Design Patterns

Source: *Design Patterns: Elements of Reusable Object-Oriented Software*

E. Gamma, R. Helm, R. Johnson, J. Vlissides

Overview

- How to Solve a Design Problem
- What’s a Design Pattern
- Why Design patterns
- What’s the Goal of Using Design Patterns
- What to expect from Design Patterns
- Design Pattern Catalog
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

How to Solve a Design Problem

- Not to solve every problem from first principles—simple, open, scalable, etc.
- Reuse the solution that have worked in the past
- If a good solution found, use it again and again
- Result in recurring reusable patterns of classes and communicating objects, that solve specific design problems.

What is a Design Pattern

- Design pattern
  - *describes a* ***problem*** which occurs over and over again in our environment
  - *describes the core of the* ***solution*** to that problem
  - *use this solution* a million times over, without ever doing it the same way twice.

-- *Christopher Alexander*[6]

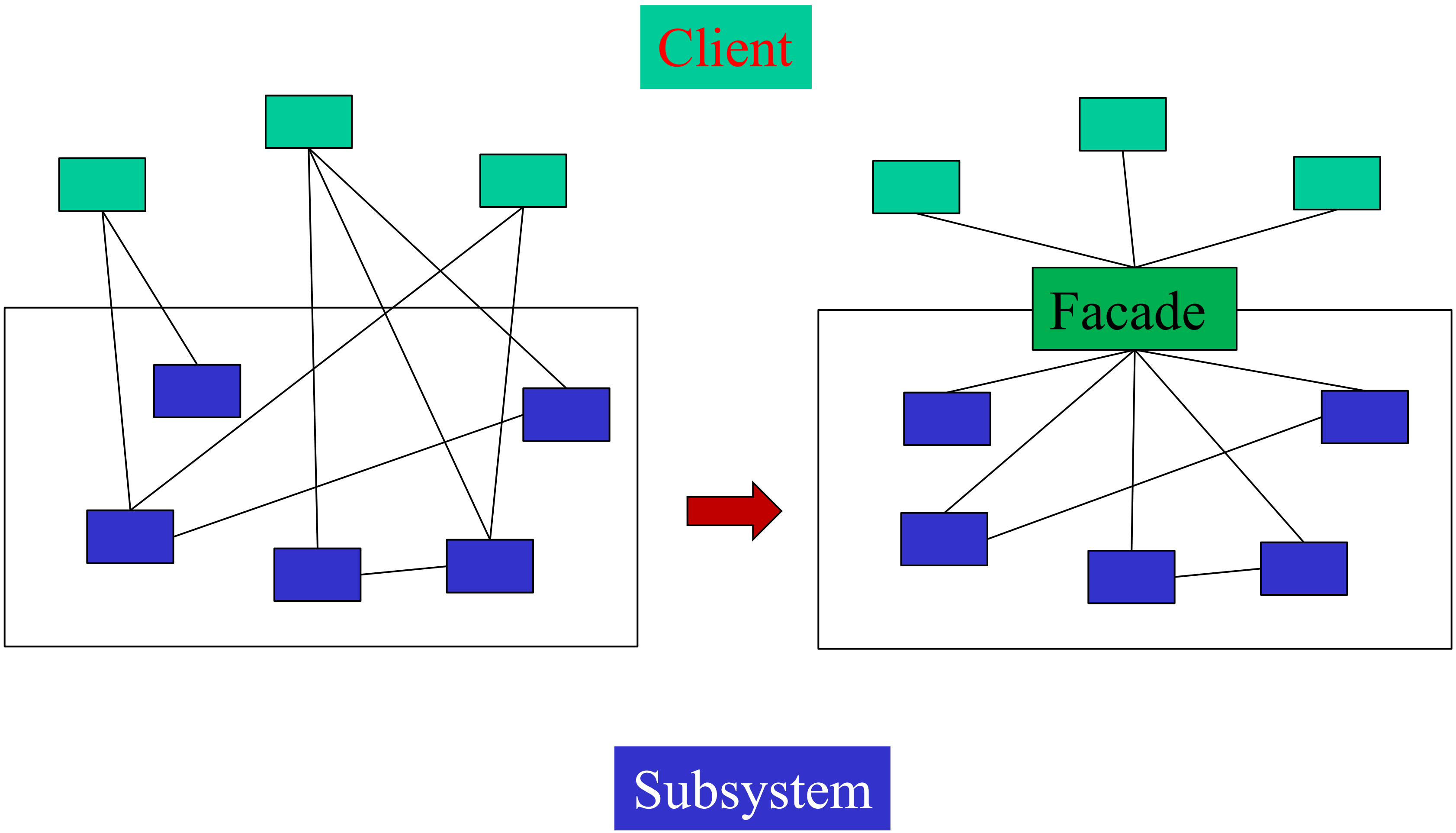
What is a Design Pattern (2)

- The Pattern Name
  - Use to describe a **design problem**, its **solution**, and **consequence** in a word or two
  - Use to communicate to others
  - E.g., Façade; Singleton
- The Problem
  - Describe **when** to apply the pattern
  - Explain the problem and its context
  - E.g., to provide a simple/high-level interface to complex subsystems

What's a Design Pattern (3)

- The Solution
  - Describes the **elements** that make up the *design*, their *relationship*, *responsibilities*, and *collaborations*
  - Can be applied to many different situations
  - E.g., clients communicate with the subsystem by sending requests to a *Facade*
- The Consequence
  - **Results** and **trade-offs** of applying the pattern
  - **Critical for evaluating design alternatives** and for **understanding** the costs and benefits of applying the pattern
  - E.g., Shield clients from subsystem components and promote **weak** coupling between the subsystem and its clients

Facade



Why Design Patterns

- Easier to **reuse** successful designs and architecture
- Expressing **proven techniques** as design patterns make them more accessible to developers
- Help you choose design **alternatives** that makes a system reusable
- Improve documentation and maintenance of existing systems
- Help a designer get a design “right” faster

What's the Goal of Using Design Patterns

- Design for change [3]
- Communicate problems and solutions [5]
- Capture **expert** design experiences in designing *Object-Oriented software*
- UML Tools:
  - IBM Rational Rose
  - ArgoUML (http://argouml.tigris.org/)

Notes:

1. UML—Unified Modeling Language
2. [3]: The evolution of a design pattern typically involves the addition or removal of a group of modeling elements, such as classes, attributes, operations, and relationships

What to expect from Design Patterns

- A Common Design Vocabulary
  - Use to communicate, document, and explore design alternatives
- A **Documentation** and Learning Aid
  - Make it easier to understand existing systems
- An Adjunct to **Existing** Methods
  - Provide experiences of expert designers
- A Target for **Refactoring**
  - Help determining how to reorganize a design

Design Pattern Catalogs

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Design Pattern Catalog

- Creational Patterns
  - **Abstract** the **instantiation process**
  - Make a system **independent** of how its objects are *created*, *composed*, and *represented*
  - E.g., Abstract Factory; Singleton

Design Pattern Catalog (2)

- Structural Patterns
  - Concern with how classes and objects are *composed* to form a **larger** structures
  - Use inheritance to compose interfaces or implementations
  - E.g., Façade; Proxy

Design Pattern Catalog (3)

- Behavioral Patterns
  - Concern with algorithms and the assignment of responsibilities between *objects*
  - Describe patterns of communication between *objects* or *class*
  - E.g., Iterator; Chain of Responsibility

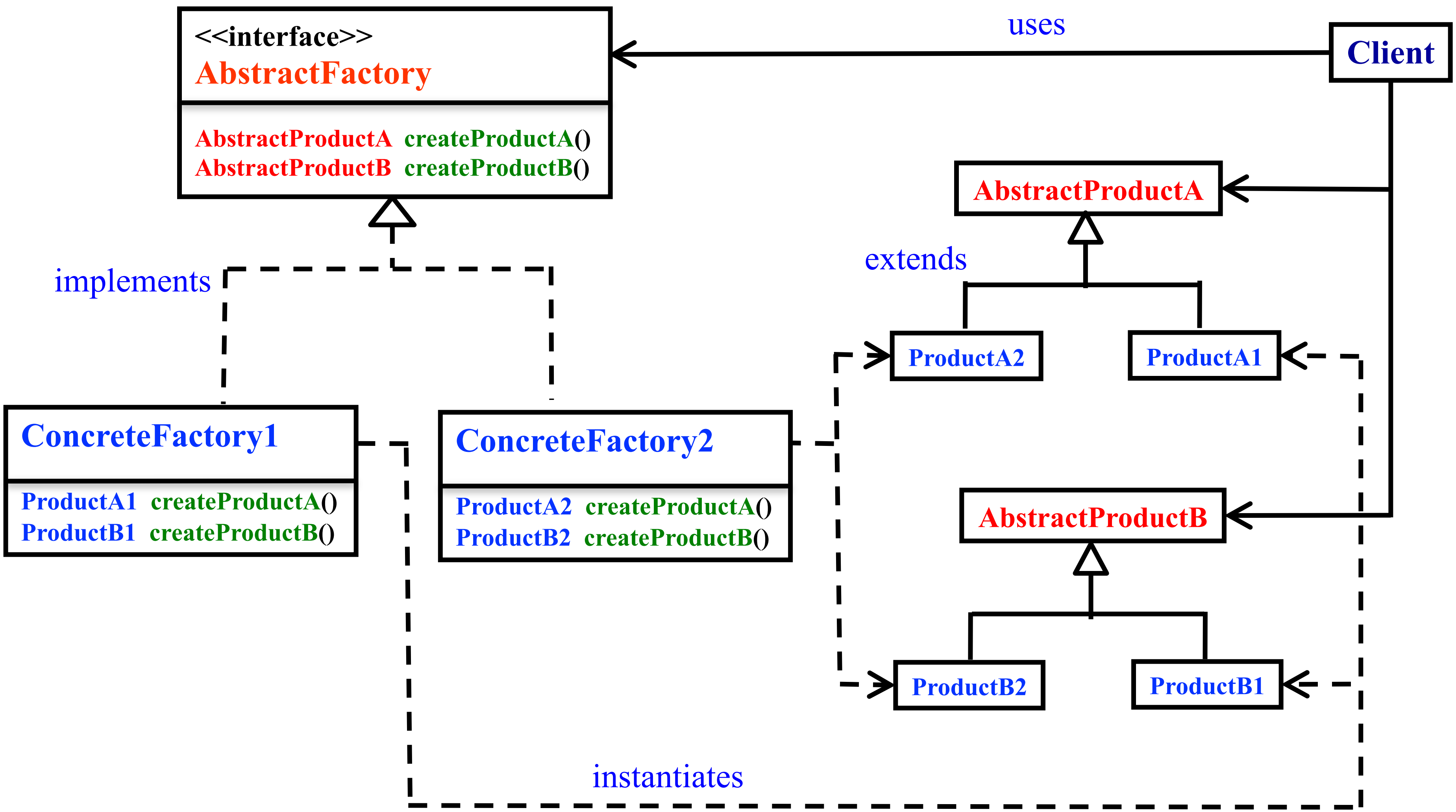
Creational Patterns

- Abstract Factory
- Factory Method
- Builder
- Prototype
- Singleton

Abstract Factory

- Provide an interface (e.g., interface *or* abstract class) for creating **families** of *related* or *dependent* **product objects** without specifying their *concrete* classes
- Implementation:
  - **Abstract** Factory declares an interface for creating product objects
  - **Concrete** Factories as singletons
  - **Product** objects are actually created within a *Concrete* Factory

Abstract Factory (2)



Abstract Factory (3)

```
// sample code
AbstractFactory concreteFactory1 = new ConcreteFactory1();
AbstractProductA productA1 =  concreteFactory1.createProductA();
                                          // return new ProductA1();
AbstractProductB productB1 =  concreteFactory1.createProductB();
                                          // return new ProductB1();

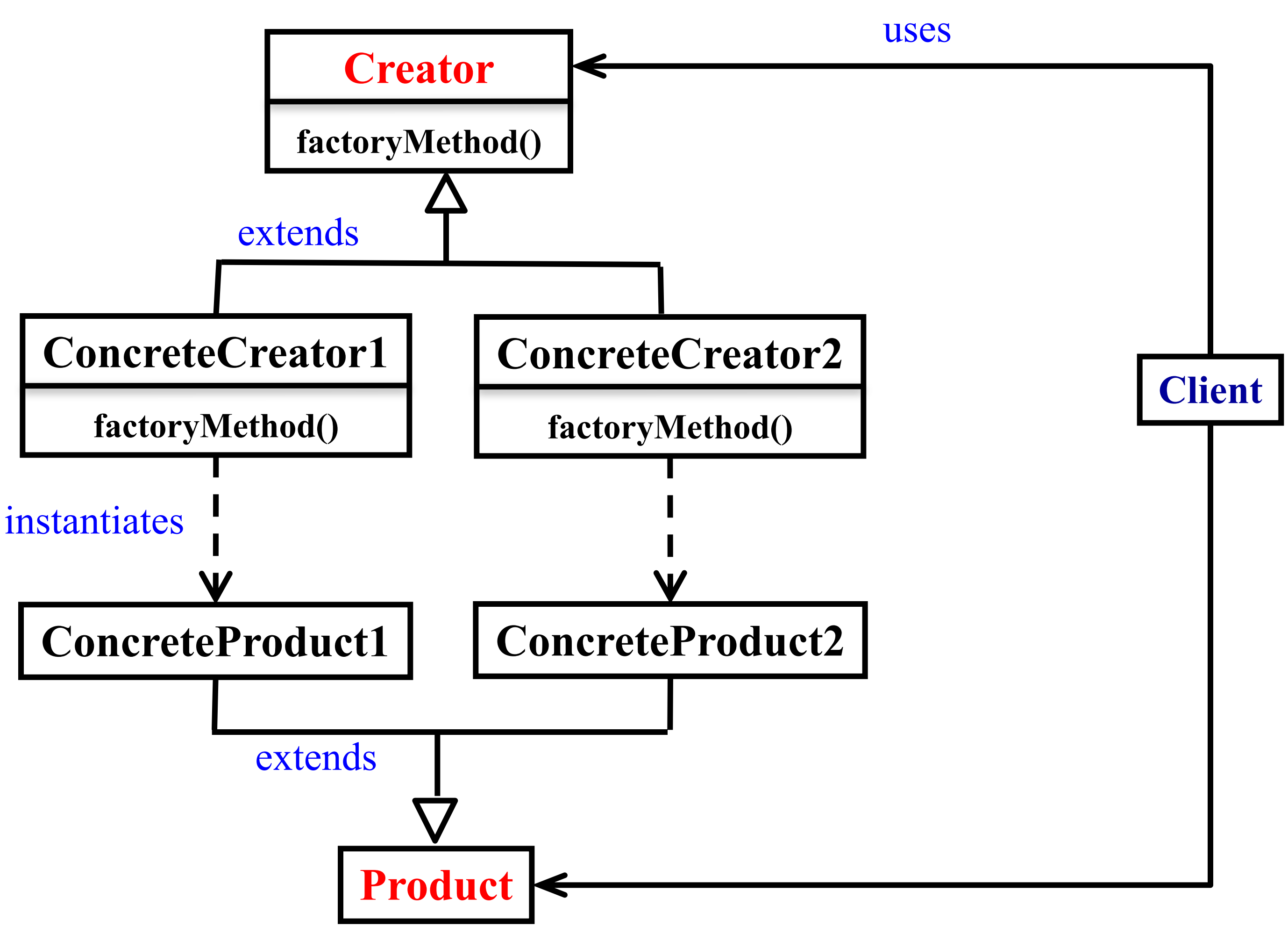
AbstractFactory concreteFactory2 = new ConcreteFactory2();
AbstractProductA productA2 =  concreteFactory2.createProductA();
                                           // return new ProductA2();
AbstractProductB productB2 =  concreteFactory2.createProductB();
                                           // return new ProductB2();
```

Ref: http://www.developer.com/java/other/article.php/626001

Factory Method

- Define an interface **for creating an object**, but let subclasses decide which *class* to instantiate
- Implementation:
  - Abstract class **Creator** declares the factory method, which returns an object of Product
  - Subclass **ConcreteCreator** **overrides** the factory method to return an instance of a **ConcreteProduct**

Factory Method (2)



Factory Method (3)

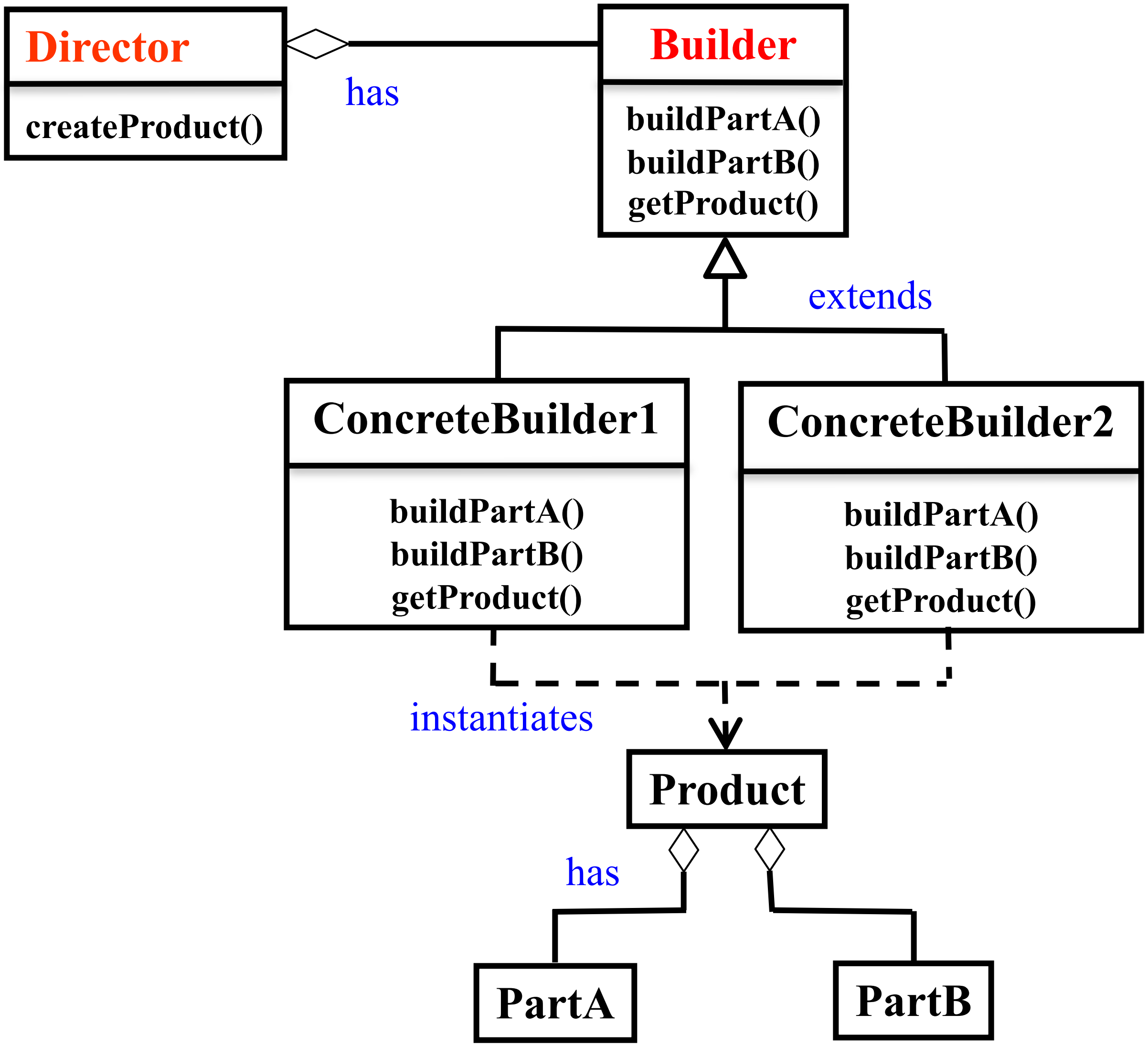
- public **abstract** class Creator {
  protected Product **product** = new **Product** ();
  public abstract Product factoryMethod();
 }
- public class ConcreteCreator1 extends Creator {
  public Product factoryMethod() { …… return product; }
 }
- public class Client {
  public static void main(String[] args) {
  // could use input "type" to determine "right" concrete creator
  Creator **myCreator** = new ConcreteCreator1();
  Product result = **myCreator**.factoryMethod(); . . . }
}

- **Ref**: http://www.apwebco.com/gofpatterns/creational/FactoryMethod.html

Builder

- Separate the **construction** of a complex object from its **representation** (i.e., parts) so that the same construction process can create different **representation**
- Implementation:
  - Use an abstract class **Builder** that defines an operation for each component/part that a **director** may ask it to create
  - Use a subclass **ConcreteBuilder** to **override** operations defined in **Builder** abstract class

Builder (2)


Director
createProduct()
has
Builder
buildPartA()
buildPartB()
getProduct()
extends
ConcreteBuilder1
buildPartA()
buildPartB()
getProduct()
ConcreteBuilder2
buildPartA()
buildPartB()
getProduct()
instantiates
Product
has
PartA
PartB

Builder (3)

- public class **Director** { // take a **builder** to create a product

  ```
  public Product createProduct (Builder builder) {
          builder.buildPartA();   builder.buildPartB();
          return  builder.getProduct(); }   }
  ```
- public **abstract** class **Builder** {

  ```
  protected Product product = new Product ();
  public abstract void buildPartA();
  public abstract void buildPartB();
  public abstract Product getProduct();    }
  ```
- public class **ConcreteBuilder1** extends **Builder** {

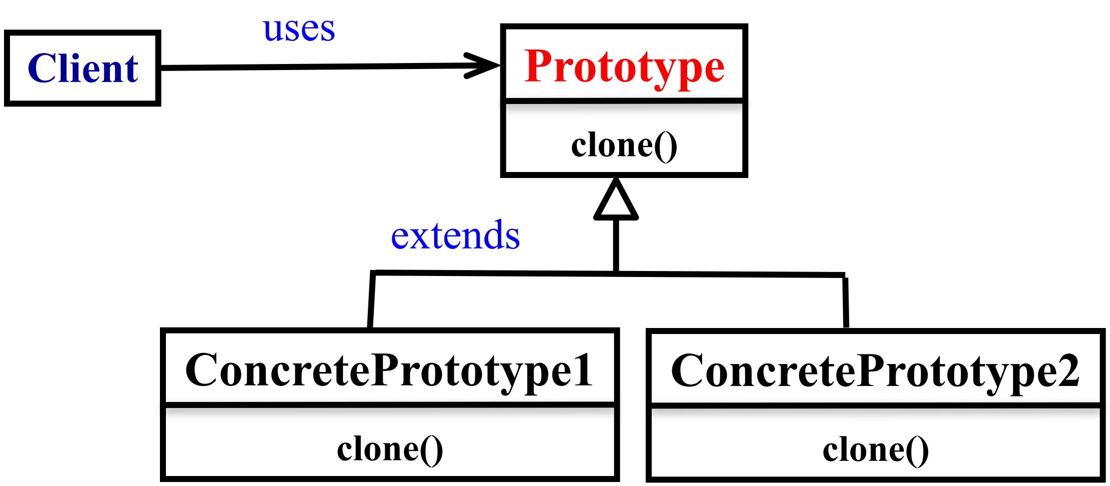
  ```
  public void buildPartA() { // add partA based on ConcreteBuilder1  preferences }
  public void buildPartB() { // add partB based on ConcreteBuilder1  preferences }
  public Product getProduct() { return product; }   }
  ```
- public class **Client** {

  ```
  public static void main(String[] args) {
  Director director = new Director();
  Builder myBuilder = new ConcreteBuilder1(); // diff builder, diff product
  Product result = director.createProduct(myBuilder);   . . . } }
  ```
- **Ref**: http://www.apwebco.com/gofpatterns/creational/Builder.html

Prototype

- Specify the **kinds of objects** to create using a prototype instance, and create ***new*** *objects* by copying this prototype
- Implementation:
  - Abstract class **Prototype** declares the **clone** method, which returns a copy of cloned object
  - Subclass **ConcretePrototype** **overrides** the clone method to return a copy of itself

Prototype (2)



Client
uses
Prototype
clone()
extends
ConcretePrototype1
clone()
ConcretePrototype2
clone()

Prototype (3)

- public **abstract** class **Prototype** implements **Cloneable** {
  public **abstract** **Prototype** **clone**();
  }
- public class **ConcretePrototype1** extends **Prototype** implements **Cloneable** {
  public **Prototype** **clone**() { …… return **new** **ConcretePrototype1()**; }
  }
- public class **Client** {
  public static void main(String[] args) {
  // could use input "type" to determine "right" concrete prototype
  Prototype **myPrototype** = new **ConcretePrototype1**();
  Prototype result = **myPrototype**.**clone**(); . . . }
  }

- **Ref**: http://www.apwebco.com/gofpatterns/creational/Prototype.html
- A class implements the Cloneable **interface** to indicate to the *Object.clone()* method that it is legal for that method to make a field-for-field copy of instances of that class.

Singleton

- Ensure a class has only one instance, and provide a **global** point of access to it
- Implementation:
  - Define getInstance operation that lets clients access its **unique** instance.
  - Define a constructor with visibility *protected* or *private*
  - Define an instance attribute with scope *static*

Singleton (2)

Singleton
- instance : Singleton
+ getInstance() : Singleton # Singleton()

+	Public
-	Private
#	Protected
/	Derived (can be combined with one of the others)
~	Package

- **Ref**: https://www.uml-diagrams.org/visibility.html

Singleton (3)

- ```
  public class Singleton {
      private static Singleton instance = null;
      protected Singleton() {
        // Exists only to defeat instantiation.
      }
      public static Singleton getInstance() {
        if( instance == null) {
          instance = new Singleton();
        }
        return instance;
      }
  }
  ```
  ```

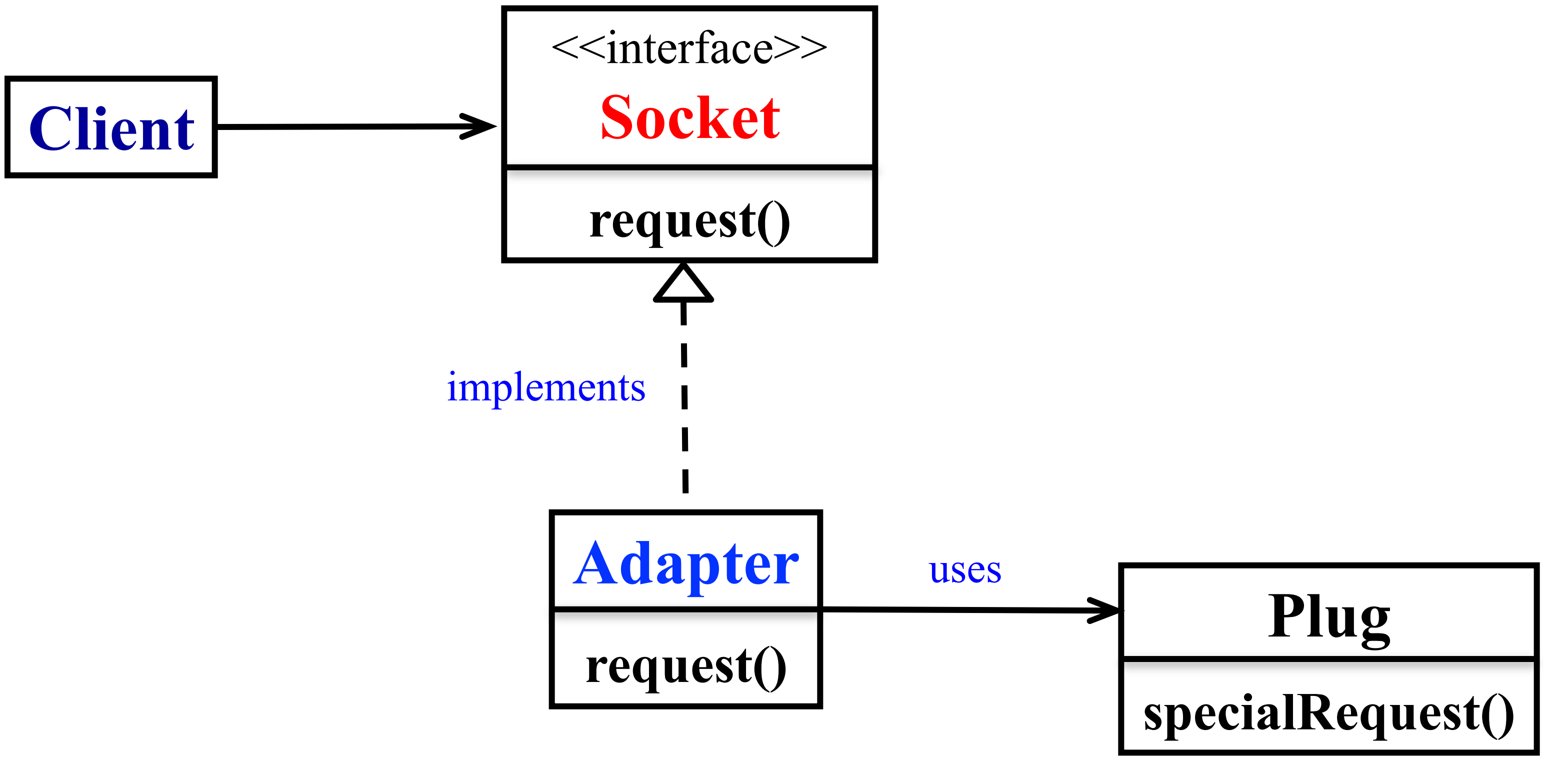
Structural Patterns

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

Adapter

- Convert the interface of a class into another interface clients expect
- Let classes work together that could not otherwise because of *incompatible interfaces*
- Implementation:
  - Interface **Socket** defines request operation
  - Subclass **Adapter** **overrides** request operation

Adapter (2)



<<interface>>
Socket
request()
Client
implements
Adapter
request()
uses
Plug
specialRequest()

Adapter (3)

- public interface **Socket** {

```
        public String getOutput();
}
```

- public class **Plug** {

```
        private String specification = "5 AMP";
        public String getInput() { return specification; }
}
```

- public class **Adapter** implements **Socket** {

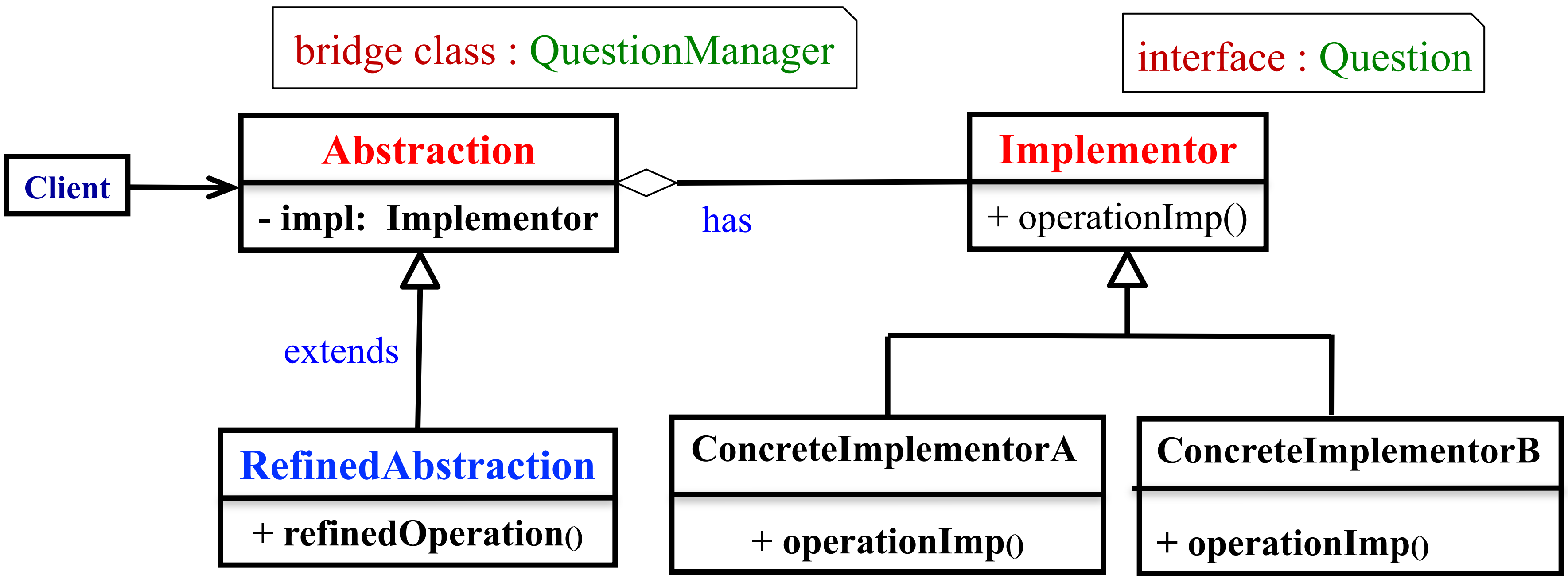
```
        public String getOutput(int input) {
                // could use "input" to determine "right" interface/class
                        Plug  plug = new Plug();
                        return plug.getInput(); }
}
```

- Ref: http://www.allapplabs.com/java_design_patterns/adapter_pattern.htm

Bridge

- Decouple an abstraction from its implementation so that two can vary independently
- Share an implementation among multiple objects
- Implementation:
  - Abstraction defines operations
  - Implementor implements the operations defined in Absraction

Bridge (2)



bridge class : QuestionManager
interface : Question
Client
Abstraction
- impl: Implementor
has
Implementor
+ operationImp()
extends
RefinedAbstraction
+ refinedOperation()
ConcreteImplementorA
+ operationImp()
ConcreteImplementorB
+ operationImp()

Bridge (3)

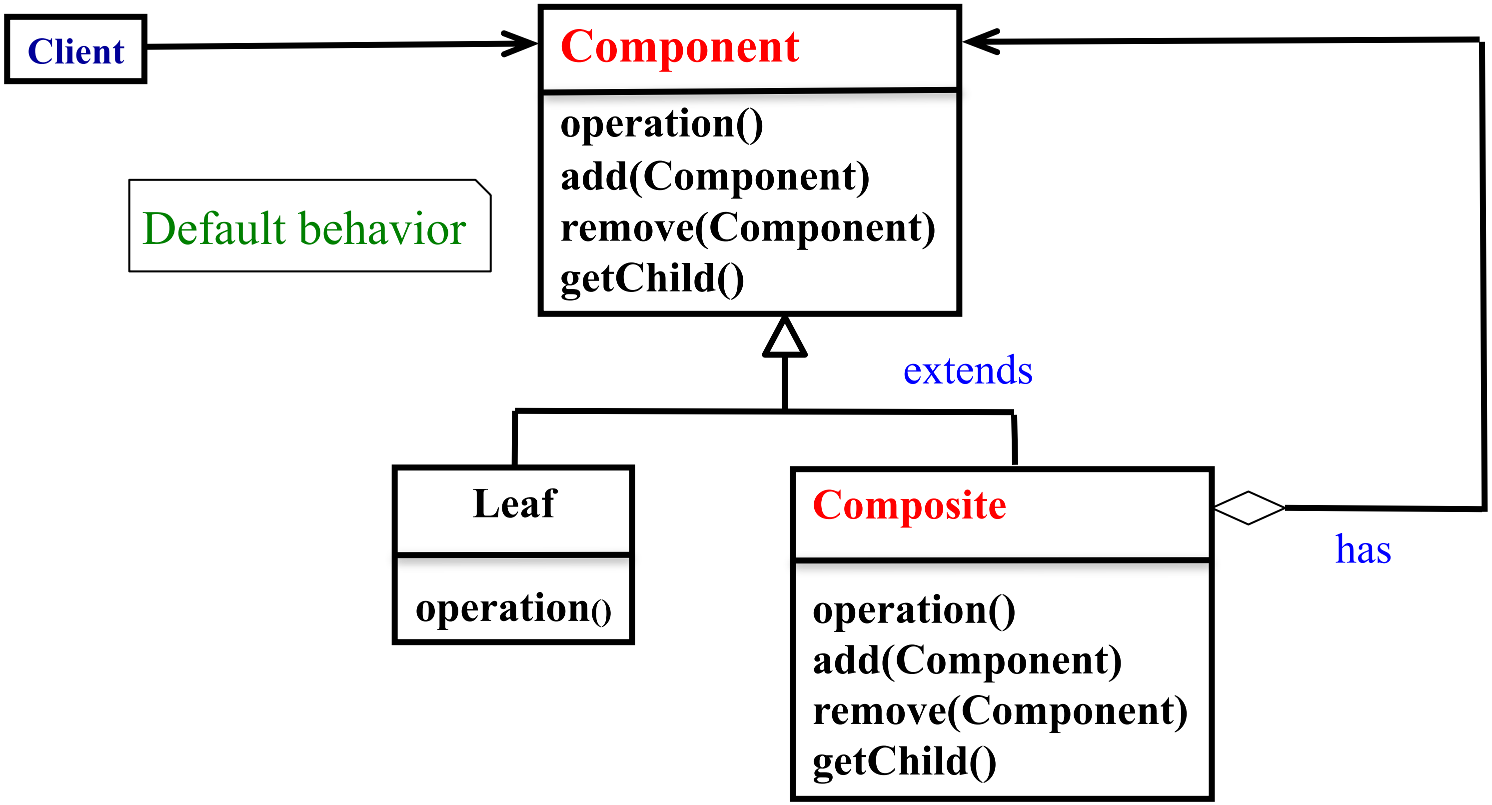
- class QuestionManager { // abstraction has implementor--Question
  protected Question questDB; //instantiate it later ...... }
- interface Question { // implementor
  public void nextQuestion();
  public void newQuestion(String q);
  public void displayQuestion(); .... }
- class QuestionFormat extends QuestionManager { // refined abstraction
  public void displayAll() { ..... } }
- class JavaQuestions implements Question { // **concrete** implementor
  public void nextQuestion() { if(current <= questions.size() - 1) current++; } ... }
- class **TestBridge** { public static void main(String[] args) {
  QuestionFormat **questions** = new QuestionFormat("Java Language"); //refined abstraction
  questions.questDB = new JavaQuestions(); // concrete implementor
  questions.display(); questions.next();
  questions.newOne("What is object? ");
  questions.displayAll(); } ..... }

Ref: http://www.javatpoint.com/bridge-pattern

Composite

- Compose objects into tree structures
- Let clients treat individual objects and compositions of objects uniformly
- Implementation:
  - Class **Component**
    - defines and implements ***default*** behavior
  - Subclass **Composite**
    - defines behavior for components having children
    - implements child-related operations in class **Component**

Composite (2)



Client
Component
operation()
add(Component)
remove(Component)
getChild()
Default behavior
extends
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild()
has

Composite (3)

- public class Employee { // Component

```
        private String name; private double salary; private Vector subordinates;
        public Vector getSubordinates() { return subordinates; }
        public void setSubordinates(Vector subordinates) { this.subordinates = subordinates; }
        public Employee(String name, double sal) { setName(name); setSalary(sal);
                subordinates = new Vector(); }
        public void add(Employee e) { subordinates.addElement(e);  }
        public void remove(Employee e) {subordinates.remove(e);  }
}
```

- public class Composite extends Employee {

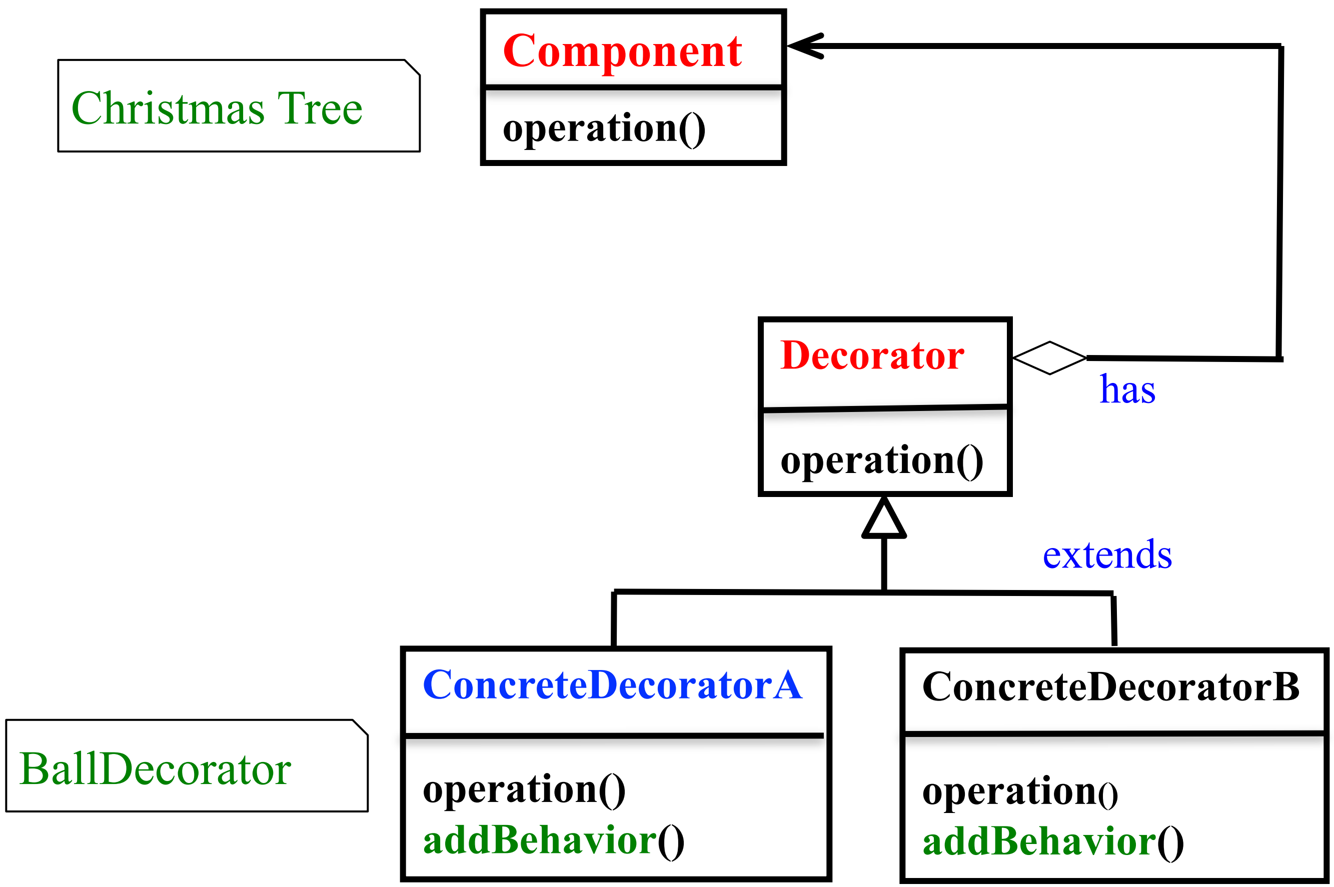
```
         private void addEmployeesToTree() {
        Employee CFO = new Employee("CFO", 300000);
        Employee headFinance1 = new Employee("Head Finance. North Zone", 20000);
        Employee headFinance2 = new Employee("Head Finance. West Zone", 22000);
        Employee accountant1 = new Employee("Accountant1", 10000);
        Employee accountant2 = new Employee("Accountant2", 9000);
        Employee accountant3 = new Employee("Accountant3", 11000);
        Employee accountant4 = new Employee("Accountant4", 12000);
        CFO.add(headFinance1); CFO.add(headFinance2);
        headFinance1.add(accountant1);      headFinance1.add(accountant2);
        headFinance2.add(accountant3);      headFinance2.add(accountant4); }
}
```

- Ref: http://www.allapplabs.com/java_design_patterns/composite_pattern.htm

Decorator

- Attach **additional** responsibilities to an object dynamically
- Provide a flexible alternative to subclasses for **extending** functionality
- Implementation:
  - Class **Decorator** defines operations
  - Subclass **ConcreateDecorator** *implements* operations and adds more operations

Decorator (2)



Christmas Tree
Component
operation()
Decorator
operation()
has
extends
BallDecorator
ConcreteDecoratorA
operation()
addBehavior()
ConcreteDecoratorB
operation()
addBehavior()

Decorator (3)

- public **abstract class Decorator** { // Decorator
  public **abstract** void **place** (**Branch** branch); // abstract method

```
}
```

- public class **ChristmasTree** { // Component
  private Branch **branch**;
  public Branch **getBranch**() {return branch; }

```
}
```

- public class **BallDecorator** extends **Decorator** { // ConcreteDecorator
  public **BallDecorator** (**ChristmasTree** tree) {
  Branch branch = tree.getBranch();
  place(branch); }
  public void **place** (Branch branch) {
  branch.put("ball"); }

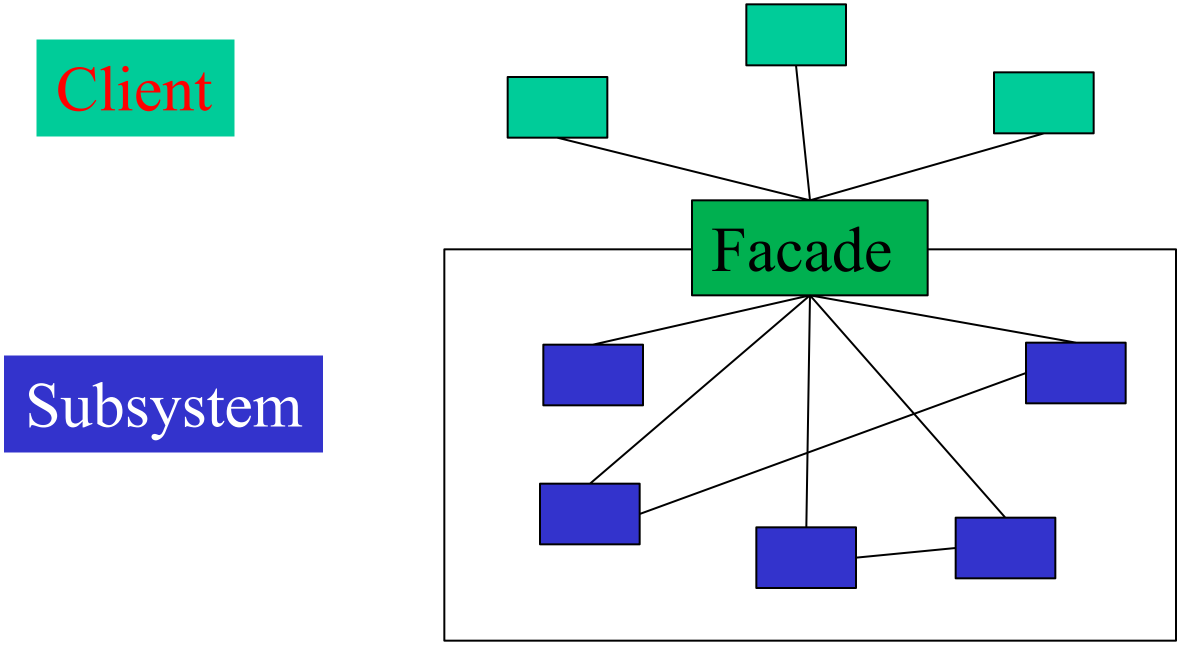
```
}
```

- **Ref**: http://www.allapplabs.com/java_design_patterns/decorator_pattern.htm

Façade

- Provide a unified interface to a set of *interfaces* in a set of subsystems
- Define a higher-level interface that makes the subsystems easier to use
- Implementation:
  - Interface **Façade** defines a set of operations
  - **Subclass** within the subsystem implements the operations in the interface

Façade (2)



Façade (3)

- public interface Store { // Facade

```
        public Goods  getGoods();
}
```

- public class FinishedGoodsStore implements Store {

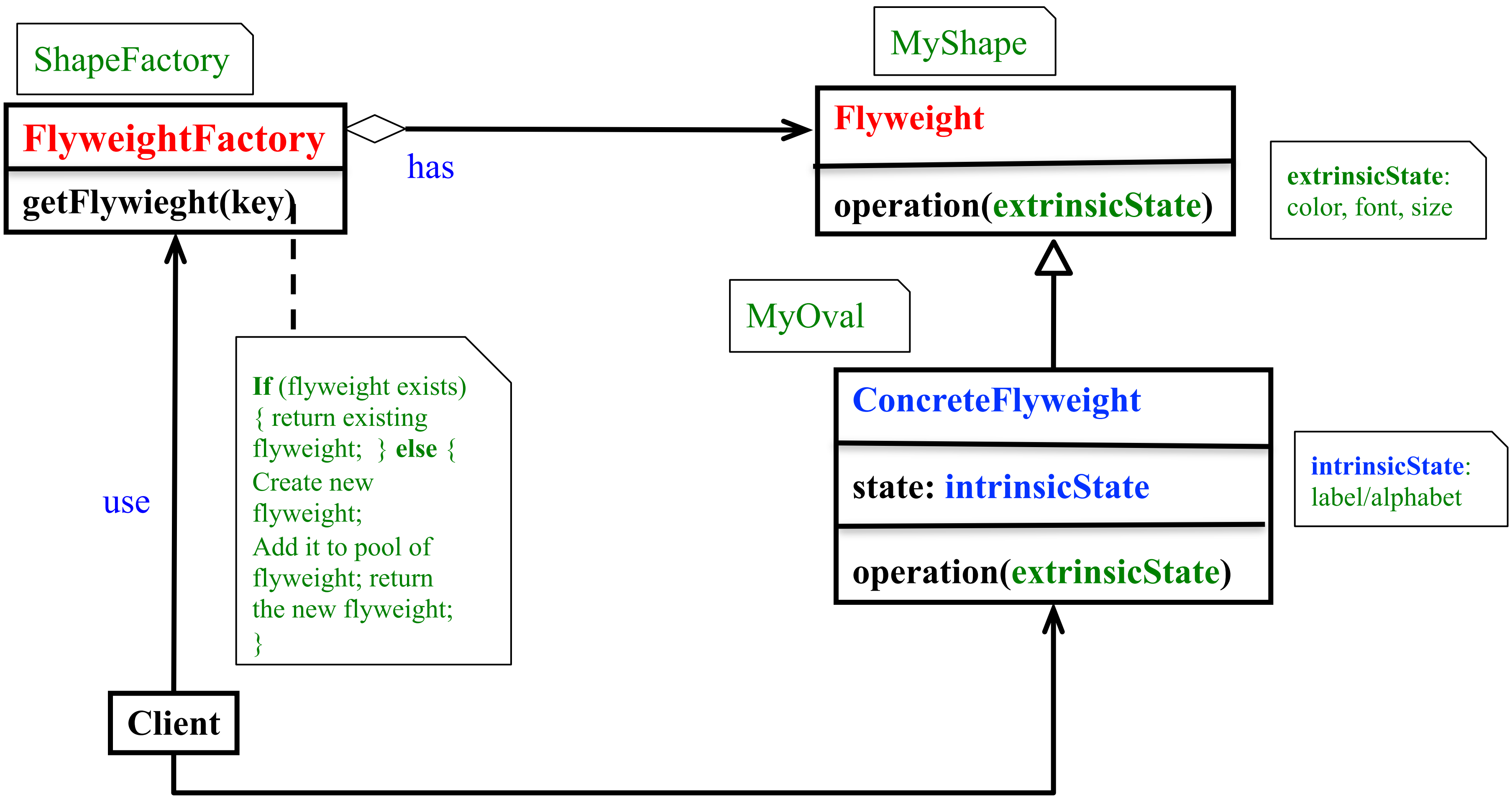
```
        public Goods  getGoods() {
                return new FinishedGoods();        }
}
```

- Ref: http://www.allapplabs.com/java_design_patterns/facade_pattern.htm

Flyweight

- Use sharing to support large numbers of *fine-grained* objects efficiently
- Implementation:
  - Class **FlyweightFactory** creates and manages flyweight objects
  - Interface **Flyweight** defines operations taking **extrinsic** state from a client
  - Subclass **ConcreteFlyweight**
    - implements the Flyweight interface
    - adds storage for **intrinsic** state, if any.
    - must be **sharable**. Any **state** it stores must be **intrinsic**, that is, it must be independent of the ConcreteFlyweight object's context.

Flyweight (2)



Ref: https://en.wikipedia.org/wiki/Flyweight_pattern

Flyweight (3)

- class ShapeFactory { // FlyweightFactory
  private static final HashMap **shapes**= new HashMap();
  public static MyShape getShape(String label) { // label: intrinsic/essential
  MyShape **concreteShape** = (MyShape) **shapes**.get(label);
  if (concreteShape == null) {
  if (label.equals("O")) {
  concreteShape = new MyOval(label); …..
  shapes.put(label, concreteShape); }
  return **concreteShape**; } }

- public interface MyShape { // Flyweight
  public void **draw**(Graphics g, int x, int y, int width, int height, Color color,
  boolean fill, String font); }. // color, font: extrinsic

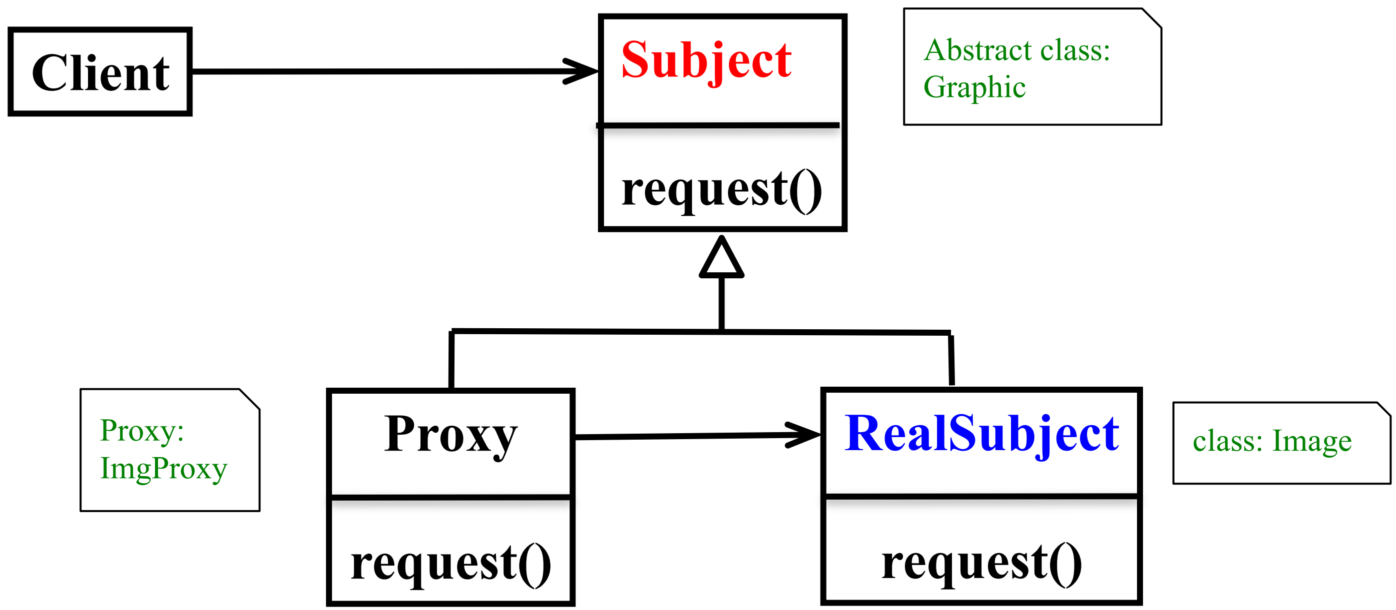
- class **MyOval** implements MyShape { // concrete flyweight
  private String label; public MyOval(String label) { this.label = label; }
  public void **draw** (Graphics oval, …) { …;
  if (fill) oval.fillOval(x, y, width, height); } }

Ref: http://javapapers.com/design-patterns/flyweight-design-pattern/

Proxy

- Provide a substitute of placeholder for another *object* to control access to it
- **Postpone** the *creation* until you need the actual object.
- Implementation:
  - Class ***Subject*** defines the common interface for ***RealSubject*** and ***Proxy***, so that a Proxy can be used anywhere a RealSubject is expected.
  - Class **Proxy** maintains a reference that lets the proxy access the **real** subject

Proxy (2)



Client
Subject
request()
Abstract class:
Graphic
Proxy:
ImgProxy
Proxy
request()
RealSubject
request()
class: Image

Proxy (3)

- abstract class Graphic { // Subject
  ```
  public abstract void load();
  public abstract void draw(); ...
  }
  ```
- class Image extends Graphic { // RealSubject
  ```
  public void load() { ... }
  public void draw() { ... } ...
  }
  ```
- class ImgProxy extends Graphic { // proxy
  ```
  public void load() {
          if( image == null ) {
                  image = new Image(filename);   } ...
  public void draw() { ... } ...
  }
  ```

Behavioral Patterns

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

Reference:

1. *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma et al.
2. *The Unified Modeling Language User Guide*, G. Booch et al.
3. *A Model Transformation Approach for Design Patterns Evolutions*, IEEE 13th International Symposium and Workshop on Engineering of Computer Based Systems, J. Dong et al.
4. *Designing Enterprise Applications with The Java 2 Platform*, N. Kassem et al.
5. *Core J2EE Patterns: Best Practices and Design Strategies*, D. Alur et al.
6. A Pattern Language, C. Alexander et al.