Design Patterns

Course	Design Patterns
Trainer	Tin Bui
Designed by	Tin Bui
Last updated	Apr. 09, 2012

Contents

- What is Design Patterns?
- Adapter
- Singleton
- Factory method
- Abstract Factory
- Decorator
- Exercises

Course Objectives

- Understand What Design Patterns is
- Understand the most basic patterns
- Apply Design Pattern in programming

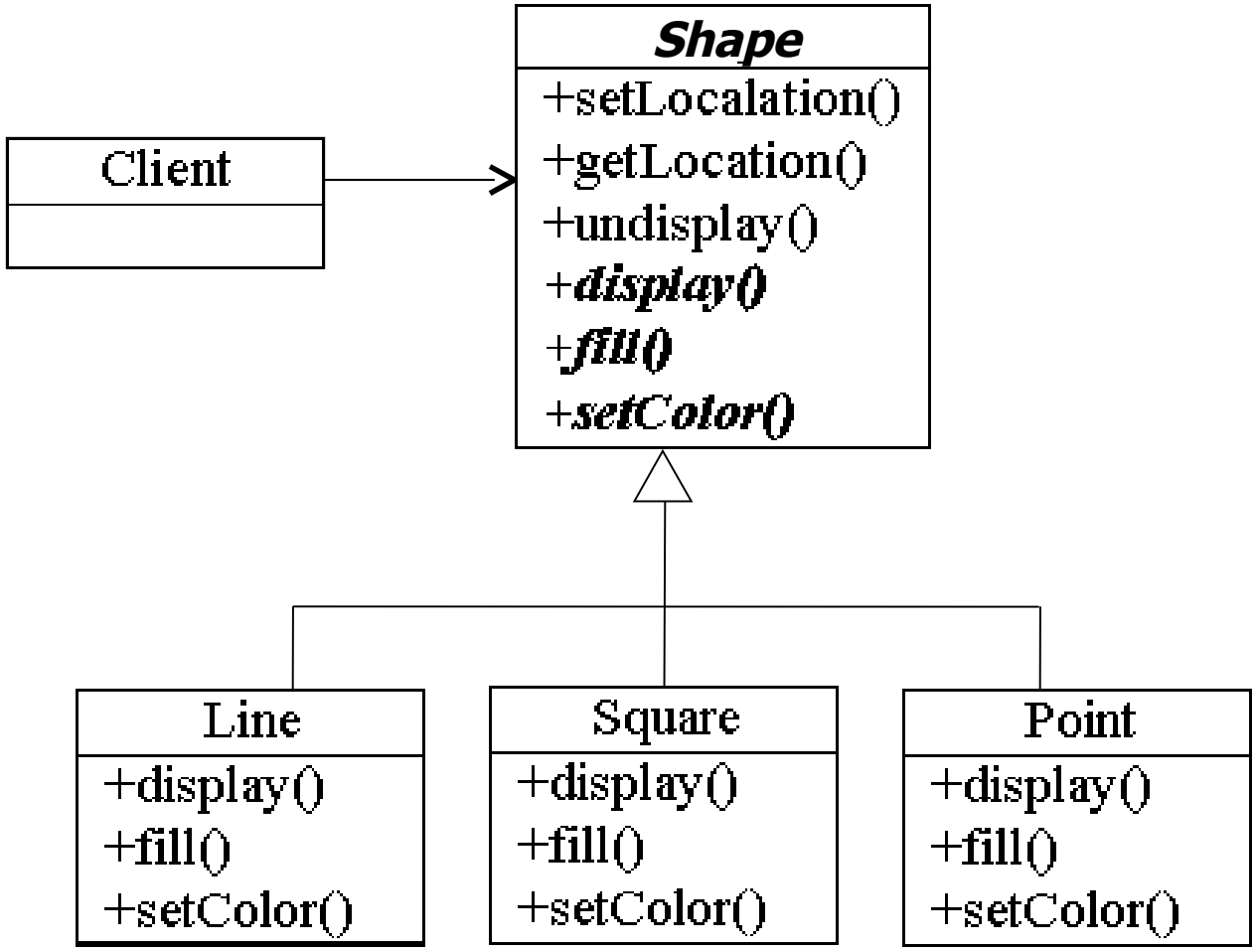
Why trainees should learn this topic?

- Reuse solutions
- Establish common terminology
- Patterns give you a higher-level perspective on the problem and on the process of design

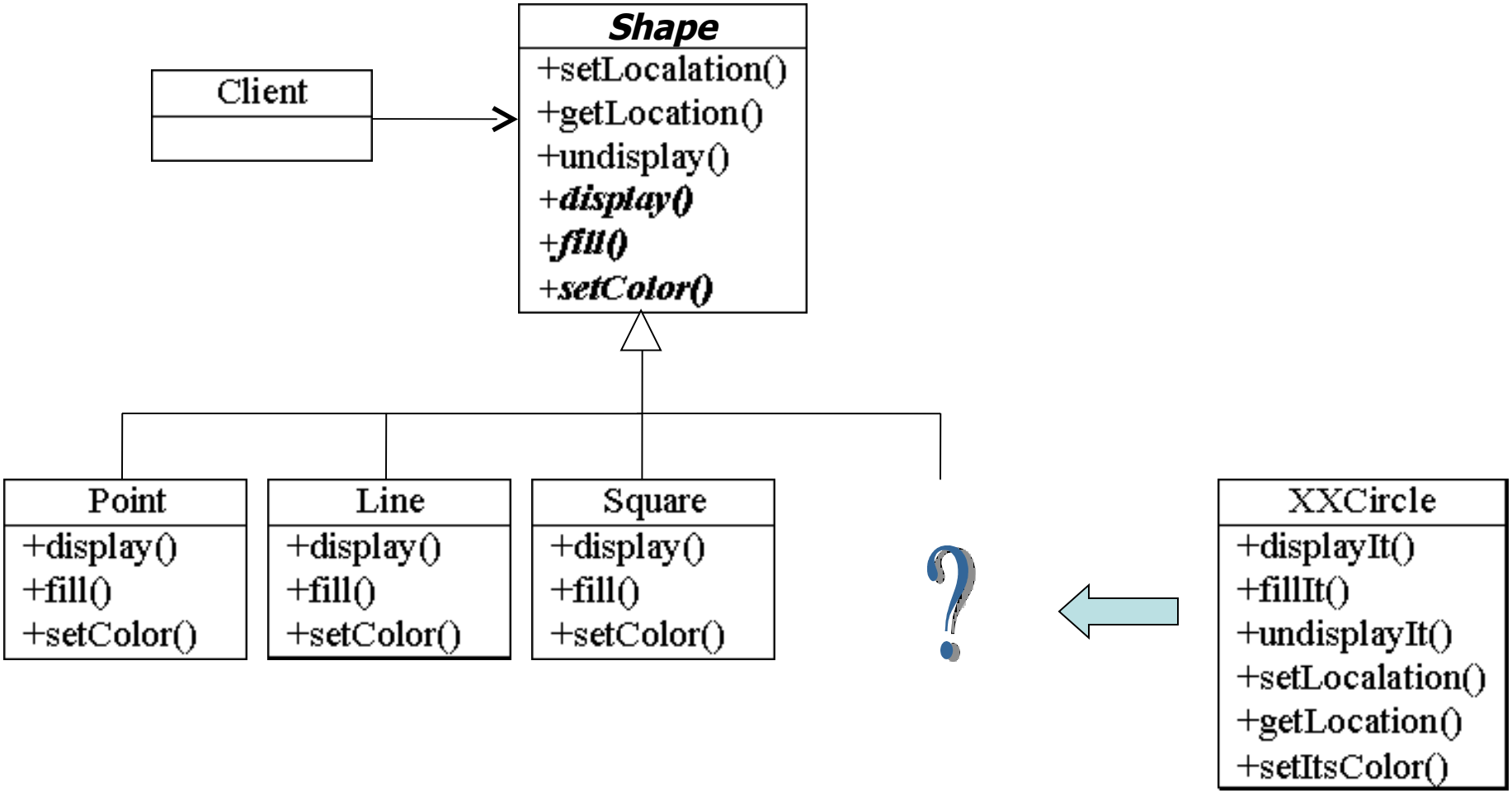
Contents

- **What is Design Patterns?**
- Adapter
- Singleton
- Factory method
- Abstract Factory
- Decorator
- Exercises

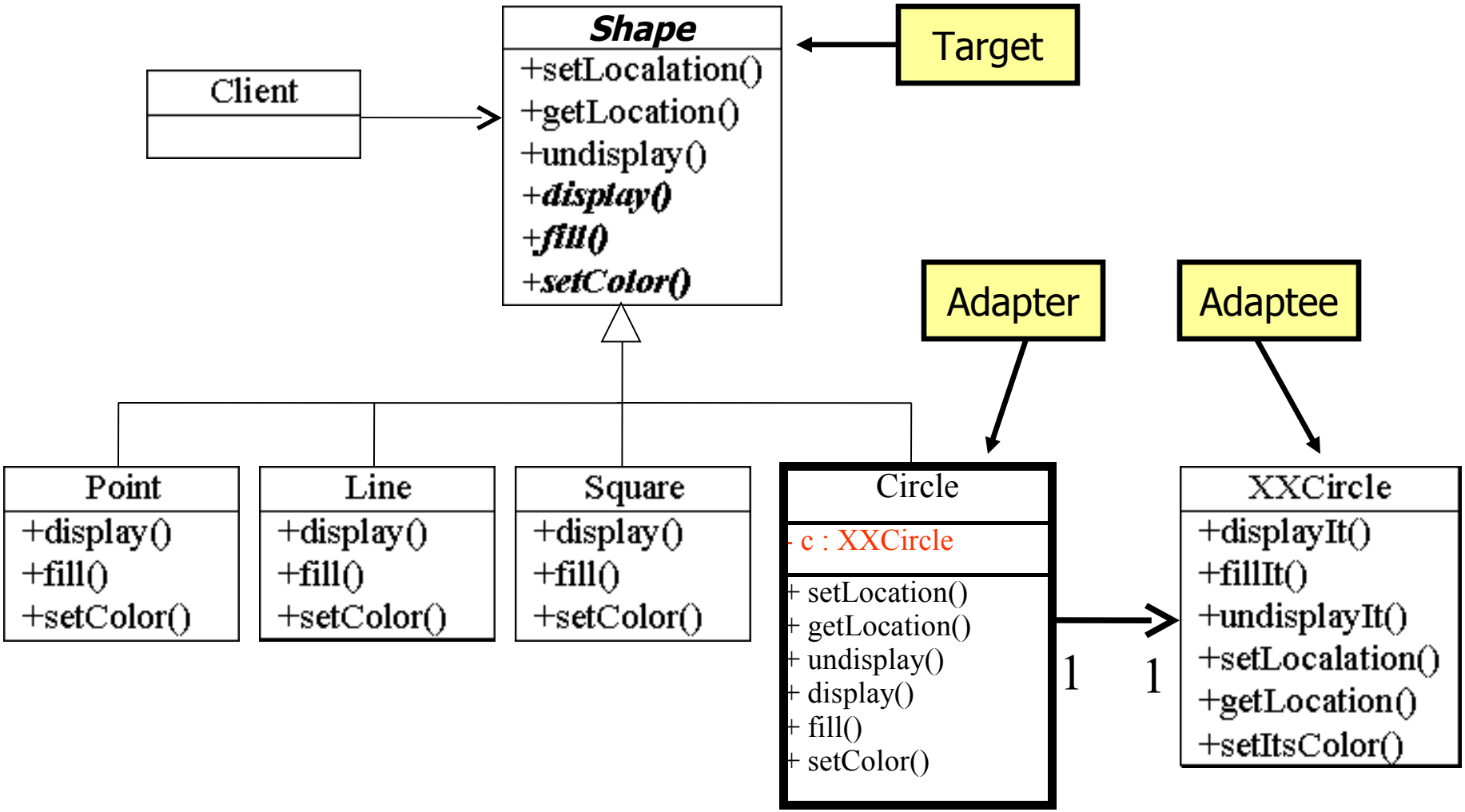
An Example


Shape
+setLocalation()
+getLocation()
+undisplay()
+display()
+fill()
+setColor()
Client
Line
+display()
+fill()
+setColor()
Square
+display()
+fill()
+setColor()
Point
+display()
+fill()
+setColor()

An Example (cont.)



An Example (cont.)



What is Design Pattern?

- *A design pattern offers guidelines on when, how, and why an implementation can be created to solve a general problem in a particular context.*

- *"Design patterns help you learn from others' successes instead of your own failures."*
  From Mark Johnson.

History of Design Patterns

- Gamma, Helm, Vlissides, and Johnson
  ... the Gang of Four (GOF)

- In 1995, published their now famous book titled "*Design Patterns, Elements of Reusable Object-Oriented Software*"
- GOF cataloged software design patterns and brought these concepts into the mainstream

Design Patterns

- Solutions to commonly occurring design problems
- Help create flexible and reusable designs
- Isolate changes in your code
- Larger scope than a single class or algorithm
- Describes the circumstances in which the pattern is applicable
- Language Independent

Essential Properties of a Pattern

- Pattern Name
- Motivation
  - Problem
    - Describes a situation
  - Solution
    - Describes the elements that make up the design
    - Their relationships, responsibilities, and collaborations
- Applicability
  - When is the pattern applied?
- Consequences
  - The costs and benefits of applying the pattern

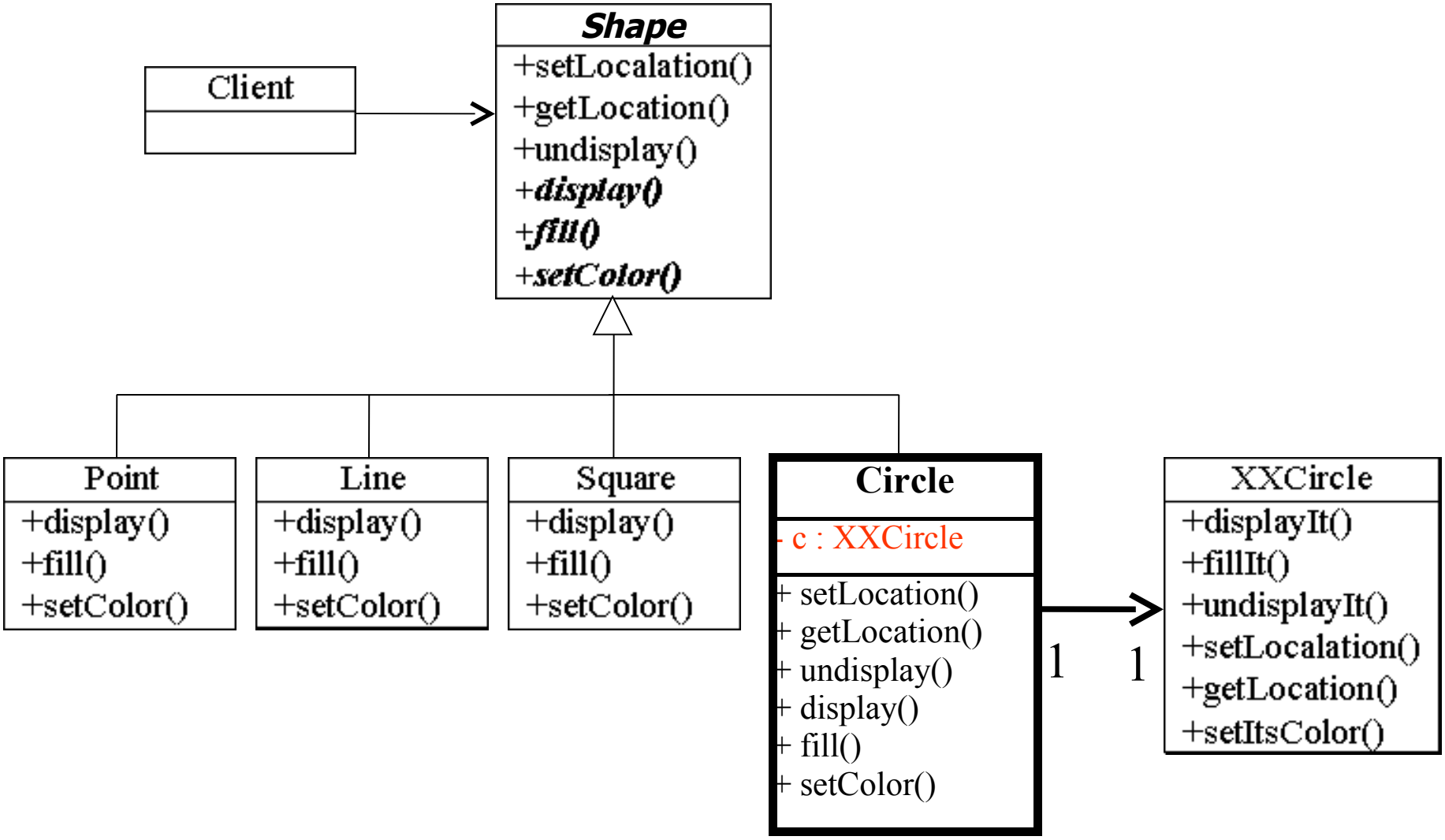
How Can Patterns Be Applied?

- A program can apply many patterns
- A pattern can be applied in many different parts
- Many patterns can be used together

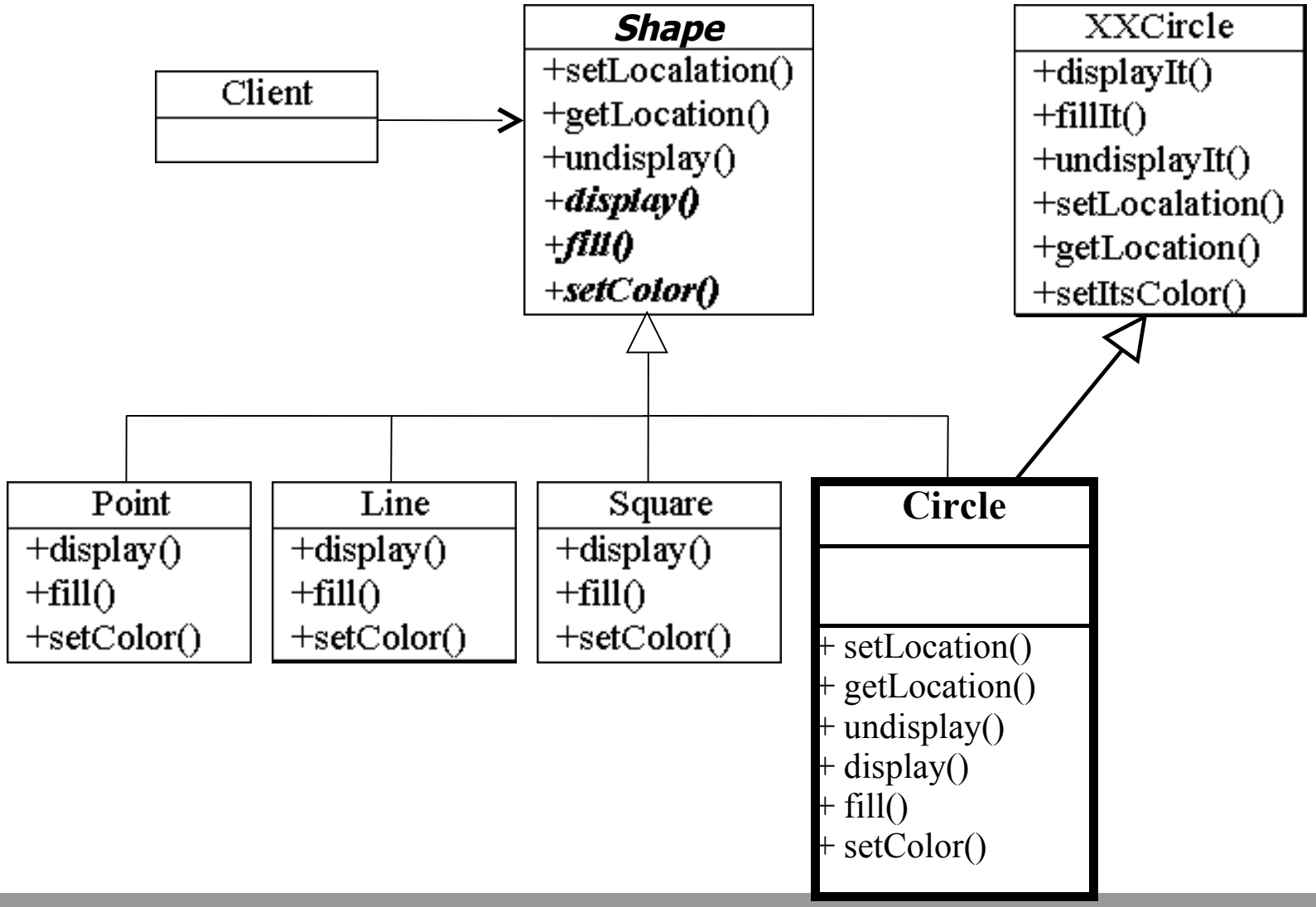
Design Pattern Categories (cont.)

Relationship	Purpose: Creational	Purpose: Structural	Purpose: Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Object Pattern Example



Class Pattern Example



Frequency of Use

- Creational Patterns

Pattern	Description	Freq
Abstract Factory	Creates an instance of several families of classes	5
Builder	Separates object construction from its representation	2
Factory Method	Creates an instance of several derived classes	5
Prototype	A fully initialized instance to be copied or cloned	3
Singleton	A class of which only a single instance can exist	4

Frequency of Use

- Structural Patterns

Pattern	Description	Freq
Adapter	Match interfaces of different classes	4
Bridge	Separates an object's interface from its implementation	3
Composite	A tree structure of simple and composite objects	4
Decorator	Add responsibilities to objects dynamically	3
Facade	A single class that represents an entire subsystem	5
Flyweight	A fine-grained instance used for efficient sharing	1
Proxy	An object representing another object	4

Frequency of Use

- Behavioral Patterns

Pattern	Description	Freq
Chain of Resp.	A way of passing a request between a chain of objects	2
Command	Encapsulate a command request as an object	4
Interpreter	A way to include language elements in a program	1
Iterator	Sequentially access the elements of a collection	5
Mediator	Defines simplified communication between classes	2
Memento	Capture and restore an object's internal state	1

Frequency of Use

- Behavioral Patterns (cont.)

Pattern	Description	Freq
Observer	A way of notifying change to a number of classes	5
State	Alter an object's behavior when its state changes	3
Strategy	Encapsulates an algorithm inside a class	4
Template Method	Defer the exact steps of an algorithm to a subclass	3
Visitor	Defines a new operation to a class without change	1

Contents

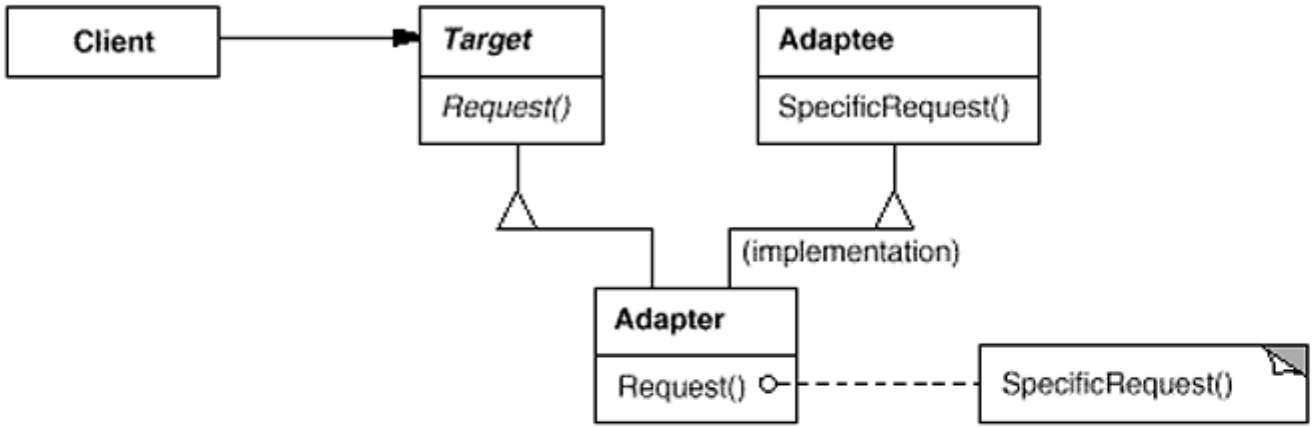
- What is Design Patterns?
- **Adapter**
- Singleton
- Factory method
- Abstract Factory
- Decorator
- Exercises

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect

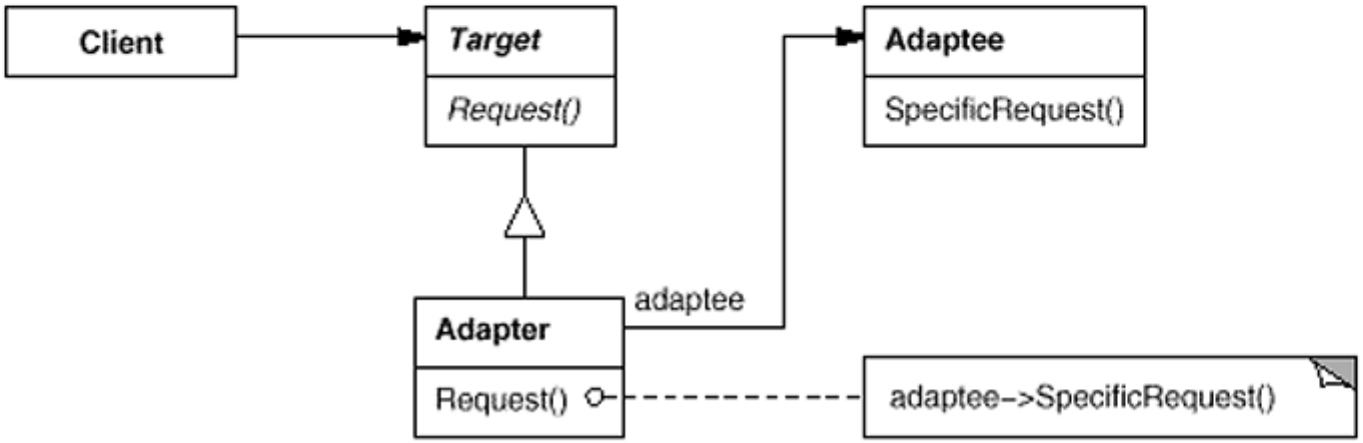
Adapter

- Structure
  - Class Adapter



Adapter

- Structure
  - Object Adapter



Adapter

- Applicability
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces

Contents

- What is Design Patterns?
- Adapter
- **Singleton**
- Factory method
- Abstract Factory
- Decorator
- Exercises

Singleton

- Intent

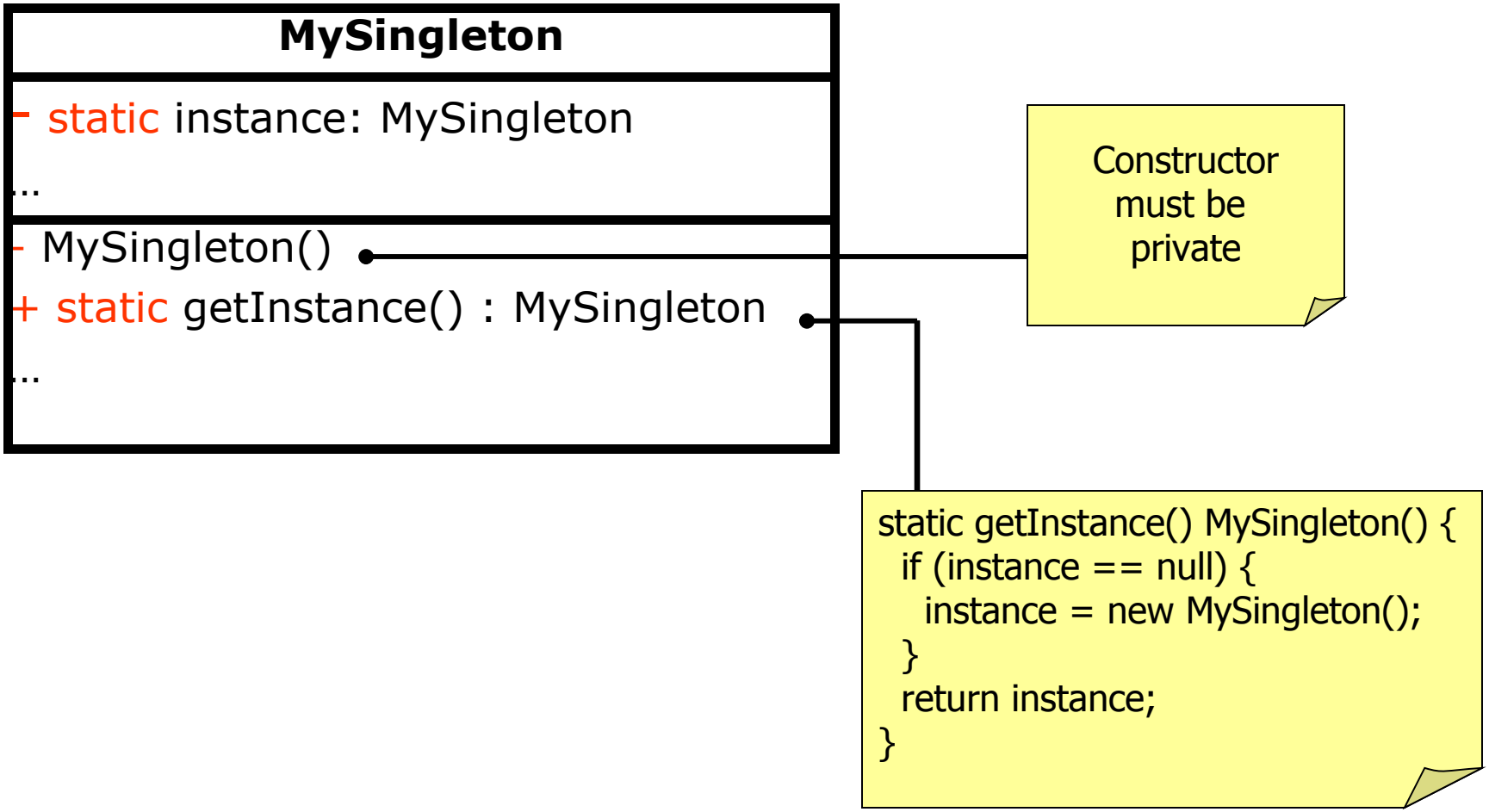
  Ensure a class only has **one instance**

E.g.

- Class to contain the connection between a client and its server
- Class to contain information of current user

Singleton

- Structure



Singleton

- Applicability
  - There must be exactly one instance of a class
  - When the sole instance should be extensible by subclassing

Singleton

- Consequences
  - ☑ Controlled access to sole instance
  - ☑ Reduced name space
    - Is an improvement over global variables
  - ☑ Permits a variable number of instances
  - ☑ More flexible than class operations

Contents

- What is Design Patterns?
- Adapter
- Singleton
- **Factory method**
- Abstract Factory
- Decorator
- Exercises

Factory Method

- Problem
  - Consider a Pizza Store System
  - Each store in the system has its own way to create its pizzas

Factory Method

- Solution

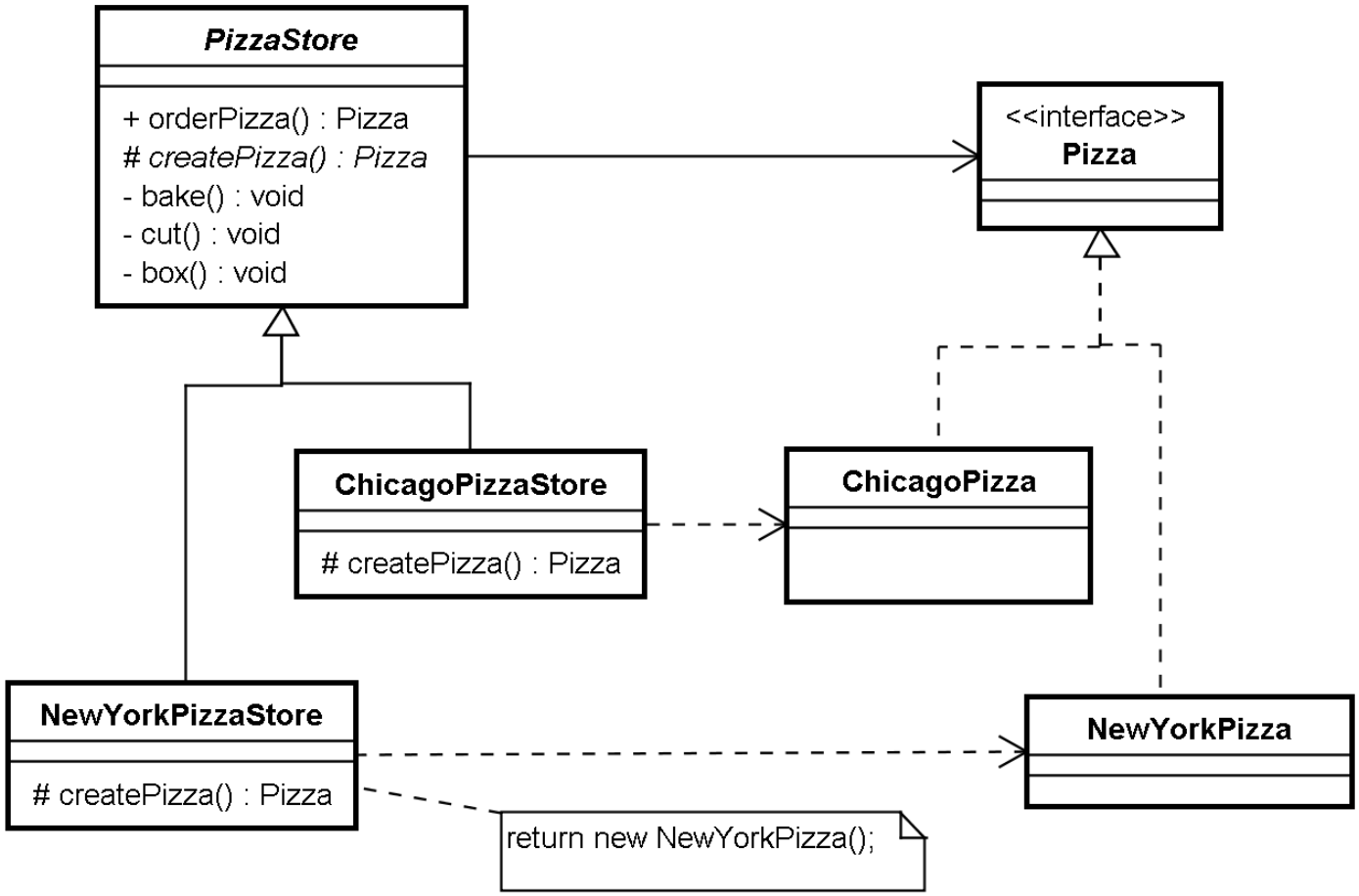
```
public abstract class PizzaStore {
    public Pizza orderPizza() {
        Pizza pizza = createPizza();
        bake(pizza);
        box(pizza);
        return pizza;
    }

    // factory method
    protected abstract Pizza createPizza();
}
```

```
public class ChicagoPizzaStore extends PizzaStore {
   protected Pizza createPizza() {
      return new ChicagoPizza();
   }
}
```

Factory Method

- Solution

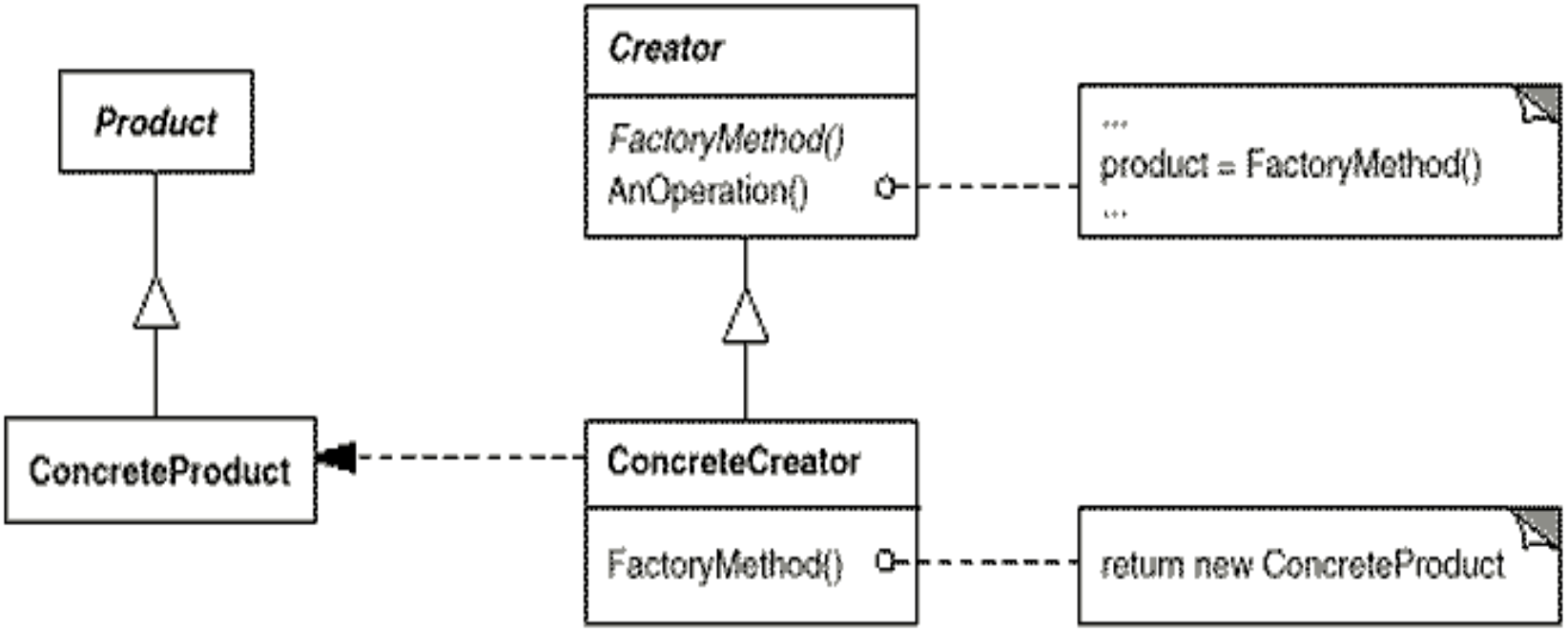


Factory Method

- Intent
  - Define an *interface* for creating an object
  - Let subclasses decide which class to instantiate

Factory Method

- Structure



Factory Method

- Participants
  - **Product** (Pizza)
    - Defines the interface of objects the factory method creates.
  - **ConcreteProduct** (ChicagoPizza)
    - Implements the Product interface.
  - **Creator** (PizzaStore)
    - Declares the factory method, which returns an object of type Product
    - May also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - May call the factory method to create a Product object.
  - **ConcreteCreator** (ChicagoPizzaStore)
    - Overrides the factory method to return an instance of a ConcreteProduct.

Factory Method

- Applicability
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates

Factory Method

- Consequences
  - ☑ Provides hooks for subclasses
  - ☑ Connects parallel class hierarchies

Contents

- What is Design Patterns?
- Adapter
- Singleton
- Factory method
- **Abstract Factory**
- Decorator
- Exercises

Abstract Factory

- Problem
  - Update your Pizza Store System to support many types of products

Abstract Factory

- Solution
  - Temporary solution

```
public abstract class PizzaStore {
    public Pizza orderPizza() {
        Pizza pizza = createPizza();
        bake(pizza);
        box(pizza);
        return pizza;
    }
    public Hotdog orderHotdog() {
        Hotdog hotdog = createHotdog();
        bake(hotdog);
        box(hotdog);
        return hotdog;
    }
    // factory method
    protected abstract Pizza createPizza();
    protected abstract Hotdog createHotdog();
}
```

Abstract Factory

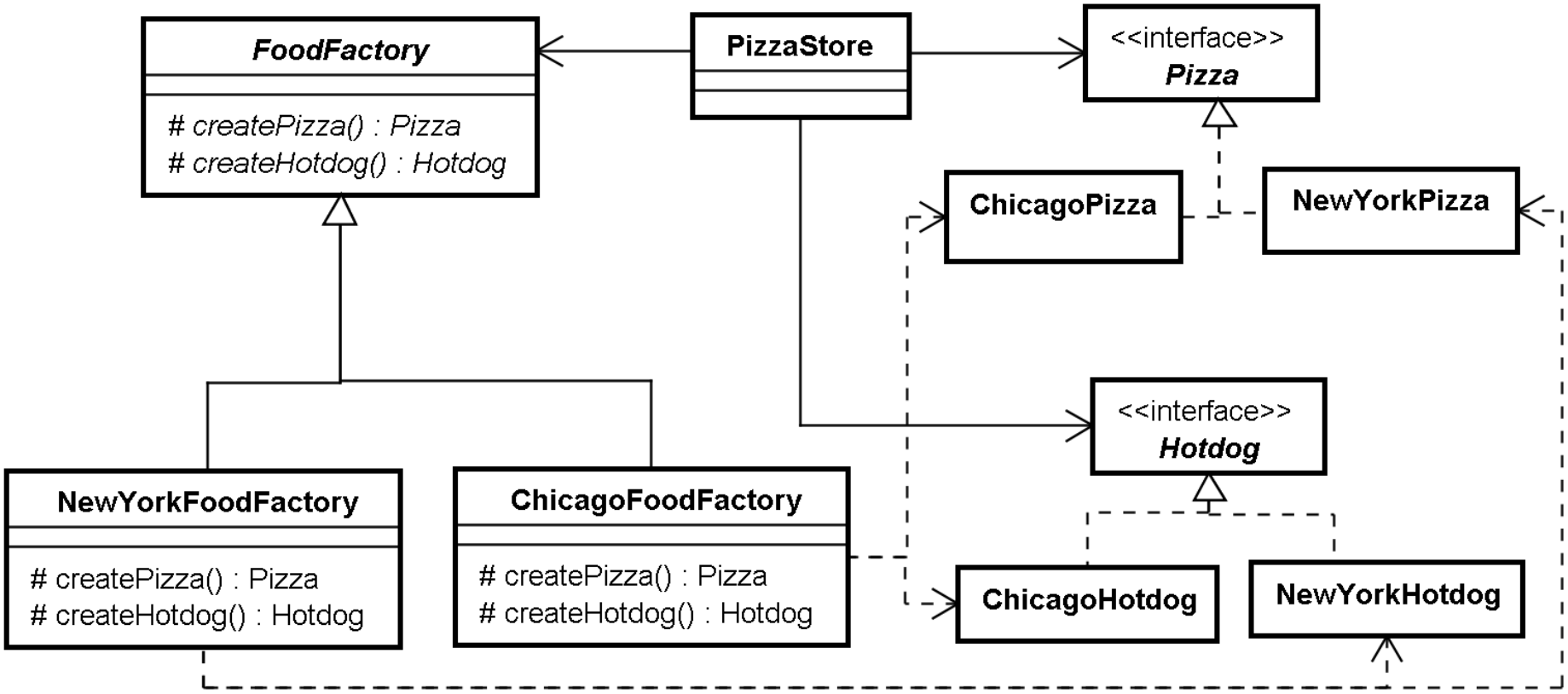
- Solution – temporary solution

```
Public class PizzaStore {
    FoodFactory factory;
    public PizzaStore(FoodFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza() {
        Pizza pizza = factory.createPizza();
        bake(pizza);
        box(pizza);
        return pizza;
    }
    public Hotdog orderHotdog() {
        Hotdog hotdog = factory.createHotdog();
        bake(hotdog);
        box(hotdog);
        return hotdog;
    }
}
```

```
interface FoodFactory {
    Pizza createPizza();
    Hotdog createHotdog();
}
```

Abstract Factory

- Solution

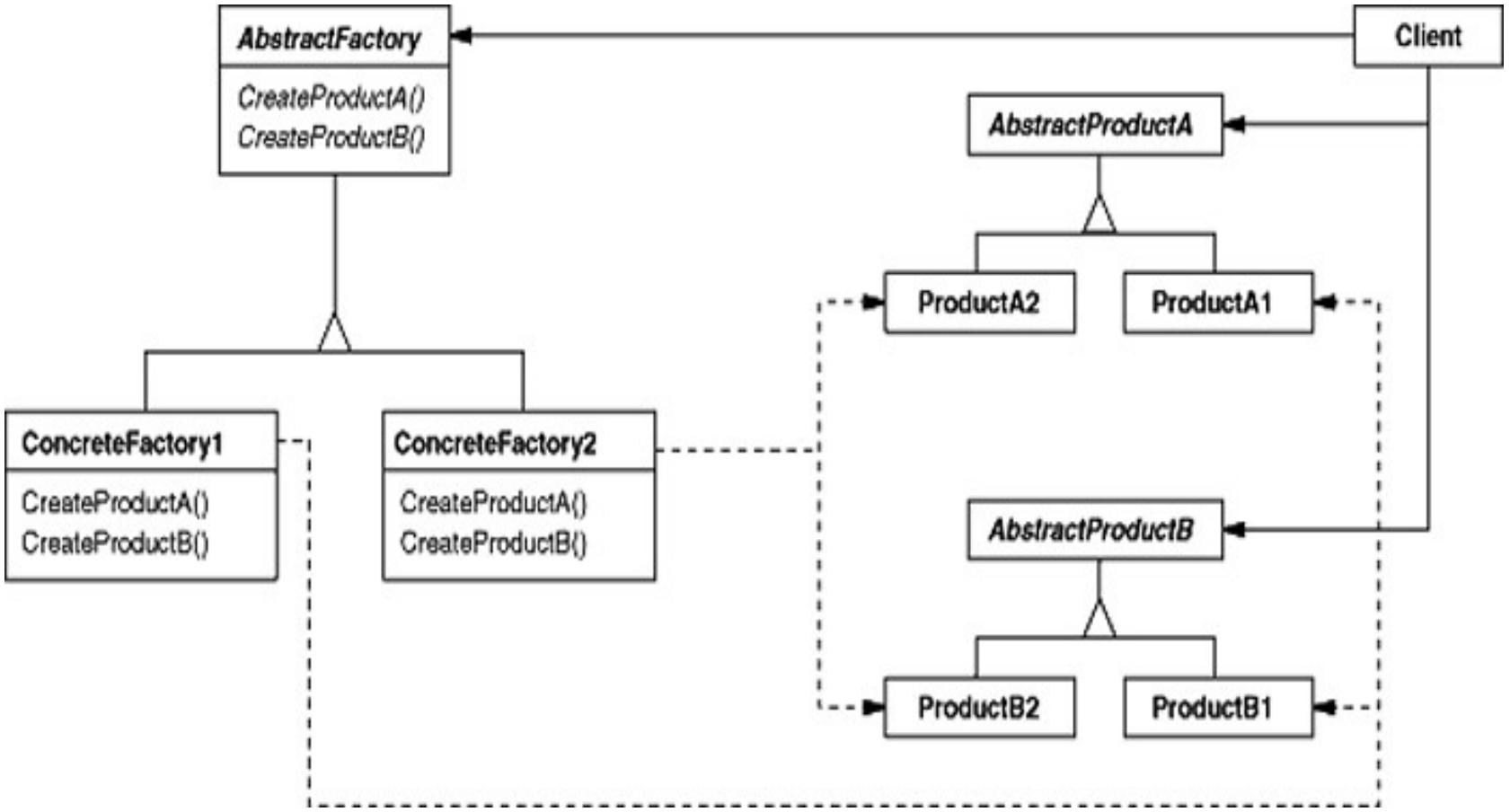


Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Abstract Factory

- Structure



Abstract Factory

- Applicability
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together, and you need to enforce this constraint
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory

- Consequences
  - ☑ Isolates concrete classes
    - Isolates clients from implementation classes
    - Clients manipulate instances through their abstract interfaces
  - ☑ Makes exchanging product families easy
  - ☑ Promotes consistency among products
    - Enforce application to use objects from only one family at a time
  - ☒ Supporting new kinds of products is difficult
    - Supporting new kinds of products requires extending the factory interface

Contents

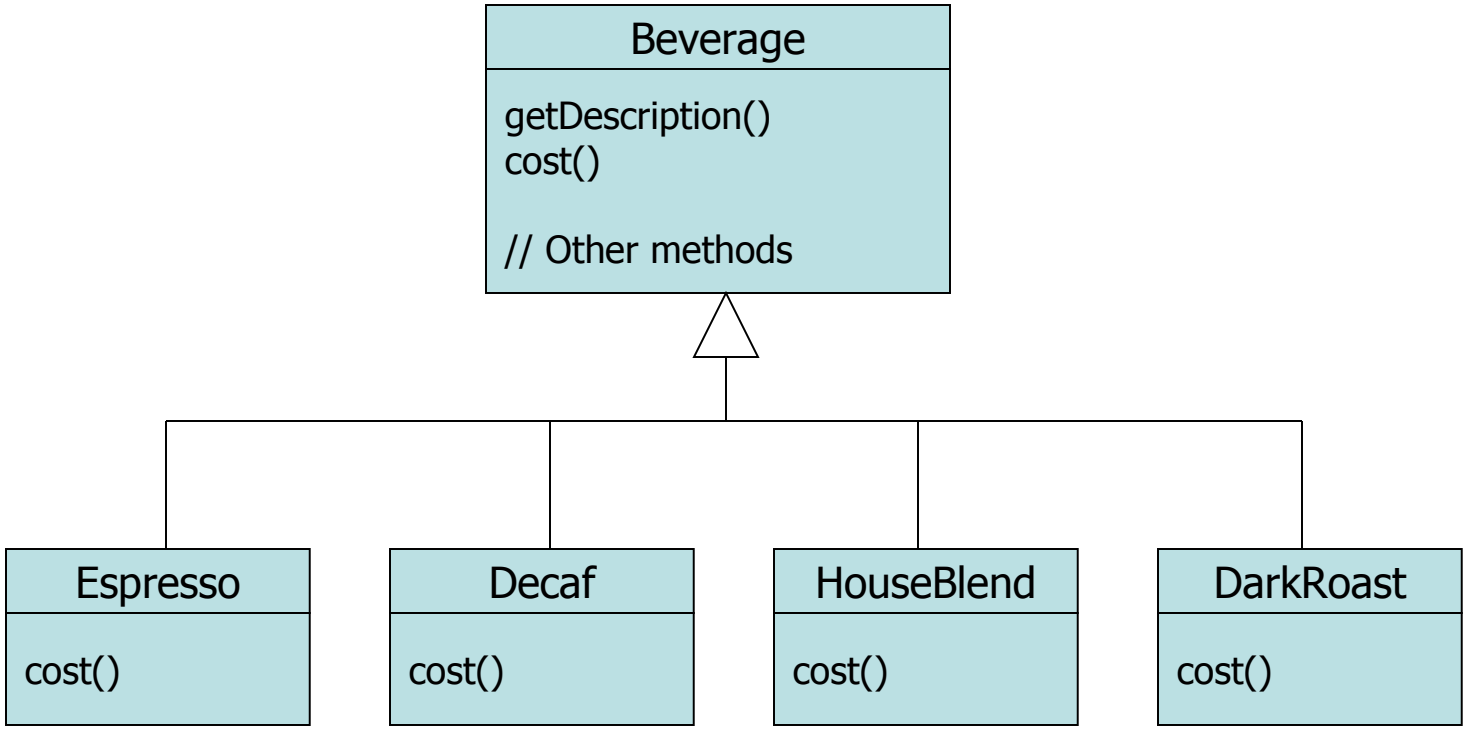
- What is Design Patterns?
- Adapter
- Singleton
- Factory method
- Abstract Factory
- **Decorator**
- Exercises

Decorator

- Problem
  - In Starbuzz Coffee, there are some kinds of coffee such as DarkRoast, Espresso, Decaf...
  - In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk.
  - Starbuzz Coffee charges a bit for each of these, so they really need to get them built into their order system.

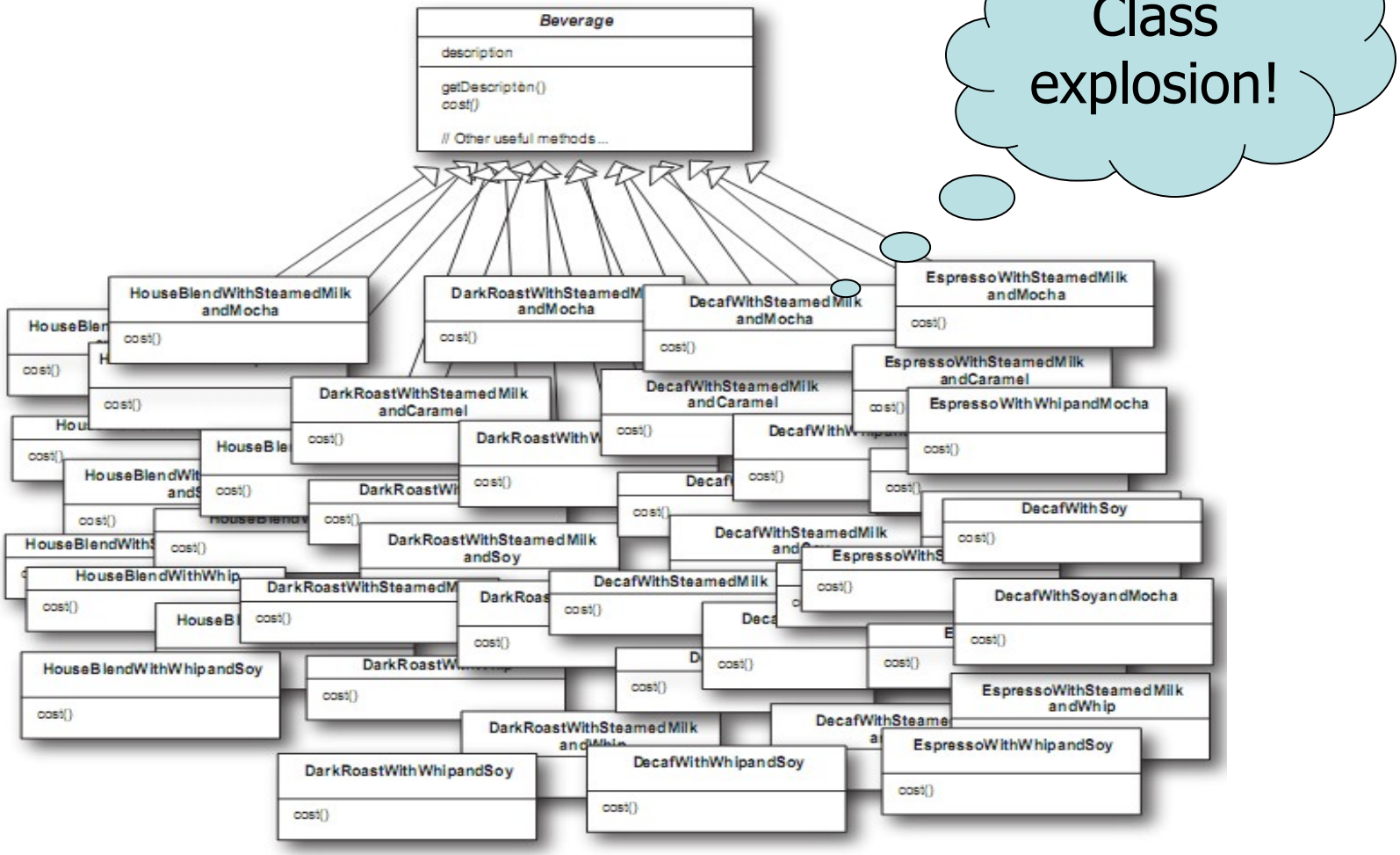
Decorator

- First attempt



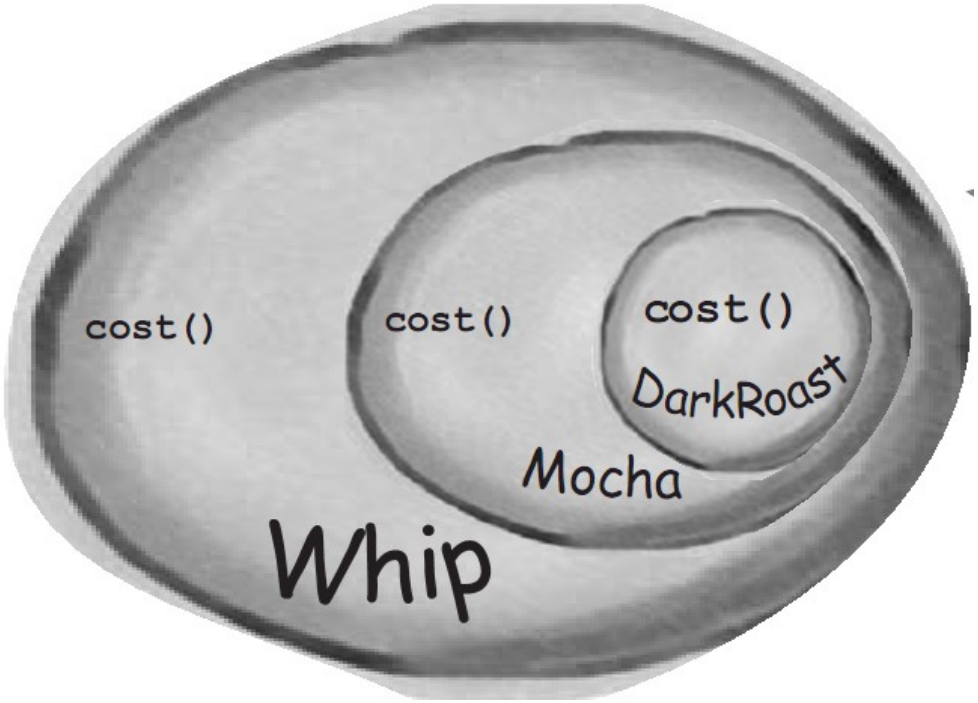
Decorator

- Add condiments support



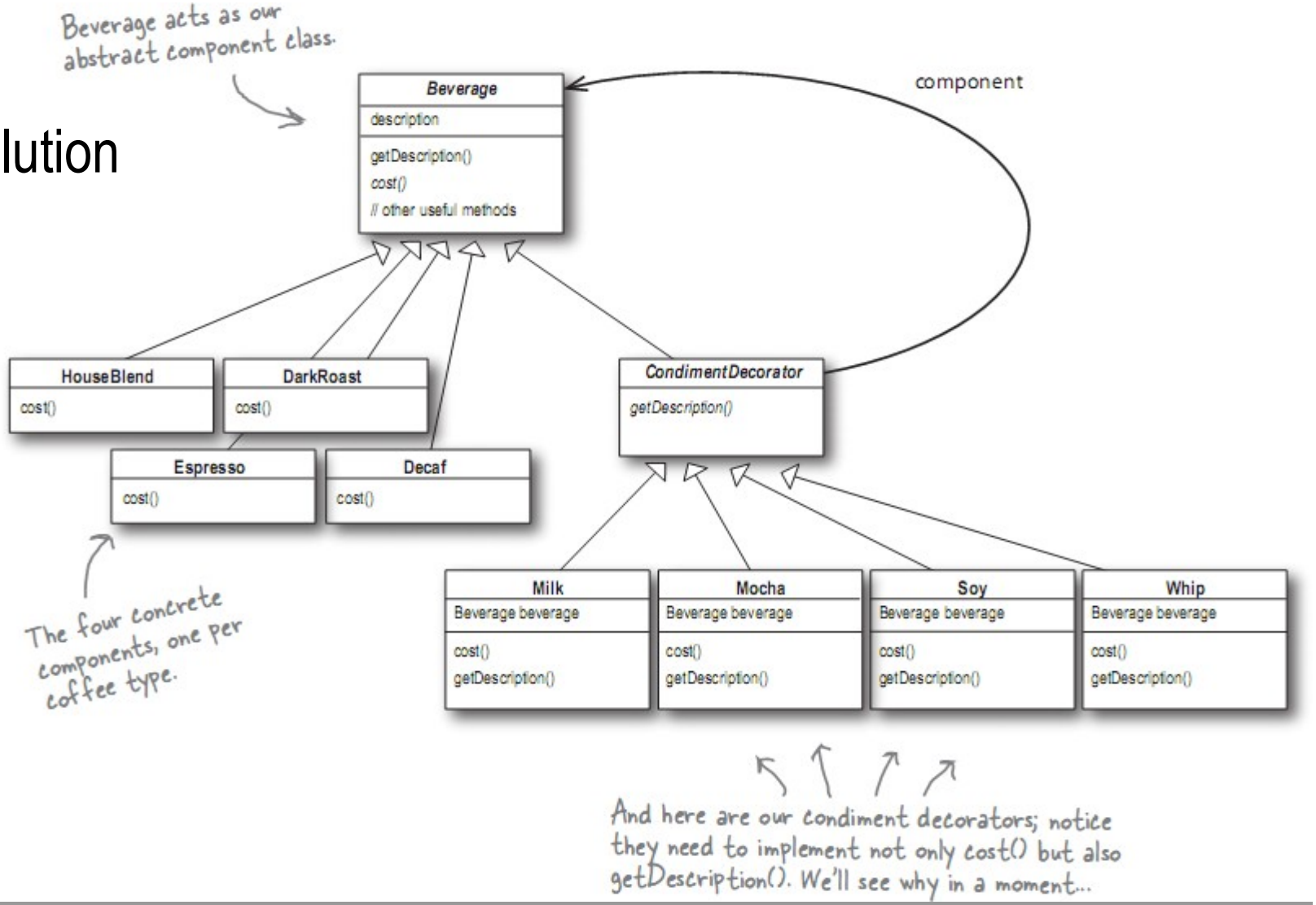
Decorator

- Redesign with Decorator pattern
  - Start with our DarkRoast object
  - Decorate with Mocha
  - And then with Whip



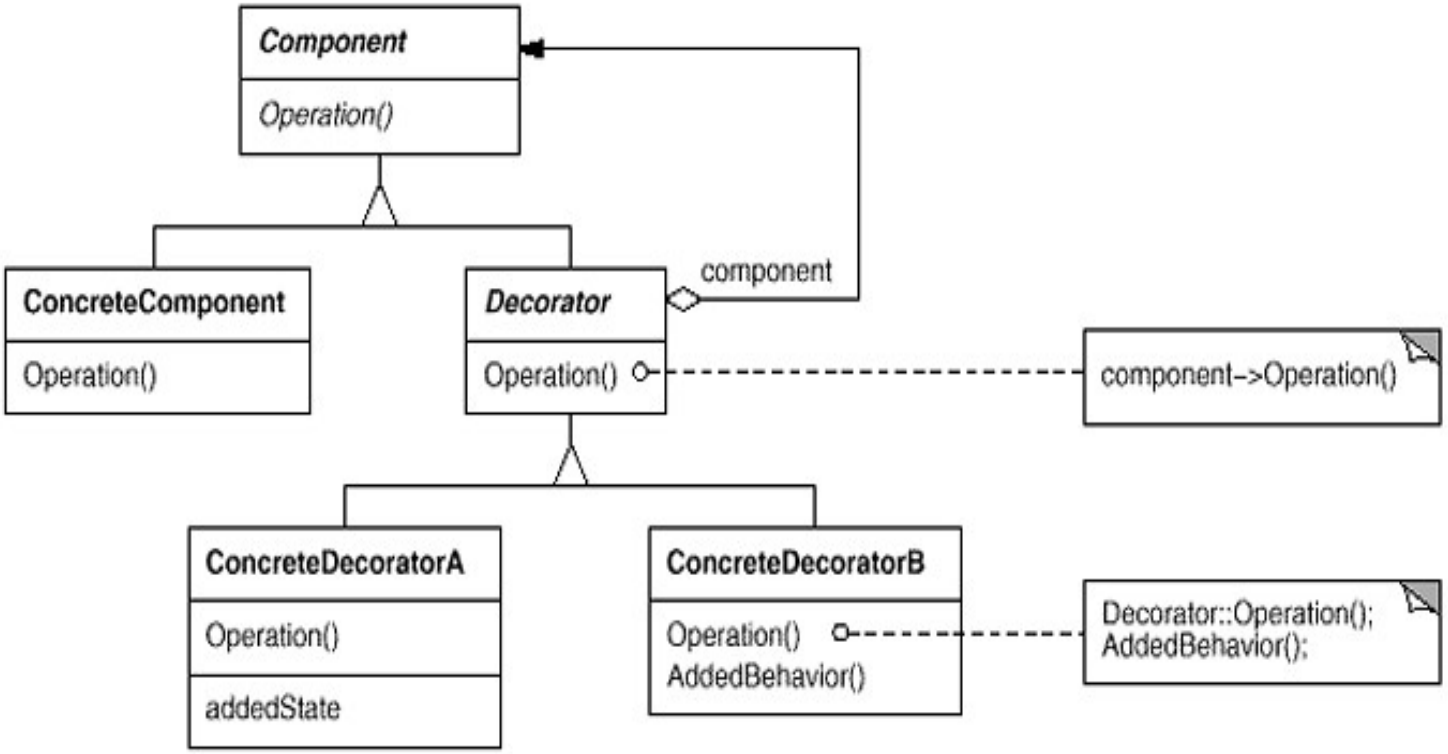
Decorator

- Solution



Decorator

- Structure



Decorator

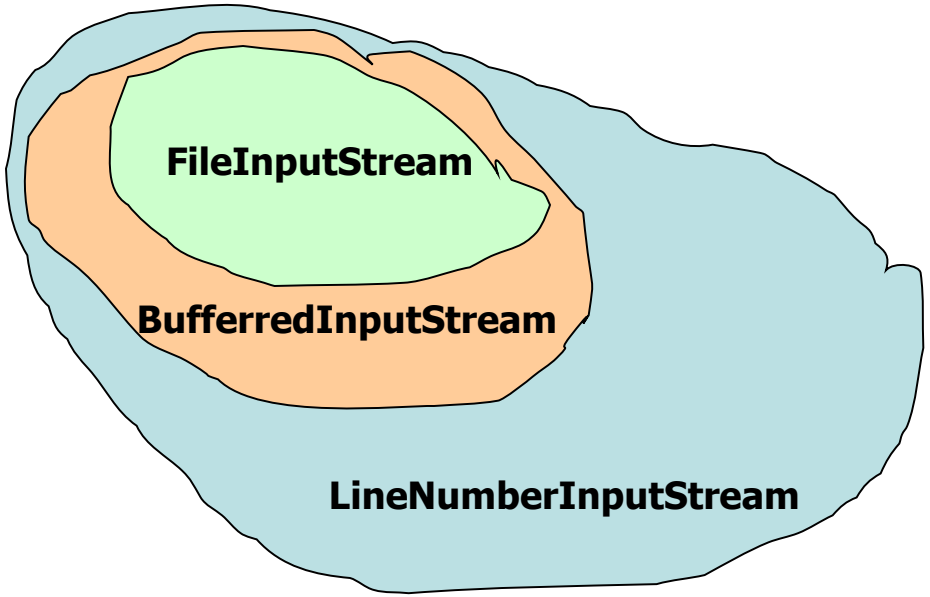
- Applicability
  - To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects
  - Decorate objects dynamically at runtime with as many decorators as we like
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical

Decorator

- Consequences
  - ☑ More flexibility than static inheritance
  - ☑ Avoids feature-laden classes high up in the hierarchy
  - ☒ A decorator and its component aren't identical
  - ☒ Lots of little objects

Decorator

- Real world example
  - Java I/O



Contents

- What is Design Patterns?
- Adapter
- Singleton
- Factory method
- Abstract Factory
- Decorator
- **Exercises**

Exercises

- See *Design Pattern Exercises.doc*

Summary

- *Design Patterns help to isolate changes*
- *Easy to maintenance code*
  - Increase flexibility
  - Simplify logic
- *Standardization*
- *Avoid duplication*

References

- Design Patterns - Elements of Reusable Object-Oriented Software (Gang of Four (GoF))
- Applied Java™ Patterns (Stephen Stelting, Olav Maassen)
- Design Patterns Java Workbook (Steven John Metsker)
- Head First Design Patterns (Eric Freeman & Elisabeth Freeman)

Document Revision History

Date	Version	Description	Revised by
13 Sep 2011	1.0	First version	Tin Bui