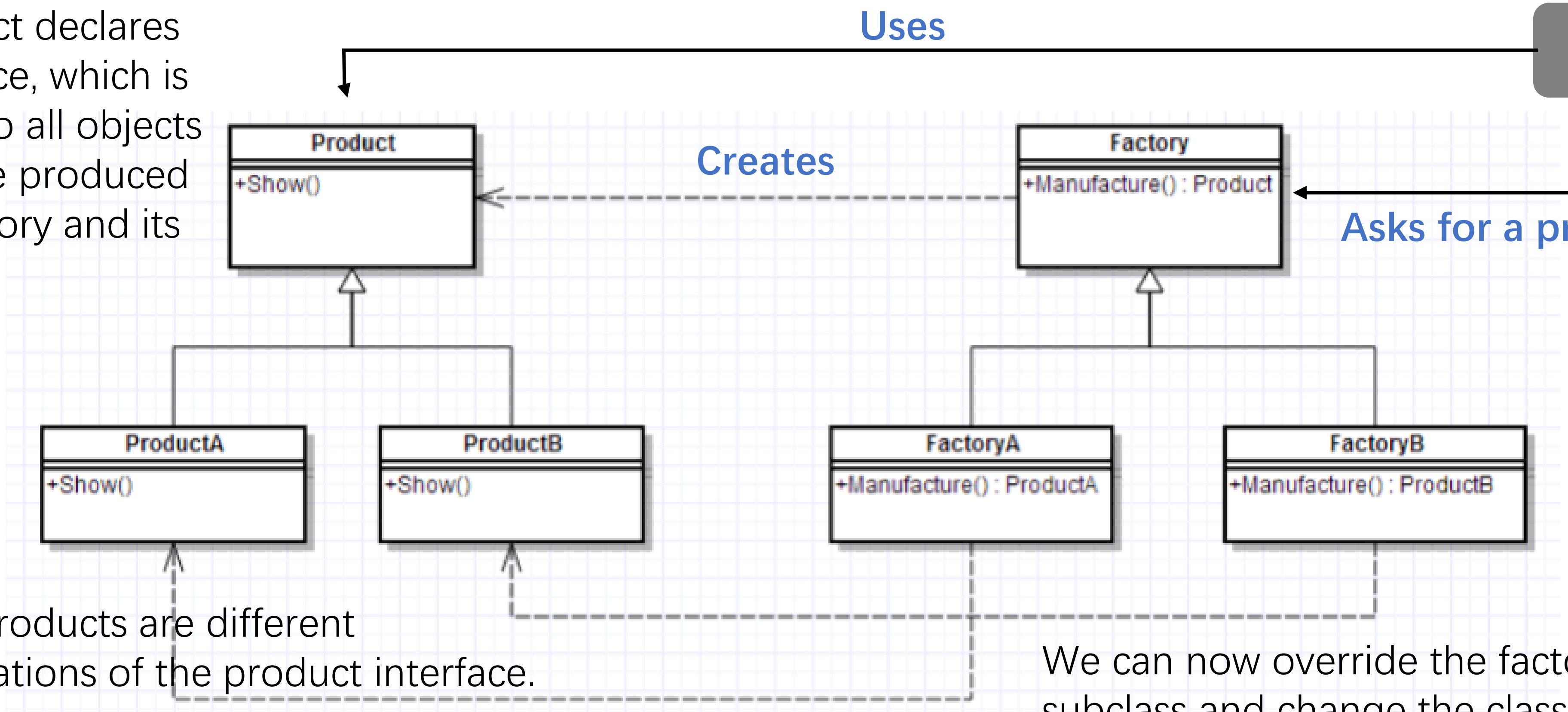
Factory Method – The Problem

- Imagine you own a factory, which creates different products; Each product has its own advertisement
- A possible OO program design
  - Use a giant Factory class, which has different methods for creating different Products and different Ads for each product
  - Adding new Products would require making (similar) changes to the entire codebase
  - Application is filled with conditionals that switch the behavior depending on the class of Product objects
  - Code is hard to manage, error-prone, hard to scale

Factory Method – The Solution

Idea: separating the creation of objects and usages of objects

The Product declares the interface, which is common to all objects that can be produced by the factory and its subclasses.



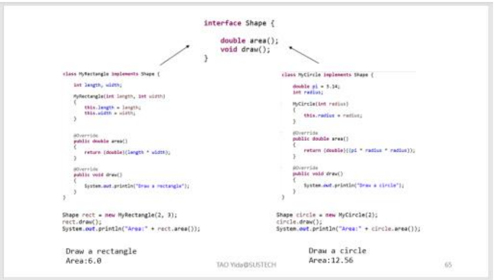
Replace direct object construction calls (using the new operator) with calls to a special factory method

Concrete Products are different implementations of the product interface.

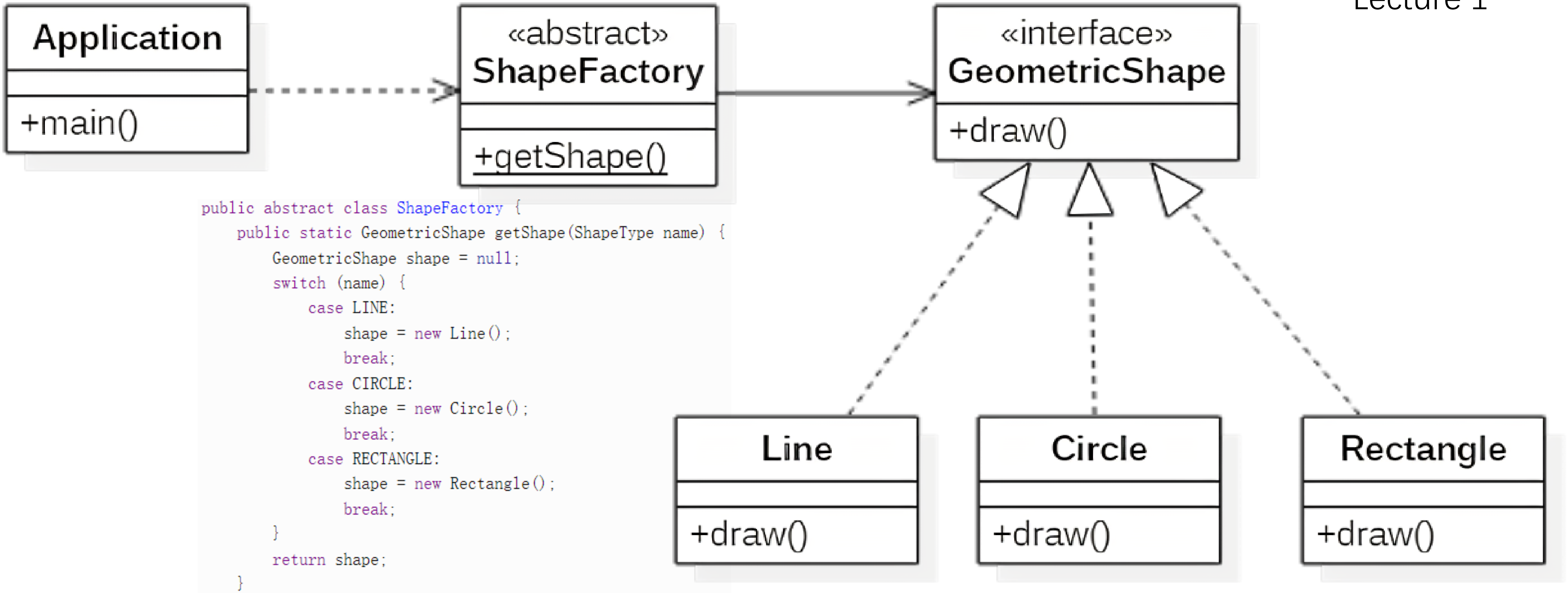
We can now override the factory method in a subclass and change the class of products being created by the factory method.

https://blog.csdn.net/carson_ho/article/details/52343584

Factory Method – Example I



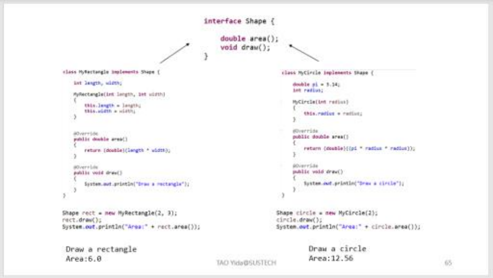
Lecture 1



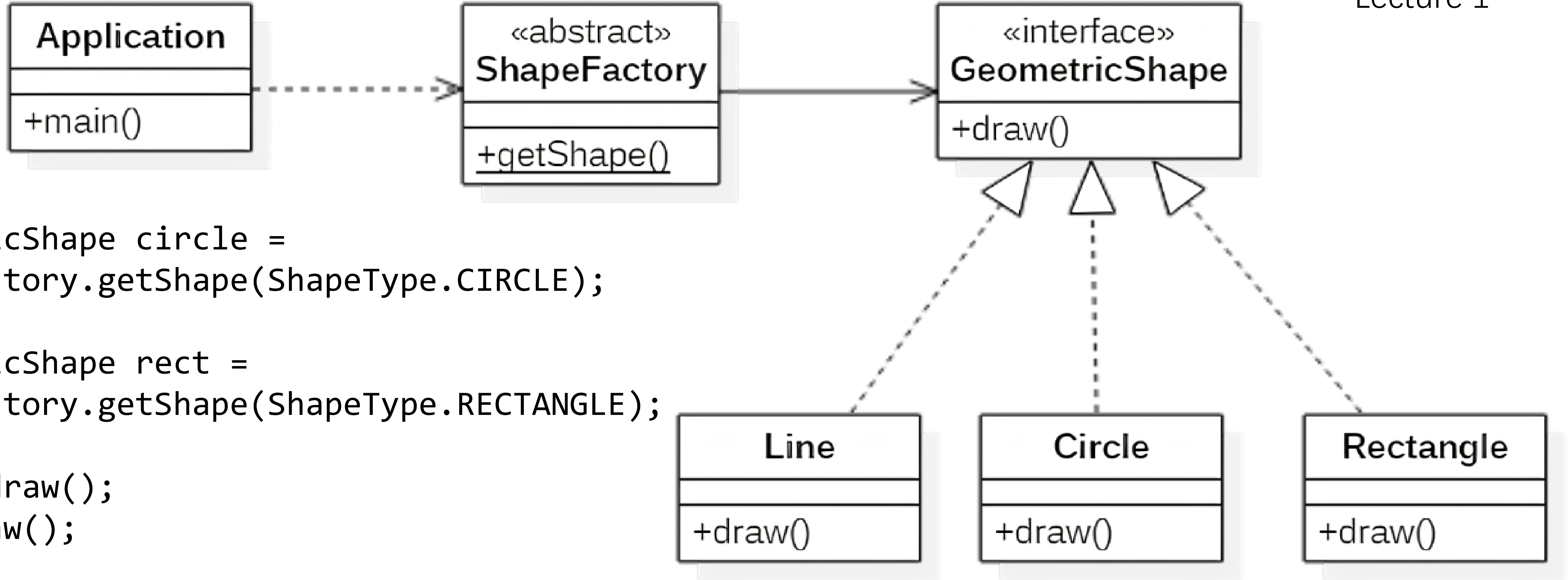

```java
public abstract class ShapeFactory {
    public static GeometricShape getShape(ShapeType name) {
        GeometricShape shape = null;
        switch (name) {
            case LINE:
                shape = new Line();
                break;
            case CIRCLE:
                shape = new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

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

Factory Method – Example I



Lecture 1

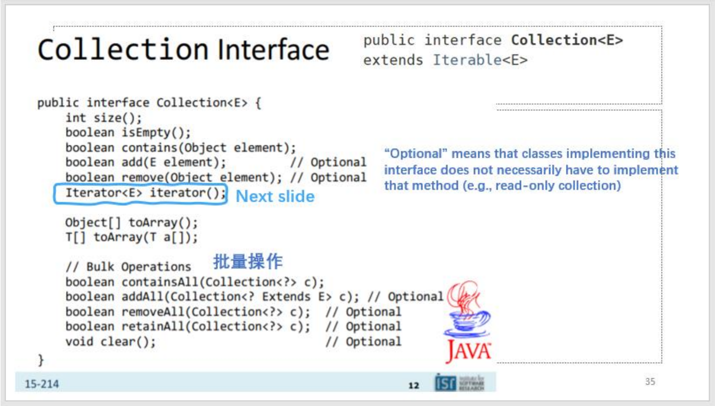



```
GeometricShape circle =
ShapeFactory.getShape(ShapeType.CIRCLE);

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

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

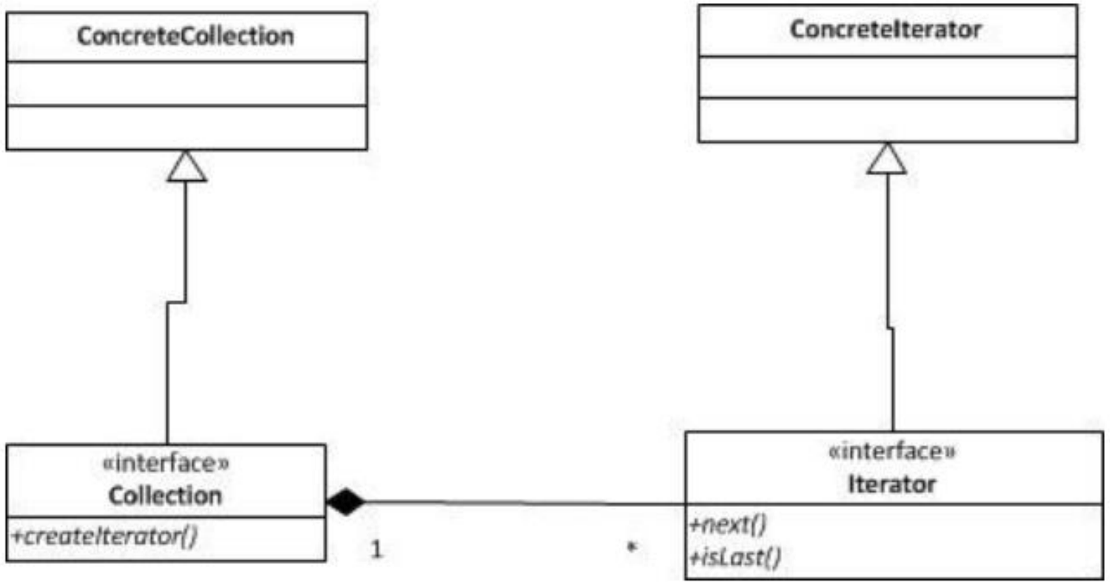
Full example: https://dzone.com/articles/factory-method-design-pattern

Factory Method – Example II

Lecture 3



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

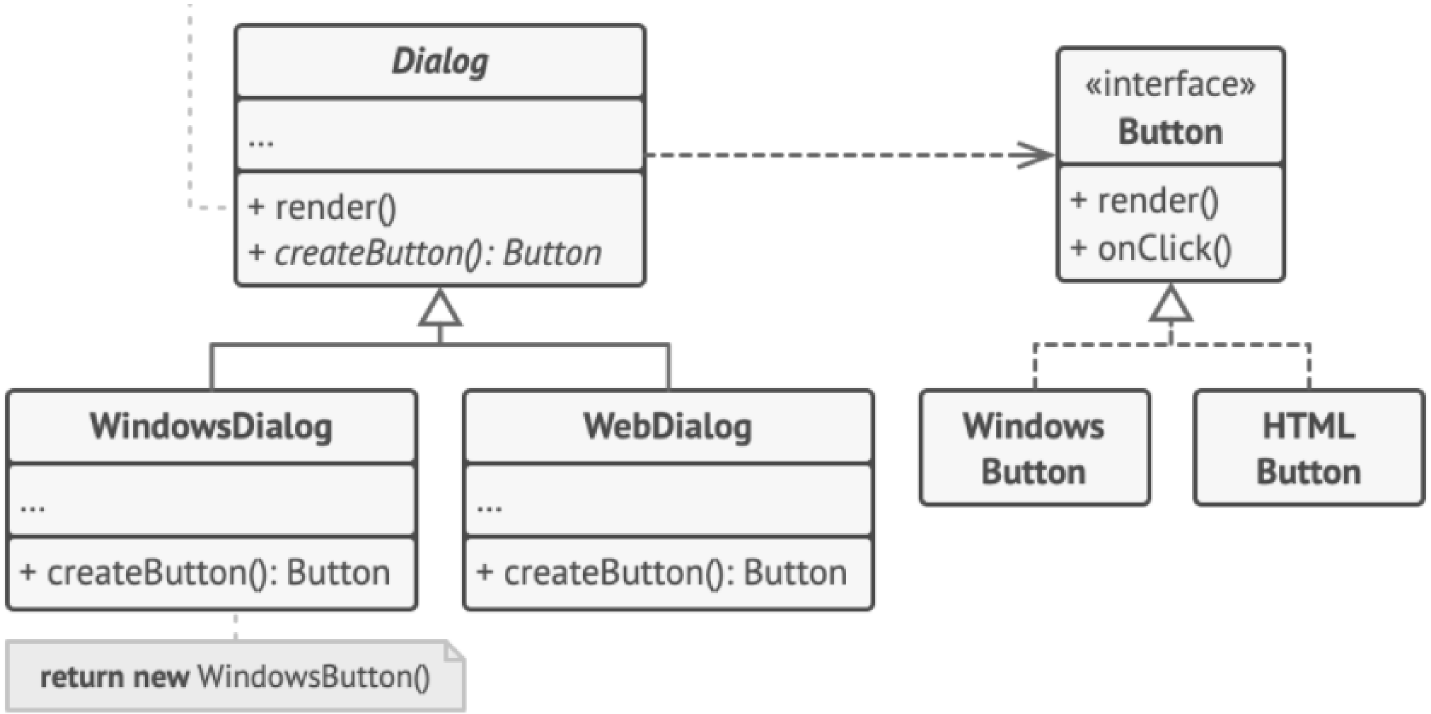



```
List<String> list = new ArrayList<String>( );
list.add( "dog" );
...
// create an iterator
Iterator<String> iter = list.iterator( );
while ( iter.hasNext() ) System.out.println( iter.next() );
```

Collection is the factory/creator, which creates the Iterator

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

Factory Method – Example III



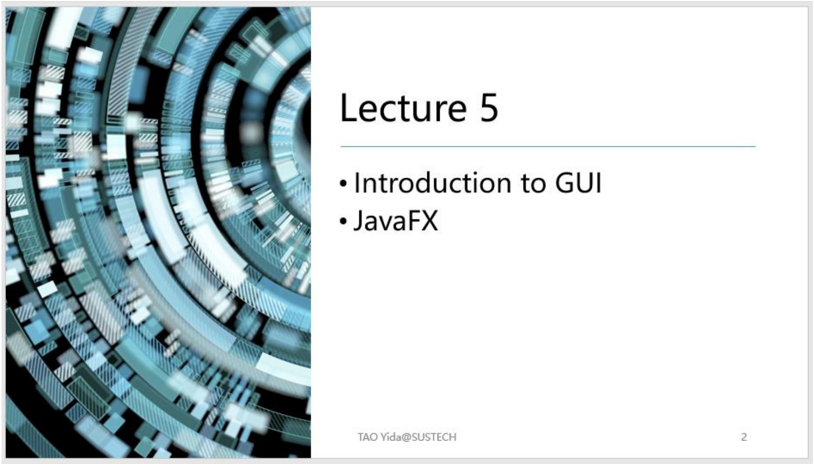
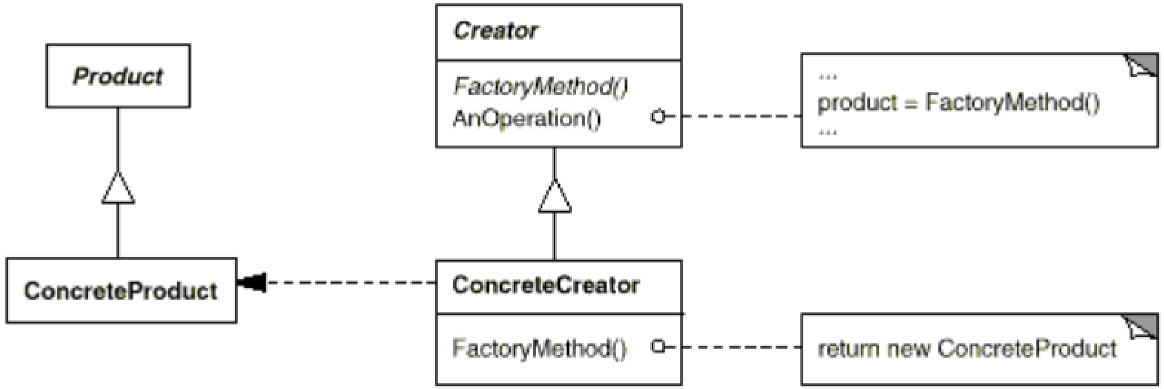


Image: https://refactoring.guru/design-patterns/factory-method

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.



Factory Method - Summary

Pros

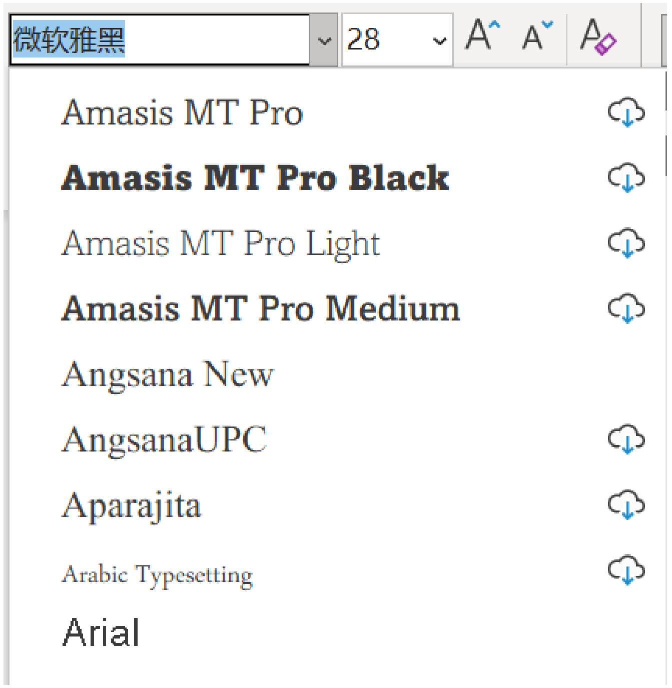
- High extensibility: it is more flexible in adding new types.
- Good testability: each component can be tested individually

Cons

- Large number of required classes

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)……

Decorator – The Problem

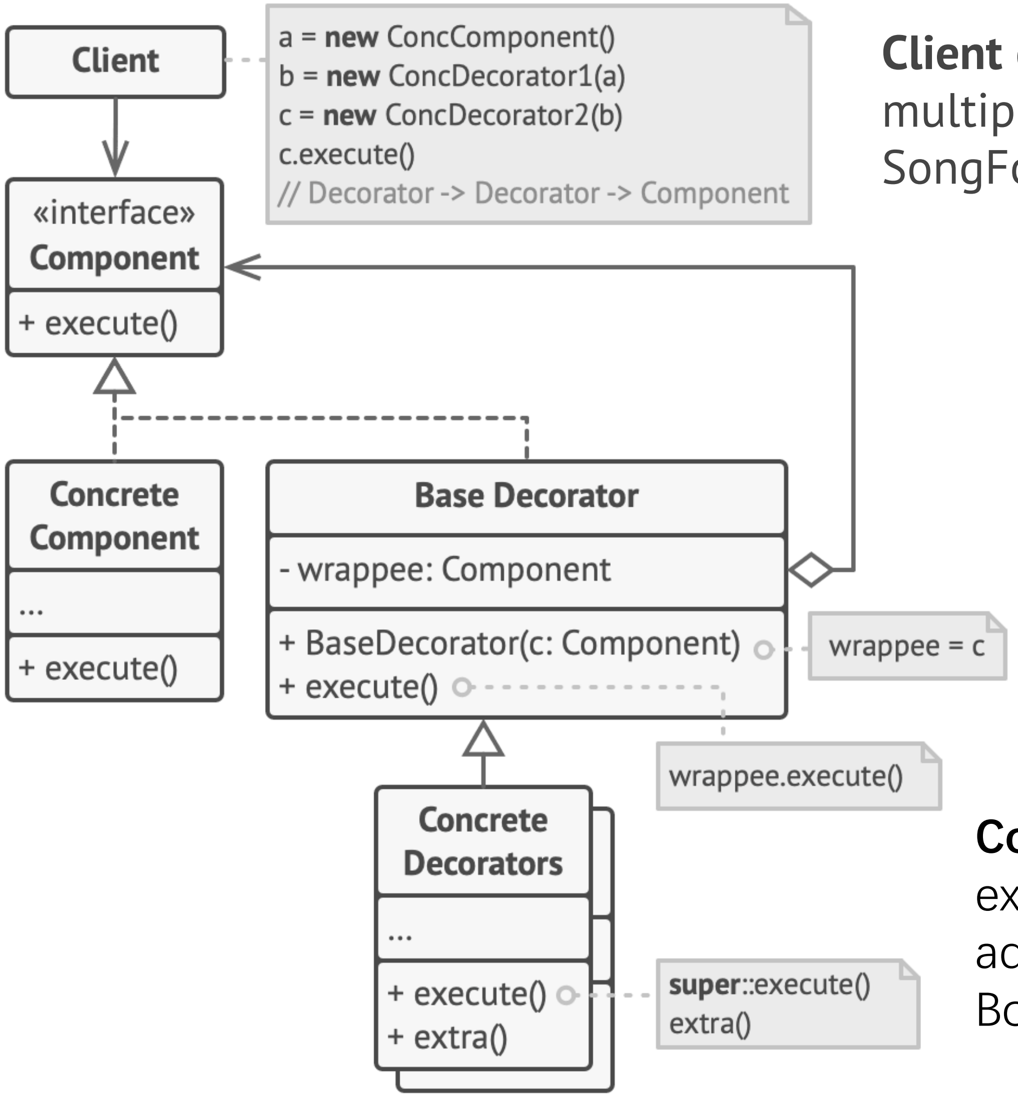
- Can we design one class for each possibility?
- Class explosion problem: n styles, 2^n-1 combinations
- Inefficient code reuse

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)
……………………………………………………

Decorator – The Solution

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())



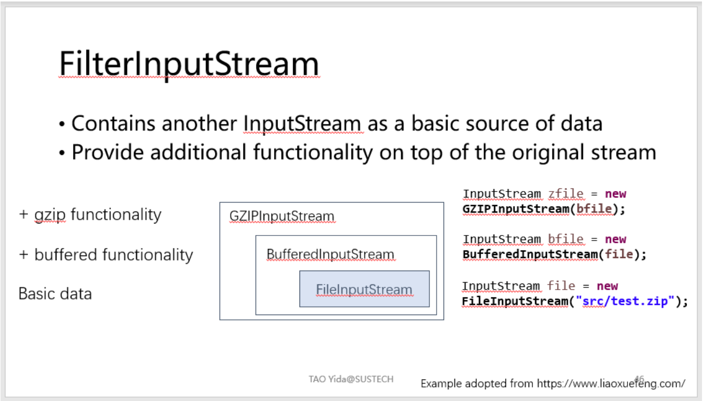
Client can wrap components in multiple layers of decorators (e.g., SongFont + Bold + Italian)

Base Decorator class has a field with the type **Component**; could be constructed with a **Component** (e.g., Font Style)

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


FilterInputStream

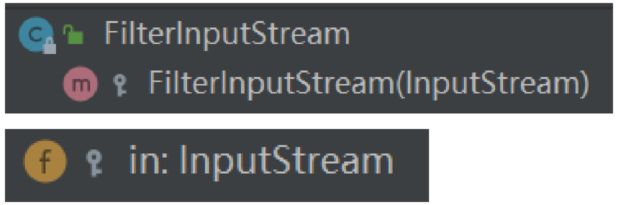
- Contains another InputStream as a basic source of data
- Provide additional functionality on top of the original stream

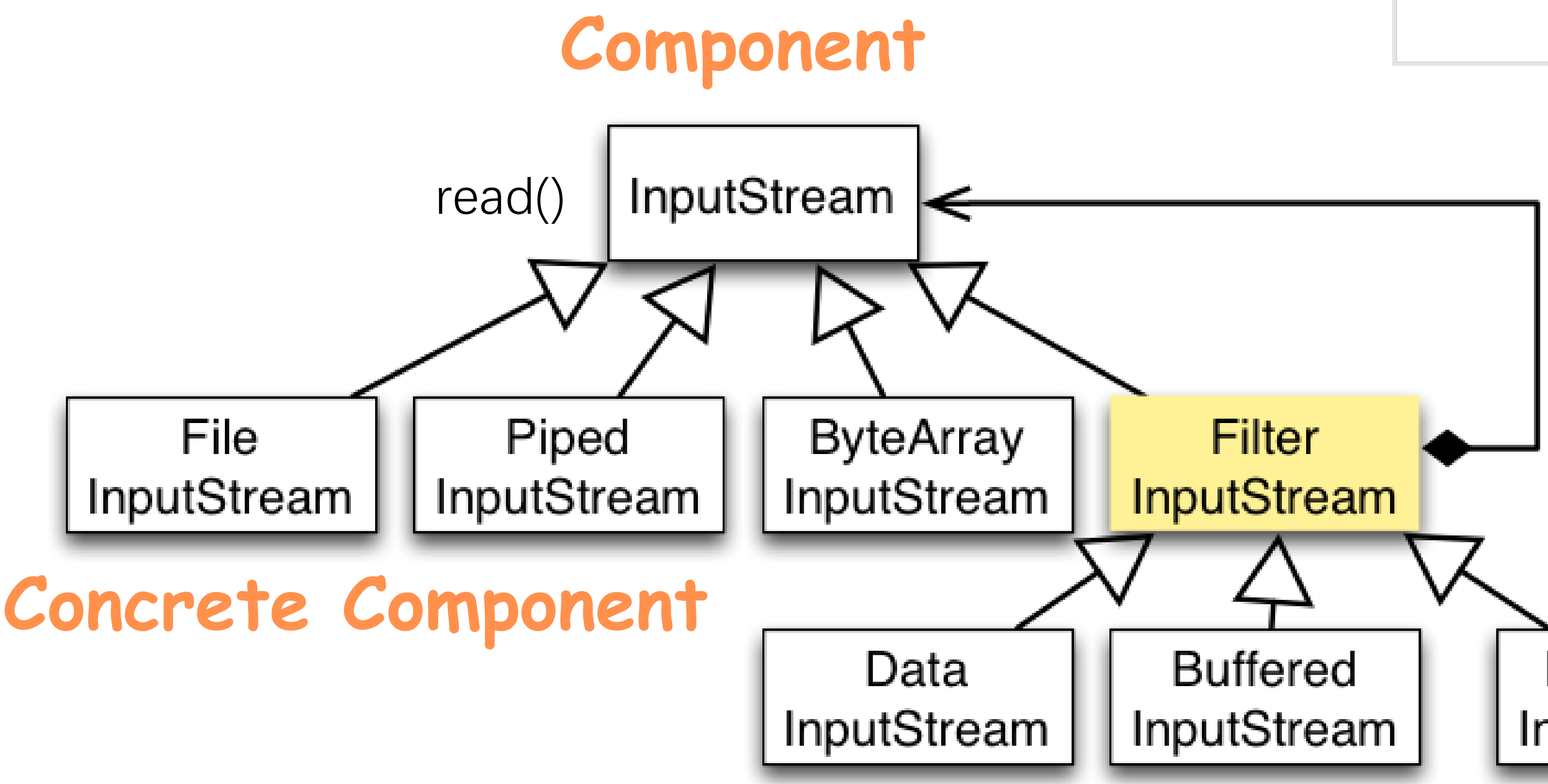

```java
InputStream zfile = new GZIPInputStream(bfile);
InputStream bfile = new BufferedInputStream(file);
InputStream file = new FileInputStream("src/test.zip");
```

TAO Yida@SUSTECH

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

Lecture 2





Base Decorator

Concrete Decorator

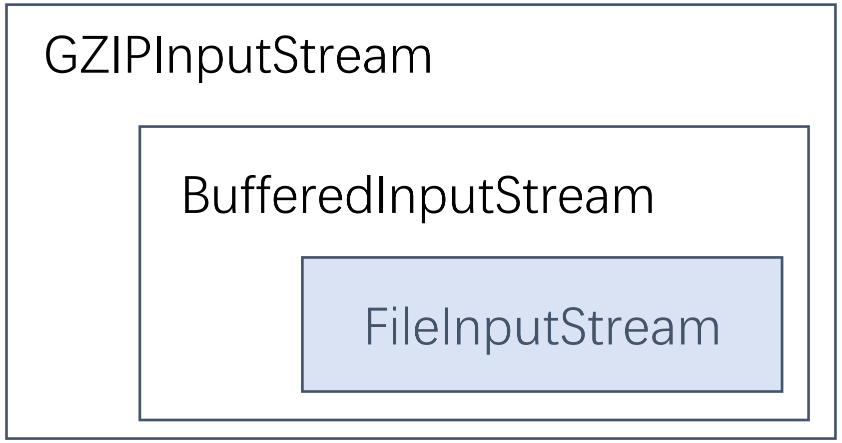
Recall: FilterInputStream

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

\+ gzip functionality

\+ buffered functionality

Basic data




```java
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 by placing these objects inside special wrapper objects that contain the new behaviors.

Strategy – The Problem

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

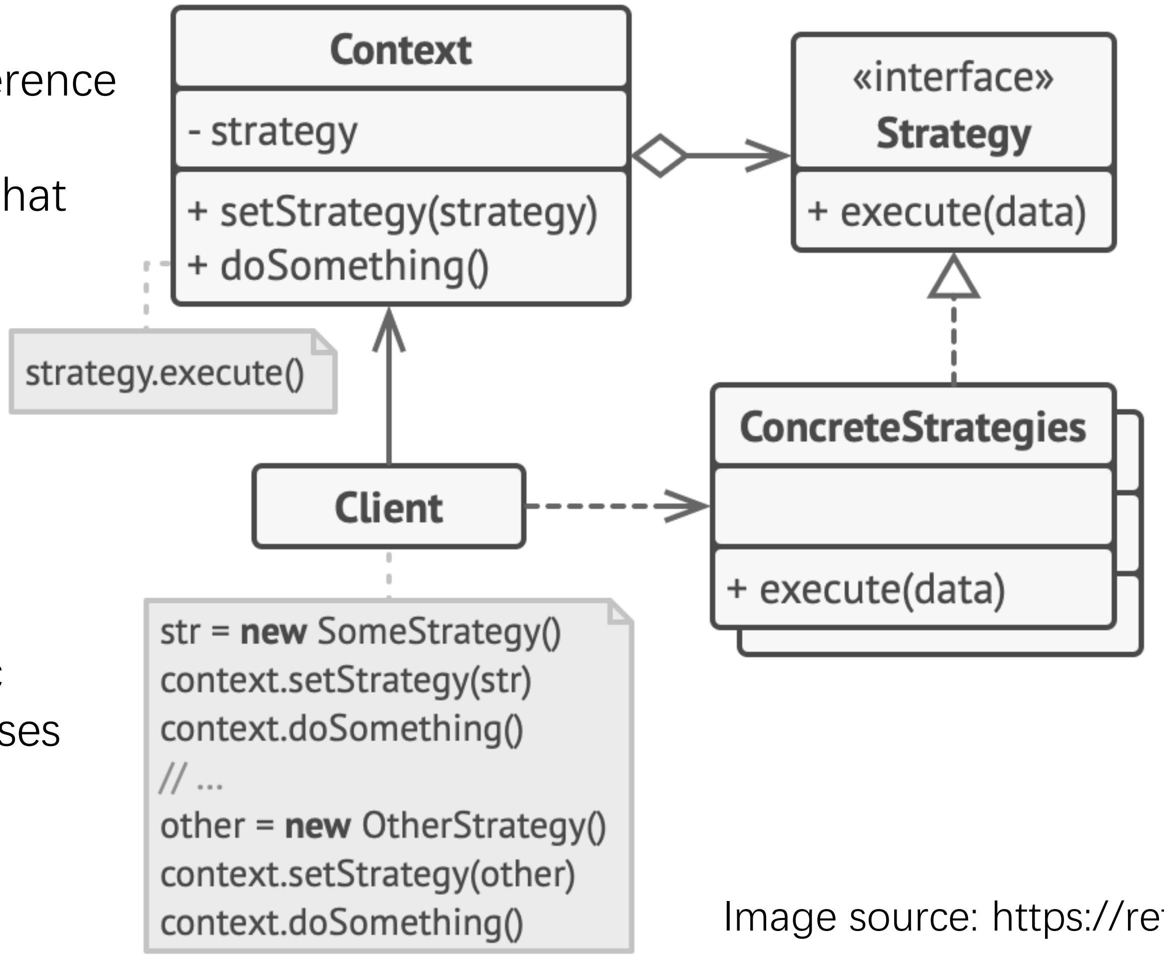
- Strategy 1: text must be all lowercased
- Strategy 2: text must be all numbers
- Strategy 3: text must contain both numbers and letters
- ……

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

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** execute a strategy. (e.g., validate())

Concrete Strategies implement different variations of a strategy 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.

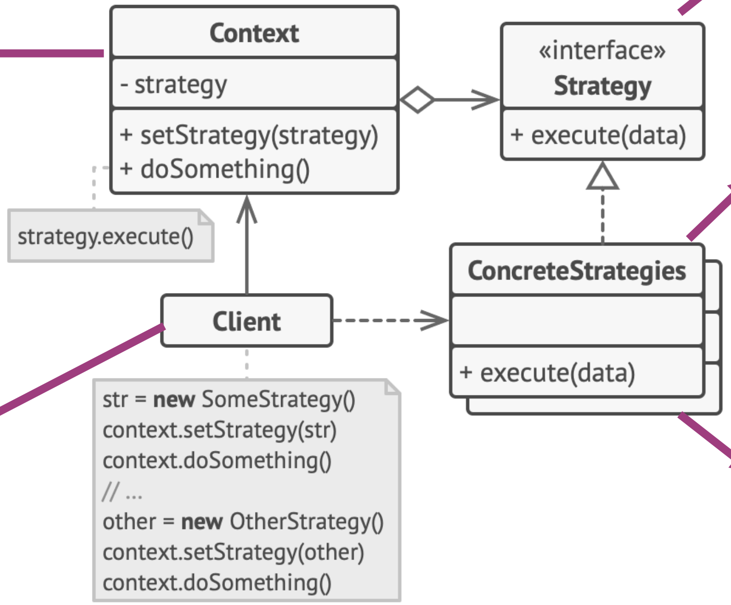
Image source: https://refactoring.guru/design-patterns/strategy

Strategy – The Solution

What is this interface?

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

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

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

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




```java
static class IsAllLowerCase implements ValidationStrategy {

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

```java
static class IsNumeric implements ValidationStrategy {

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

```java
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"));
```

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

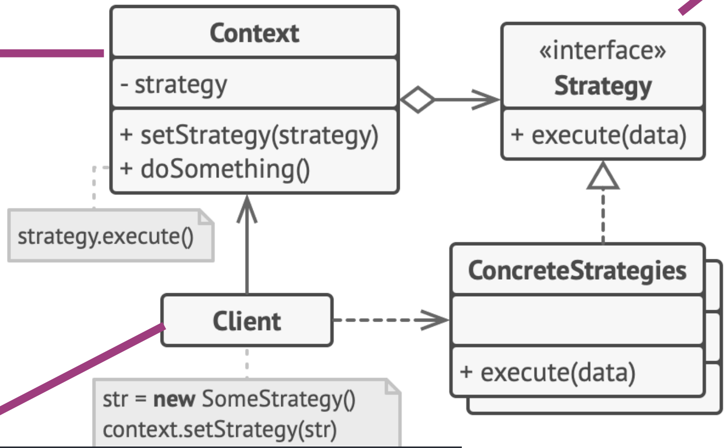
Image: https://refactoring.guru/design-patterns/strategy

Strategy – The Solution

What is this interface?

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

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

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

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




```java
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

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.

Next Lecture

- Reusable Software
- Web Scraping Libraries
- RESTful APIs