Software Engineering

Dr. Young-Woo Kwon

Today, we will discuss

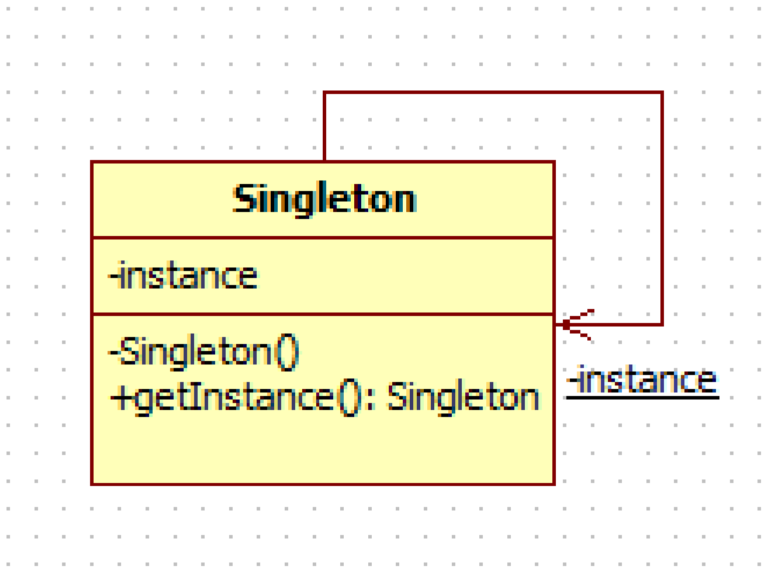
- Software Design Patterns

Outline of the Lecture

- **Design Patterns**
  - **Usefulness of design patterns**
  - **Design Pattern Categories**
- Patterns covered
  - **Composite**
  - **Adapter**
  - **Bridge**
  - Facade
  - Proxy
  - Command
  - Observer
  - Strategy
  - Abstract Factory
  - Builder

Before we start,

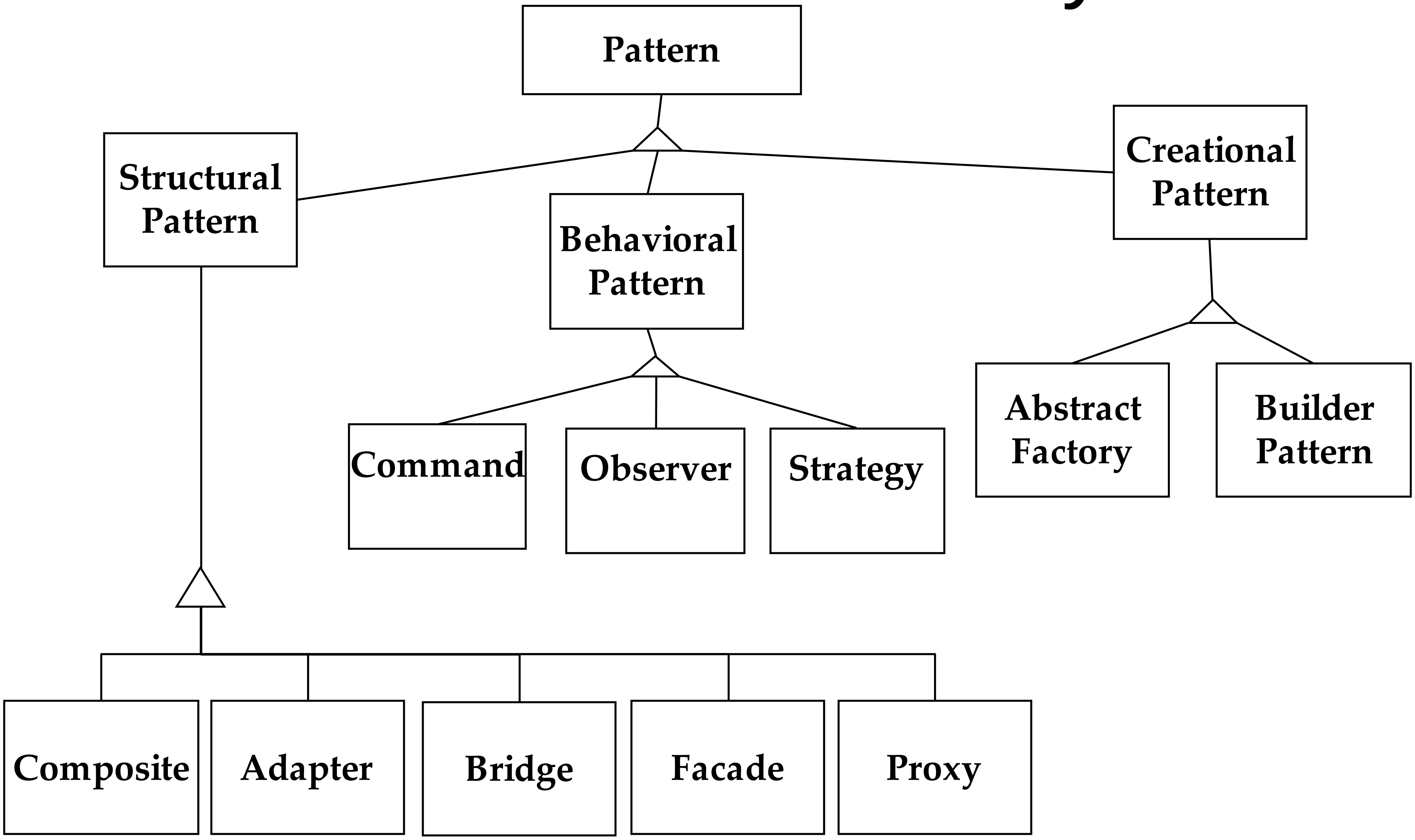
- Write a class that can be instantiated only once in Java. Only one object is created and it is shared by multiple objects.



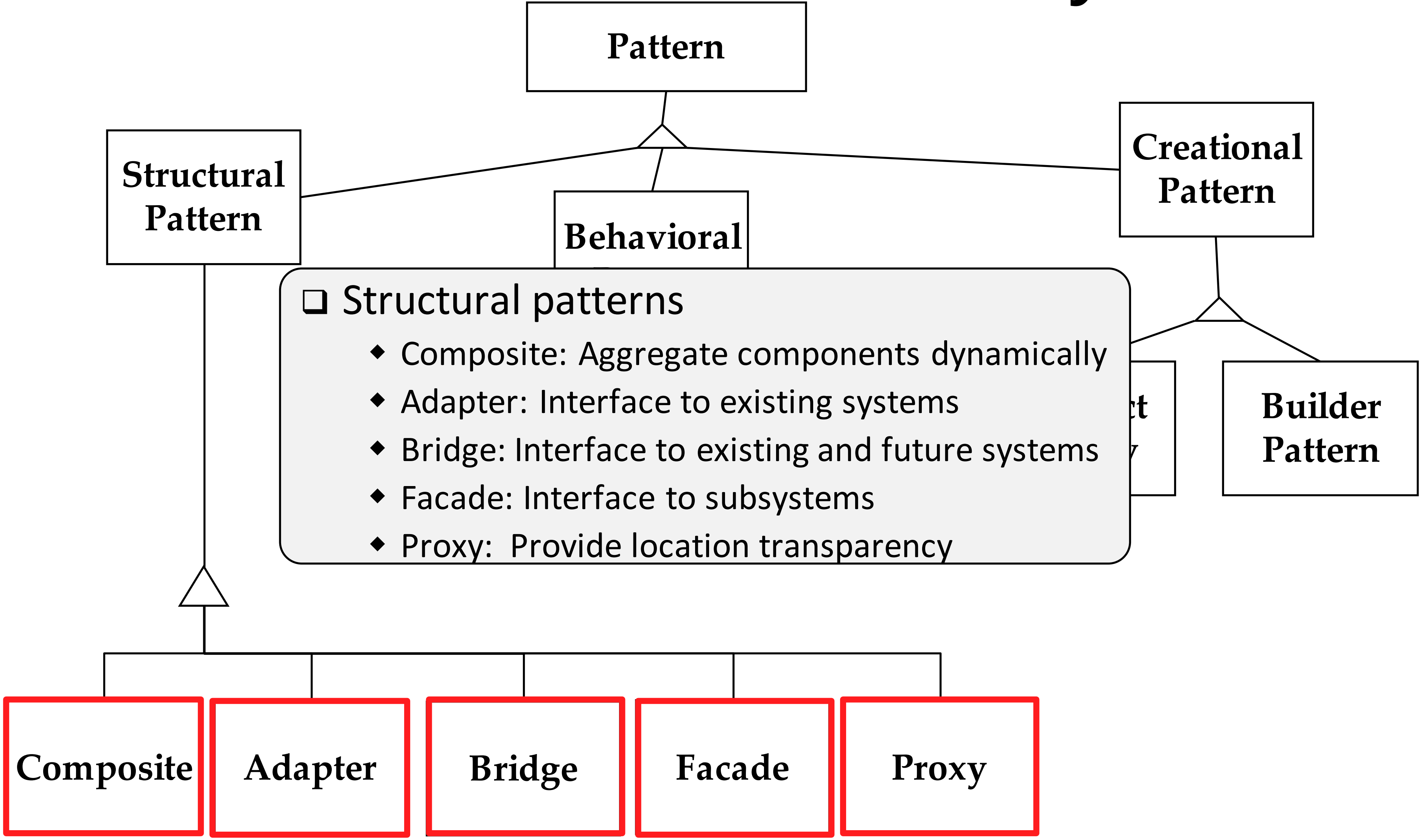
Towards a Pattern Taxonomy

- Structural Patterns
  - Reduce the coupling between two or more classes.
  - Encapsulate complex structures.
- Behavioral Patterns
  - Characterize complex control flows that are difficult to follow at runtime.
- Creational Patterns
  - Provide a simple abstraction for a complex instantiation process, when creating and composing objects.

A Pattern Taxonomy



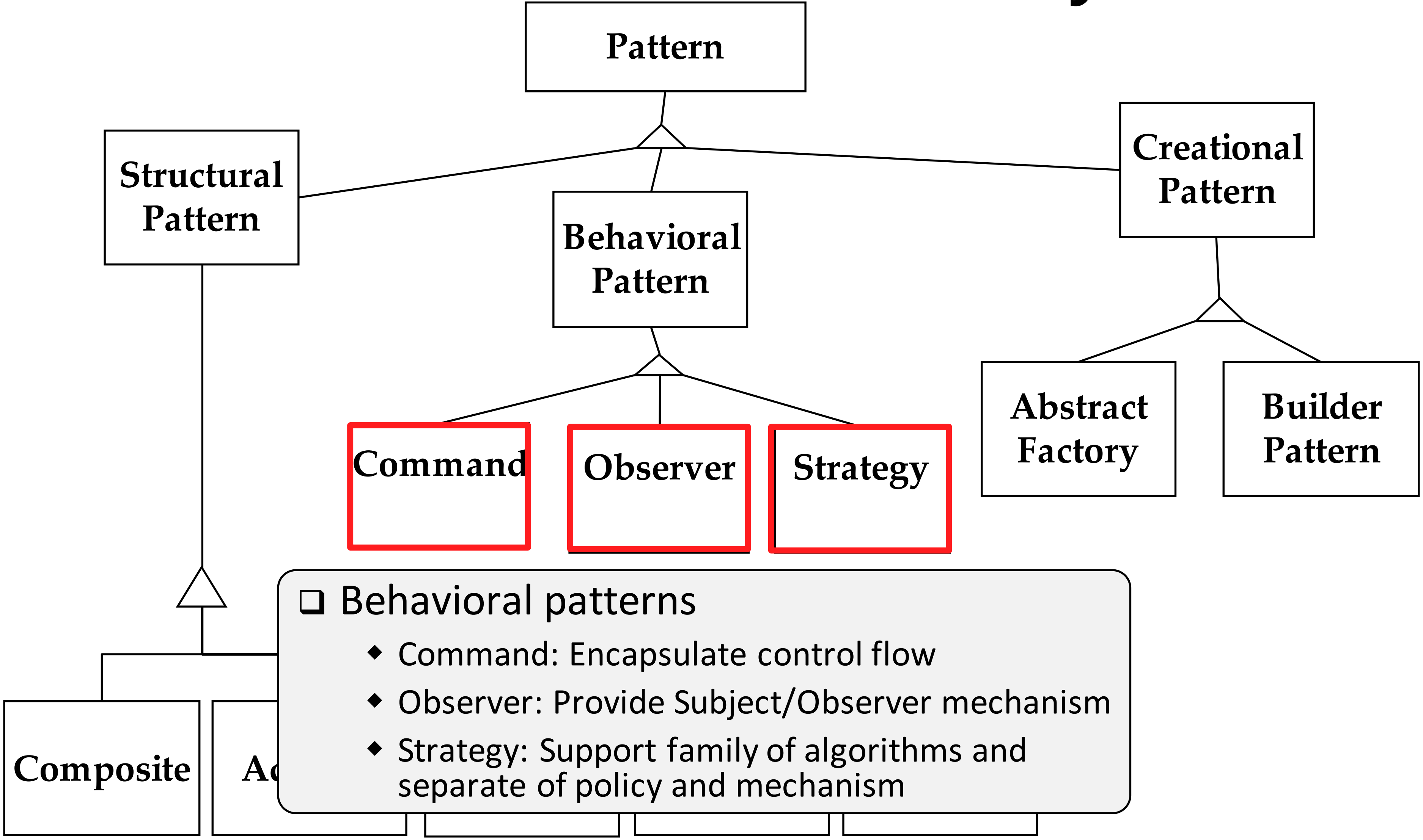
A Pattern Taxonomy



❑ Structural patterns

- Composite: Aggregate components dynamically
- Adapter: Interface to existing systems
- Bridge: Interface to existing and future systems
- Facade: Interface to subsystems
- Proxy: Provide location transparency

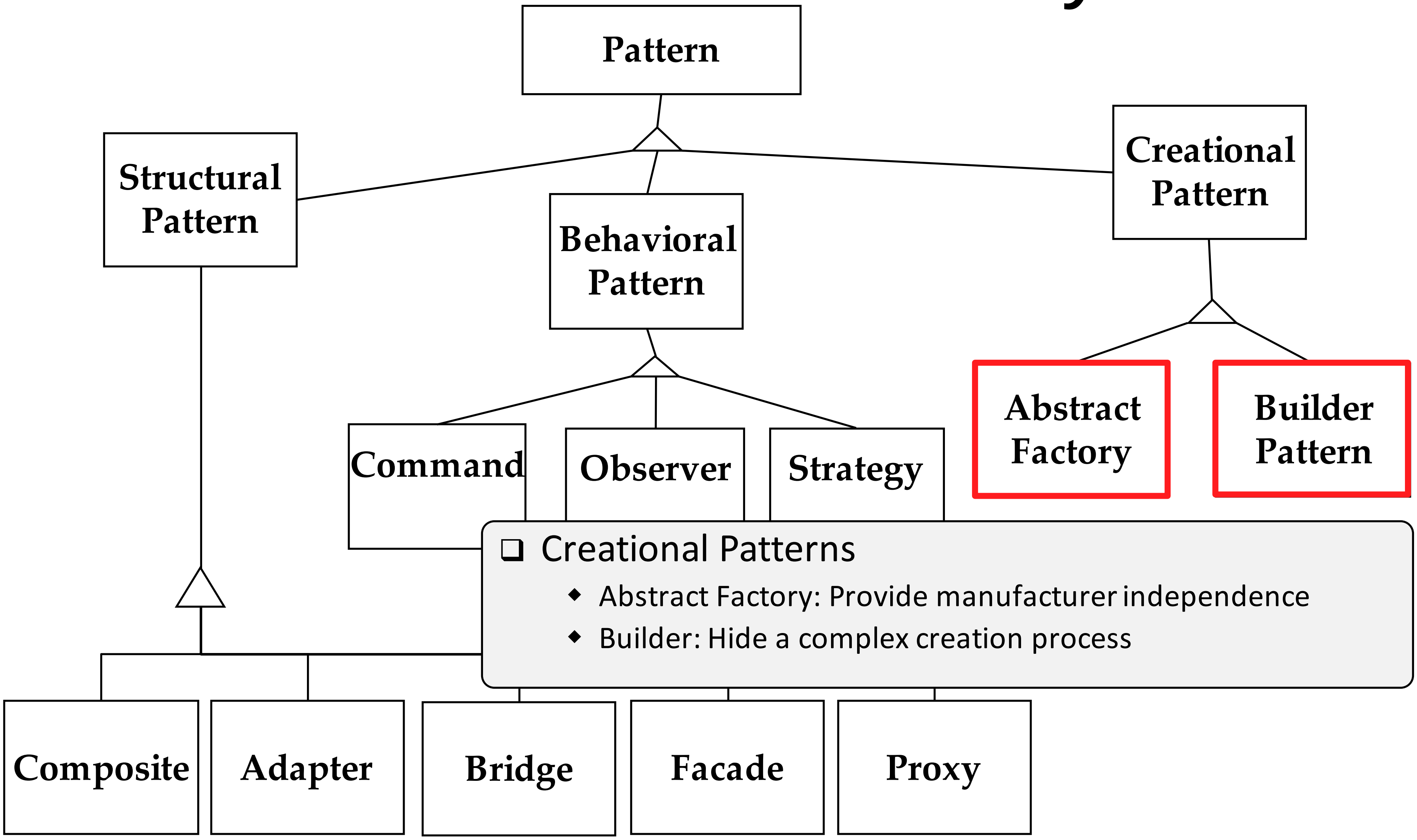
A Pattern Taxonomy



❑ Behavioral patterns

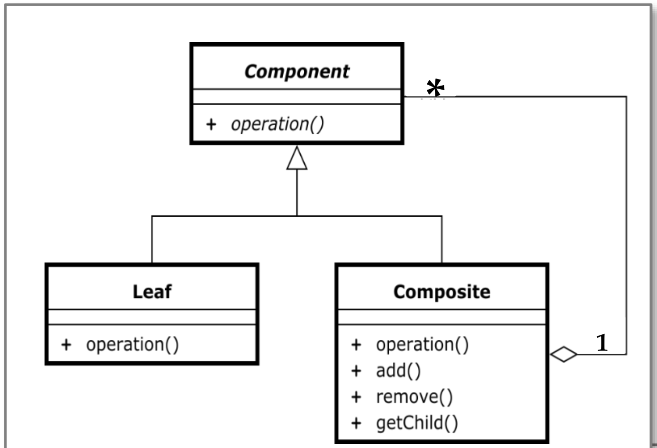
- Command: Encapsulate control flow
- Observer: Provide Subject/Observer mechanism
- Strategy: Support family of algorithms and separate of policy and mechanism

A Pattern Taxonomy

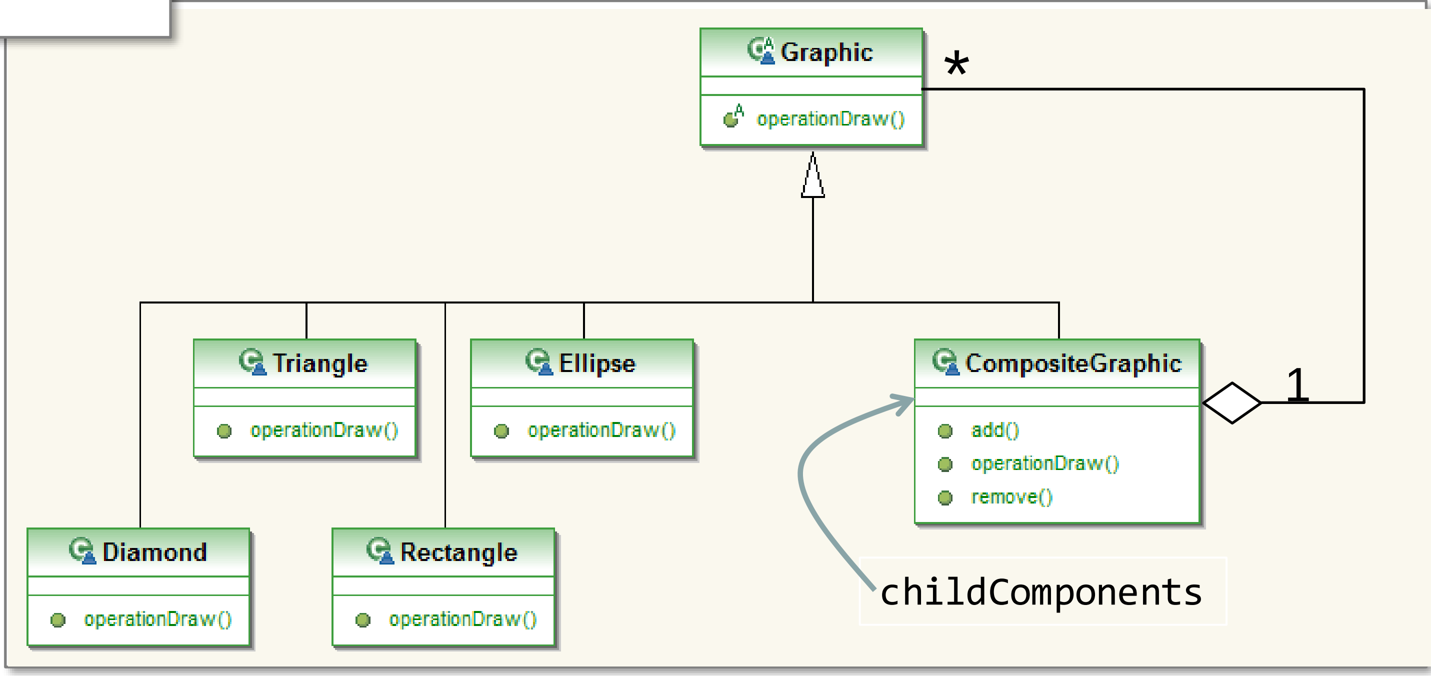


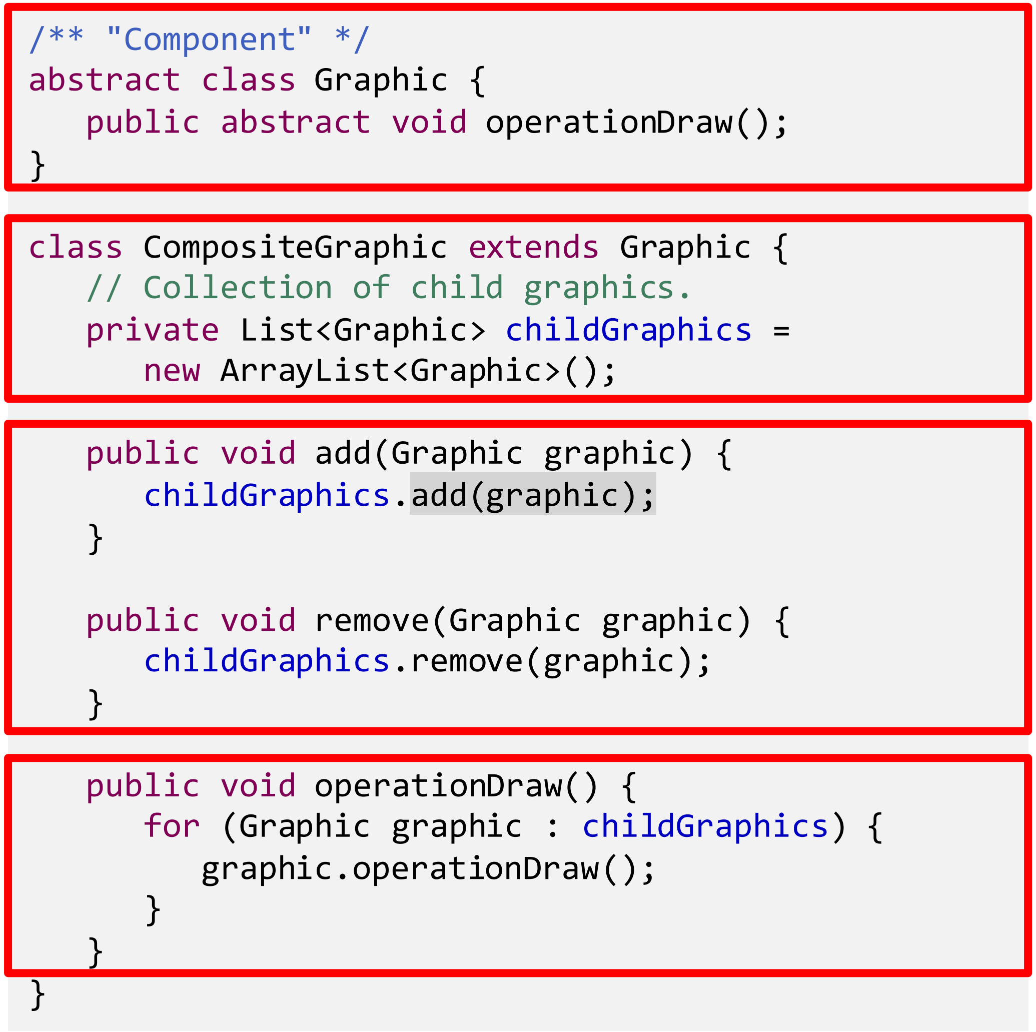
- ❑ Creational Patterns
  - Abstract Factory: Provide manufacturer independence
  - Builder: Hide a complex creation process

Composite Pattern

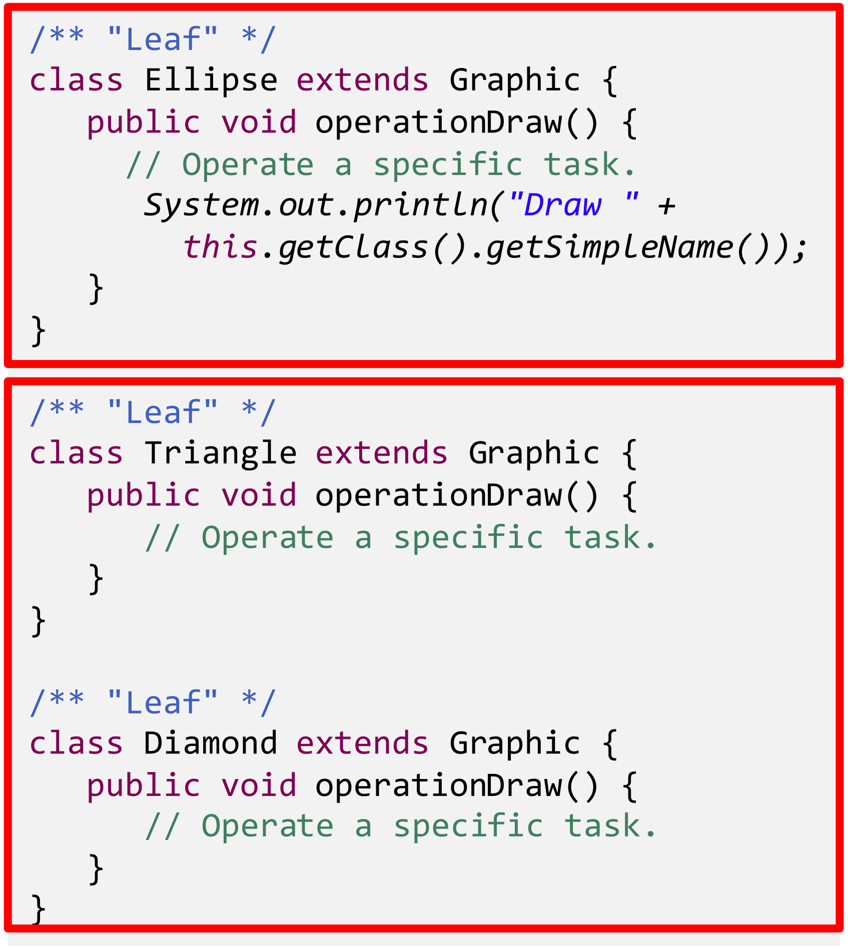


- Composite should be used when clients ignore the difference between compositions of objects and individual objects.

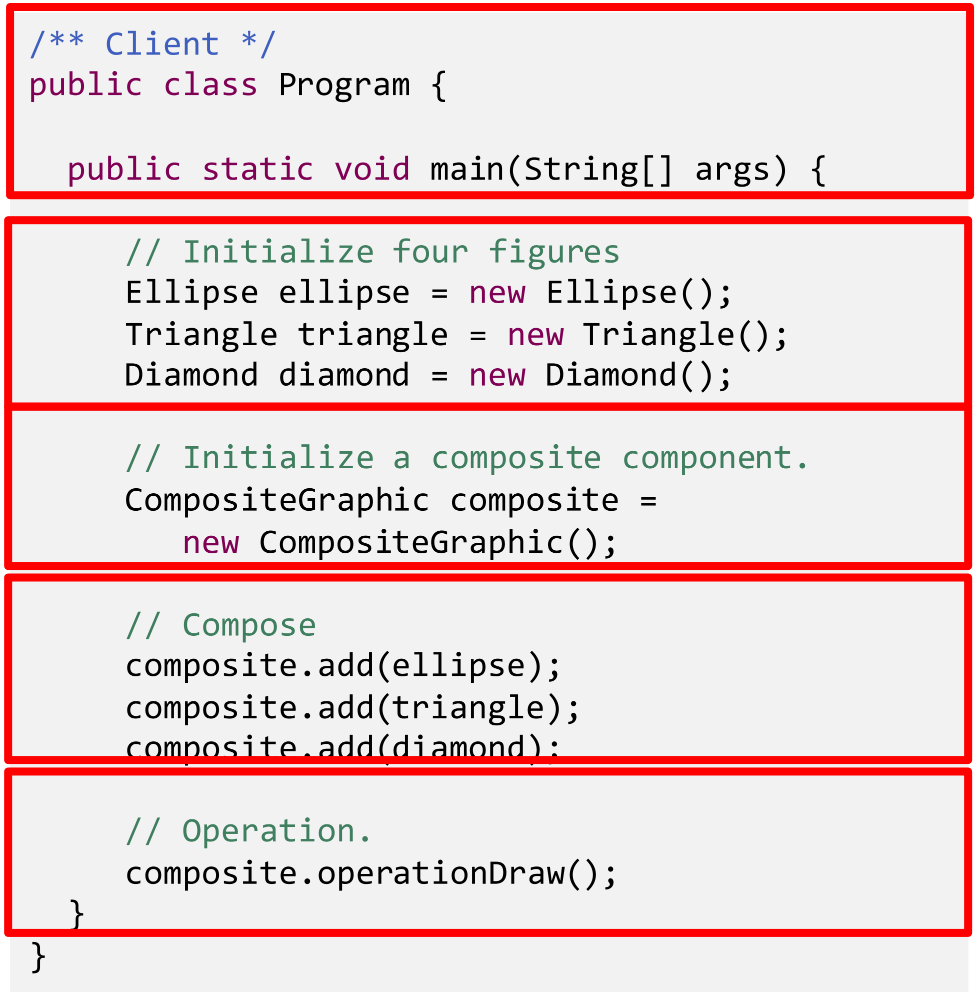




```java
/** "Component" */
abstract class Graphic {
   public abstract void operationDraw();
}

class CompositeGraphic extends Graphic {
   // Collection of child graphics.
   private List<Graphic> childGraphics =
      new ArrayList<Graphic>();

   public void add(Graphic graphic) {
      childGraphics.add(graphic);
   }

   public void remove(Graphic graphic) {
      childGraphics.remove(graphic);
   }

   public void operationDraw() {
      for (Graphic graphic : childGraphics) {
         graphic.operationDraw();
      }
   }
}
```

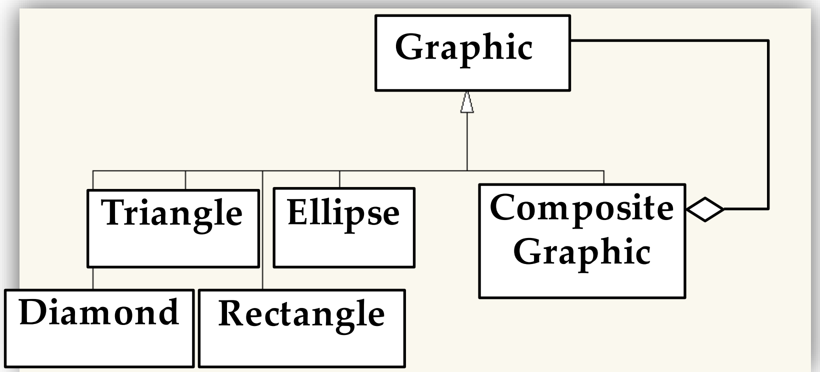


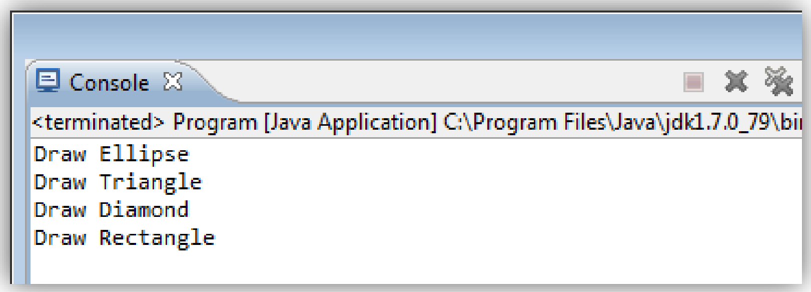
```java
/** "Leaf" */
class Ellipse extends Graphic {
   public void operationDraw() {
     // Operate a specific task.
      System.out.println("Draw " +
        this.getClass().getSimpleName());
   }
}

/** "Leaf" */
class Triangle extends Graphic {
   public void operationDraw() {
      // Operate a specific task.
   }
}

/** "Leaf" */
class Diamond extends Graphic {
   public void operationDraw() {
      // Operate a specific task.
   }
}
```



```
/** Client */
public class Program {

  public static void main(String[] args) {

     // Initialize four figures
     Ellipse ellipse = new Ellipse();
     Triangle triangle = new Triangle();
     Diamond diamond = new Diamond();

     // Initialize a composite component.
     CompositeGraphic composite =
        new CompositeGraphic();

     // Compose
     composite.add(ellipse);
     composite.add(triangle);
     composite.add(diamond);

     // Operation.
     composite.operationDraw();
  }
}
```

- A programmer uses multiple objects, such as Ellipse, Triangle, Diamond, and Rectangle, in the same way.
- The objects have nearly identical code to handle each of them
- The composite pattern is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.

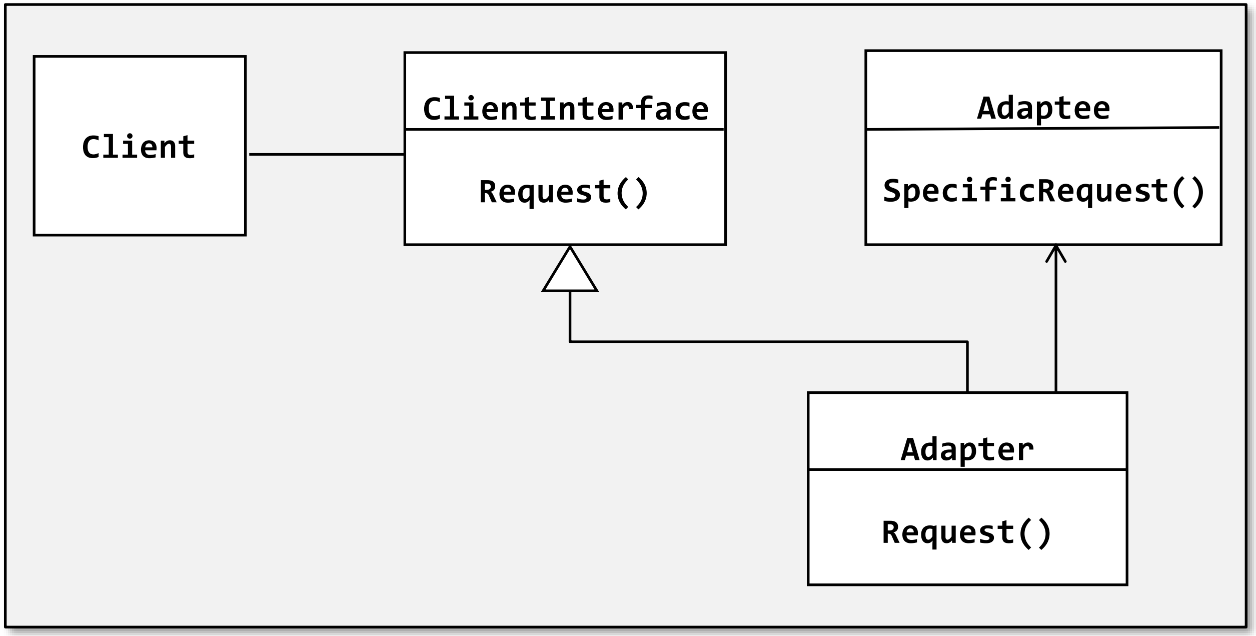




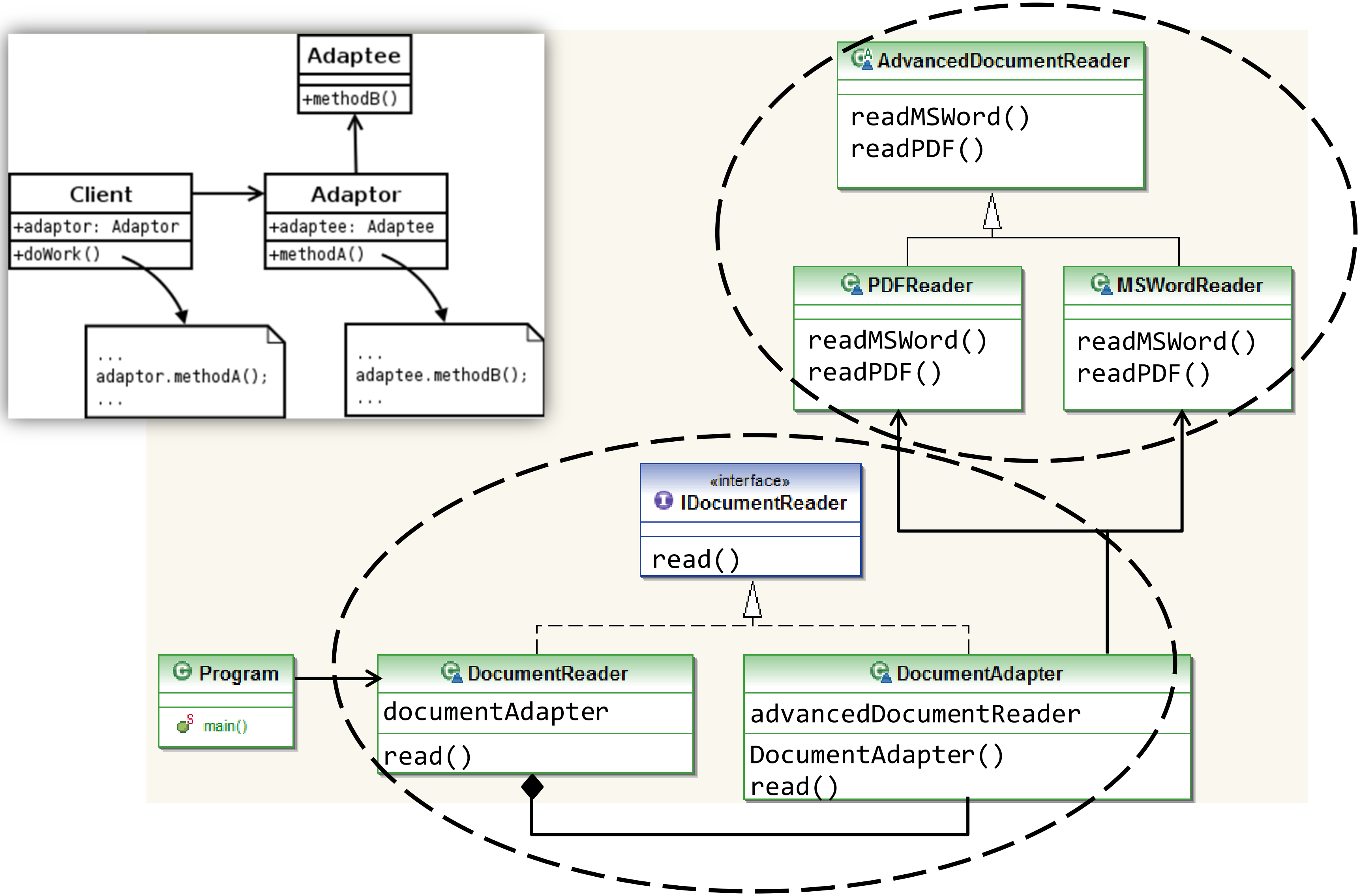
```
<terminated> Program [Java Application] C:\Program Files\Java\jdk1.7.0_79\bi
Draw Ellipse
Draw Triangle
Draw Diamond
Draw Rectangle
```

Adapter Pattern

- Convert the interface of a legacy class into a different interface expected by the client, so that the client and the legacy class can work together without changes.
- Delegation is used to bind an **Adapter** and an **Adaptee.**
- Interface inheritance is used to specify the interface of the **Adapter** class.
- Used to make existing classes, such as **ClientInterface** and **Adaptee**, work with others without modifying their source code (or small modification).



A Document Reader Example Using Adapter Pattern



```
...
adaptor.methodA();
...
```

```
...
adaptee.methodB();
...
```

```java
public class Program {
    public static void main(String[] args) {
        IDocumentReader docReader =
           new DocumentReader();
        docReader.read("mypdf.pdf");
        docReader.read("mymsword.docx");
        docReader.read("mytext.txt");
        docReader.read("mypowerpoint.pptx");
    }
}
```

```java
interface IDocumentReader {
  public abstract
    void read(String fileName);
}
```

```java
class DocumentReader implements IDocumentReader {
    DocumentAdapter documentAdapter;
    public void read(String fileName) {
        // documentAdapter provides support to read other file formats.
        if (fileName.endsWith(".pdf") || fileName.endsWith(".docx")) {
            documentAdapter = new DocumentAdapter(fileName);
            documentAdapter.read(fileName);
        }
        // use a built-in text editor supporting to read text files.
        else if (fileName.endsWith(".txt")) {
            System.out.println("Reading text file: " + fileName);
        }
        else {
            System.out.println("Invalid file: " + fileName);
        }
    }
}
```

```java
class DocumentAdapter implements IDocumentReader {
  AdvancedDocumentReader advancedDocumentReader;

  public DocumentAdapter(String fileName) {
    if (fileName.endsWith(".pdf")) {
      advancedDocumentReader = new PDFReader();
    } else if (fileName.endsWith(".docx")) {
      advancedDocumentReader = new MSWordReader();
    }
  }

  public void read(String fileName) {
    if (fileName.endsWith(".pdf")) {
      advancedDocumentReader.readPDF(fileName);
    } else if (fileName.endsWith(".docx")) {
      advancedDocumentReader.readMSWord(fileName);
    }
  }
}
```

```java
abstract class AdvancedDocumentReader {
  public abstract void readPDF(String fileName);
  public abstract void readMSWord(String fileName);
}
```

```java
// Adaptee
class PDFReader
  extends AdvancedDocumentReader {

  public void readPDF(String fileName) {
    System.out.println(
      "Reading PDF file: " + fileName);
  }

  public void readMSWord(
                      String fileName) {
    // do nothing
  }
}
```

```java
// Adaptee
class MSWordReader
  extends AdvancedDocumentReader {

  public void readMSWord(
                      String fileName) {
    System.out.println(
      "Reading MSWord file: " + fileName);
  }

  public void readPDF(String fileName) {
    // do nothing
  }
}
```

```
class DocumentAdapter implements IDocumentReader {
  AdvancedDocumentReader advancedDocumentReader;

  public void read(String fileName) {
    if (fileName.endsWith(".pdf")) {
      advancedDocumentReader.readPDF(fileName);
    }
  }
}
```

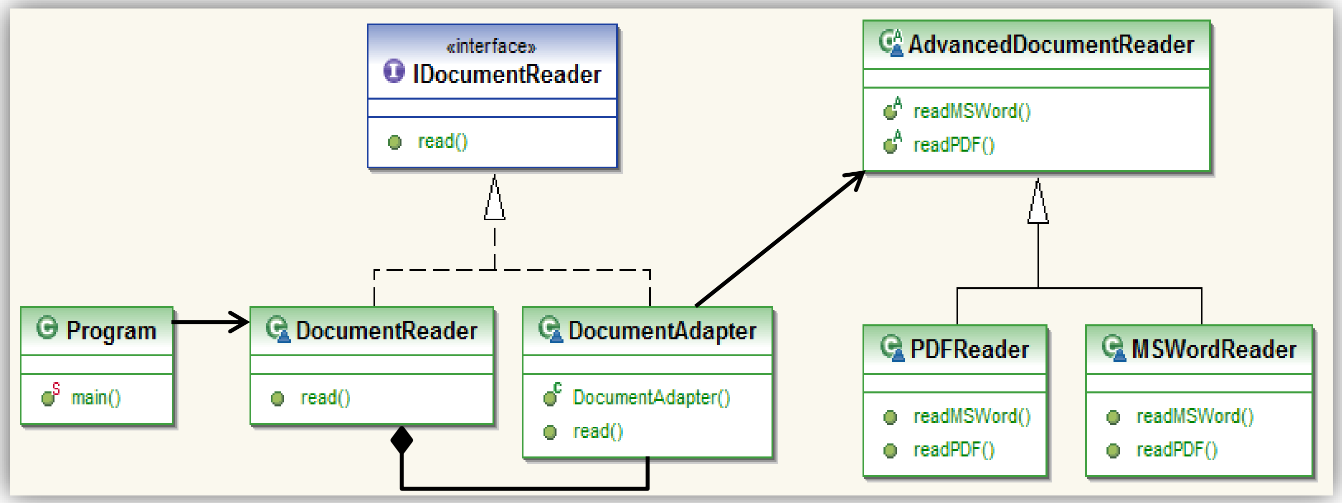
Wrap & Reuse

```
// Adaptee
class PDFReader
  extends AdvancedDocumentReader {

  public void readPDF(String fileName) {
    System.out.println(
      "Reading PDF file: " + fileName);
  }
}
```

```
public class Program {
      public static void main(String[] args) {
            IDocumentReader docReader =
               new DocumentReader();
            docReader.read("mypdf.pdf");
            docReader.read("mymsword.docx");
            docReader.read("mytext.txt");
            docReader.read("mypowerpoint.pptx");
      }
}
```

Call

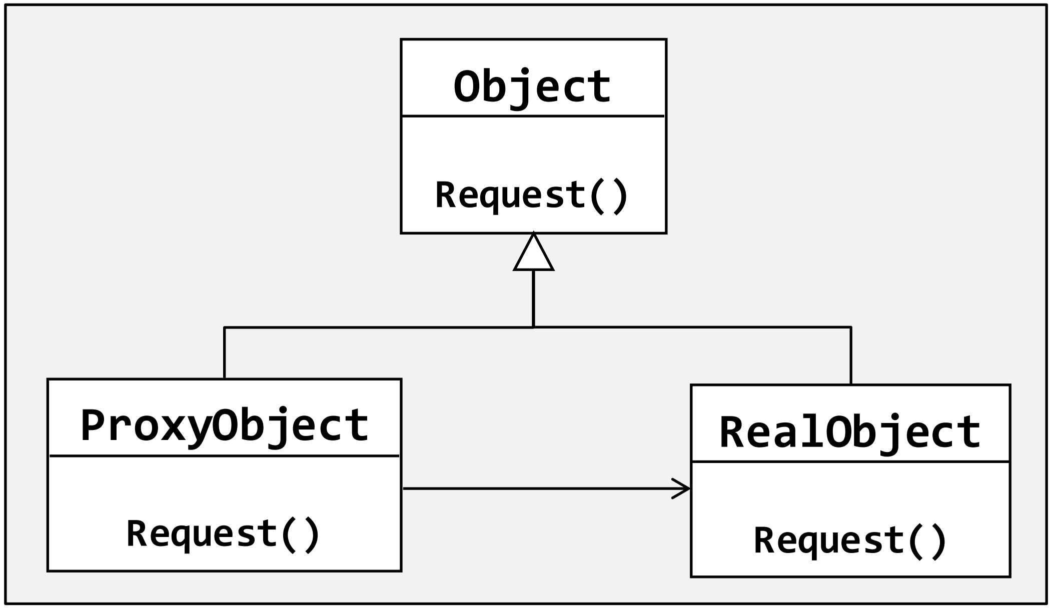
```
class DocumentReader implements IDocumentReader {
      DocumentAdapter documentAdapter;
      public void read(String fileName) {
            if (fileName.endsWith(".pdf") || ..) {
                  // Reused features.
                  documentAdapter.read(fileName);
            }
      }
}
```



- Reuse an off the shelf component that offers compelling functionality.
- Support compatibility with reusable features.

Proxy Pattern

- The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface. The ProxyObject stores a subset of the attributes of the RealObject.
- The ProxyObject handles certain requests completely (e.g., investigating the size of an image), where others are delegated to the RealObject. After delegation, the RealObject is created and loaded in memory.

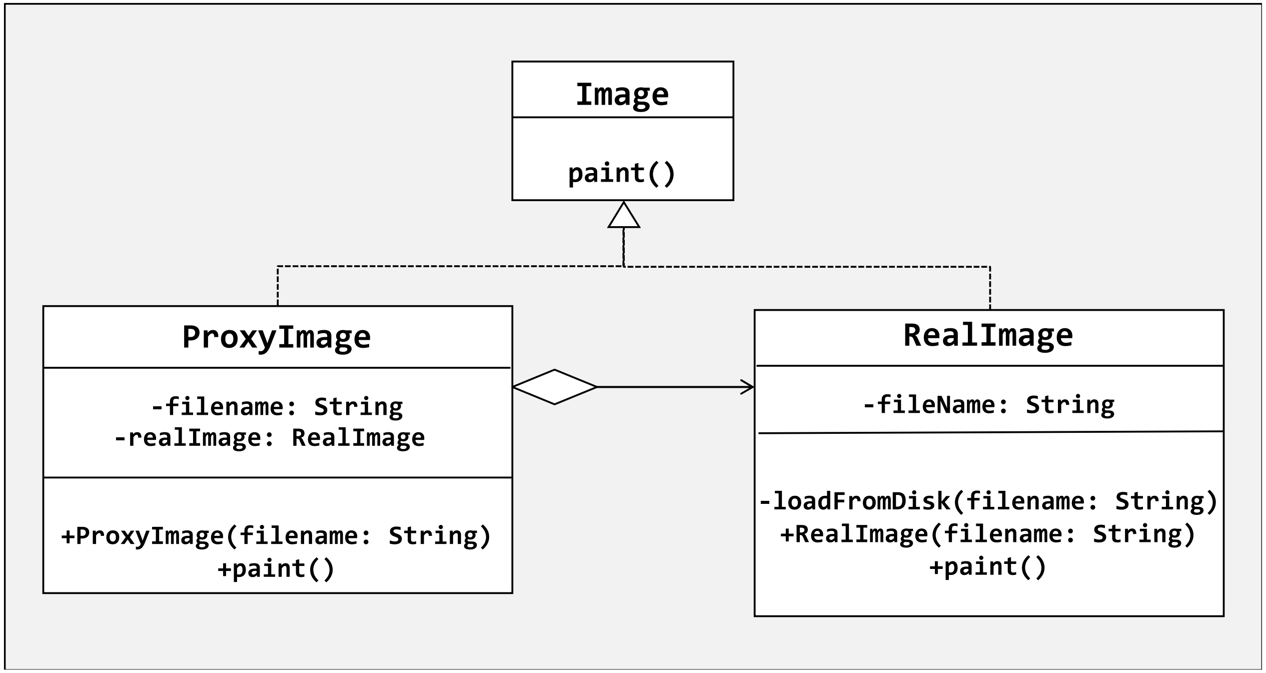


Proxy Applicability

- Remote Proxy
  - A local object which is accessible to an associated object in a different address space (e.g., distributed systems).
- Virtual Proxy
  - A skeleton object which represents a complex or heavy object (e.g., large, high-resolution display systems).
- Protection Proxy
  - Proxy provides access control to the real object.

Lab 1

- Write an interface and classes (ProxyImage and RealImage) using the following class diagram



Image

paint()

ProxyImage

-filename: String
-realImage: RealImage

+ProxyImage(filename: String)
+paint()

RealImage

-fileName: String

-loadFromDisk(filename: String)
+RealImage(filename: String)
+paint()

Observer Pattern

- Defines a “one-to-many” dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

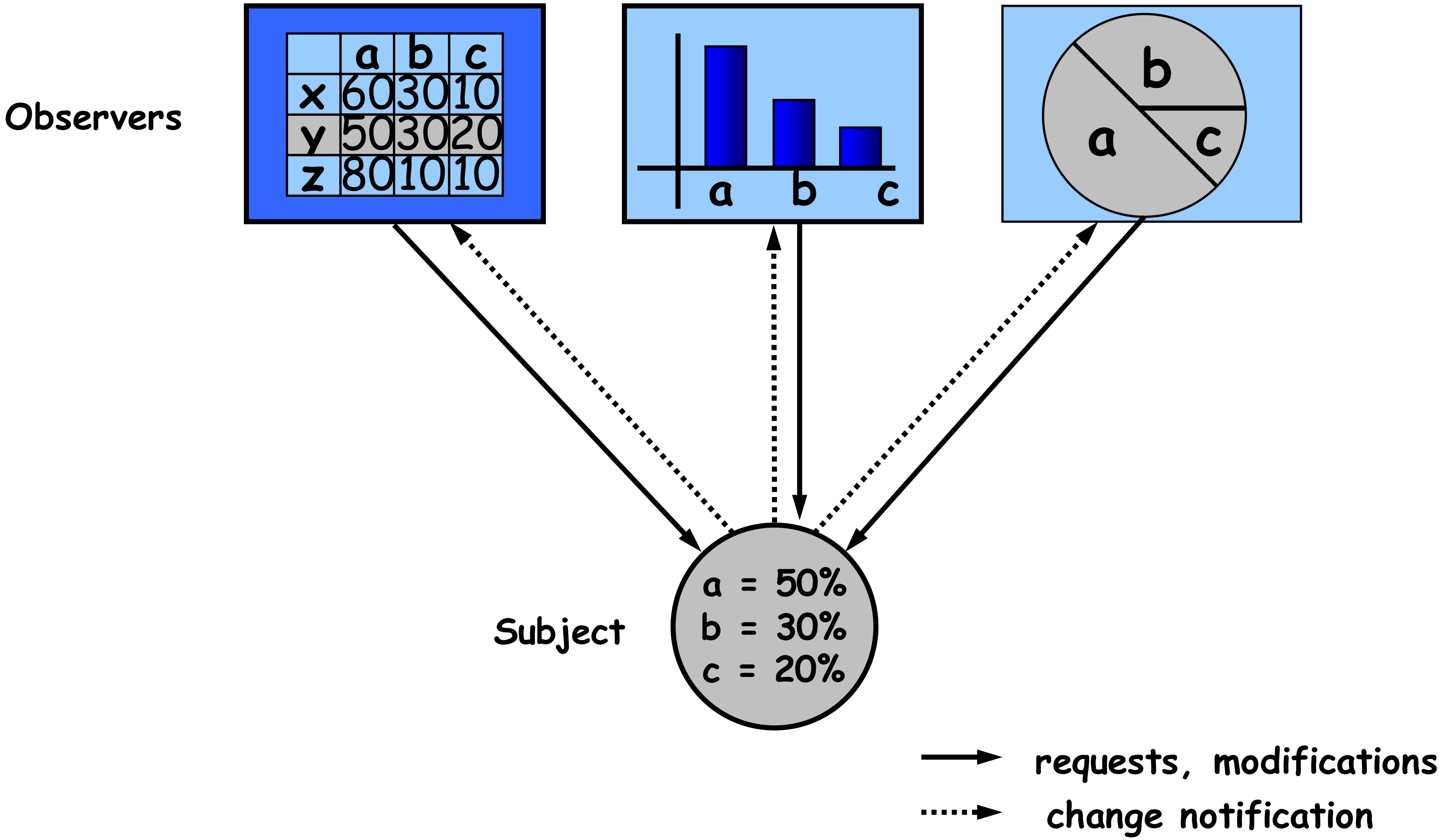
- a.k.a Dependence mechanism / publish-subscribe / broadcast / change-update

Subject & Observer

- Subject
  - – the object which will frequently change its state and upon which other objects depend

- Observer
  - – the object which depends on a subject and updates according to its subject's state.

Observer Pattern - Example



Observers

	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Subject

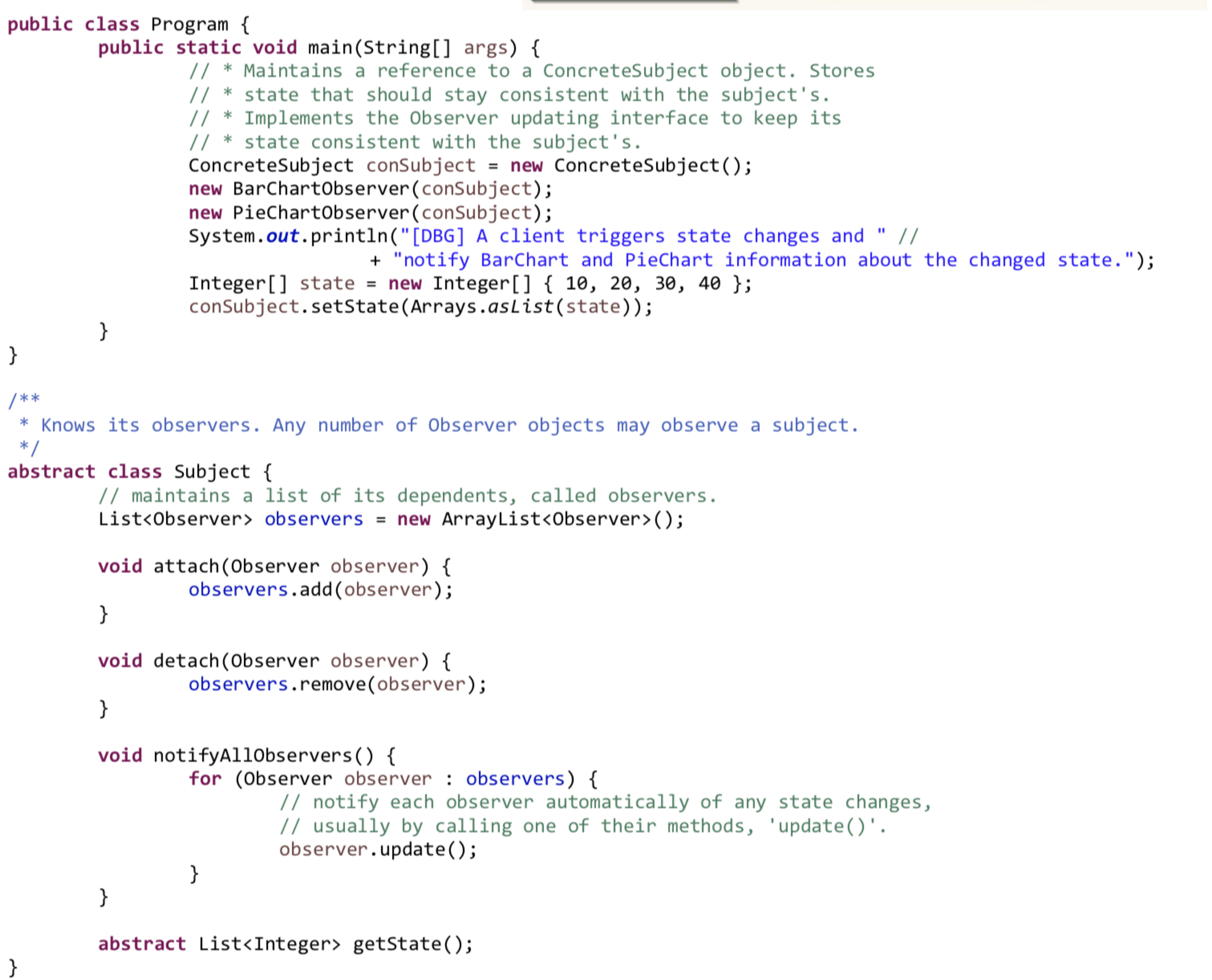
a = 50%
b = 30%
c = 20%

requests, modifications

change notification

Observer Pattern - Key Players

- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores “state of interest” to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface



```
public class Program {
        public static void main(String[] args) {
                // * Maintains a reference to a ConcreteSubject object. Stores
                // * state that should stay consistent with the subject's.
                // * Implements the Observer updating interface to keep its
                // * state consistent with the subject's.
                ConcreteSubject conSubject = new ConcreteSubject();
                new BarChartObserver(conSubject);
                new PieChartObserver(conSubject);
                System.out.println("[DBG] A client triggers state changes and " //
                                 + "notify BarChart and PieChart information about the changed state.");
                Integer[] state = new Integer[] { 10, 20, 30, 40 };
                conSubject.setState(Arrays.asList(state));
        }
}

/**
 * Knows its observers. Any number of Observer objects may observe a subject.
 */
abstract class Subject {
        // maintains a list of its dependents, called observers.
        List<Observer> observers = new ArrayList<Observer>();

        void attach(Observer observer) {
                observers.add(observer);
        }

        void detach(Observer observer) {
                observers.remove(observer);
        }

        void notifyAllObservers() {
                for (Observer observer : observers) {
                        // notify each observer automatically of any state changes,
                        // usually by calling one of their methods, 'update()'.
                        observer.update();
                }
        }

        abstract List<Integer> getState();
}
```

```
/**
 * Stores state of interest to ConcreteObserver objects. Sends a notification to
 * its observers when its state change.
 */
class ConcreteSubject extends Subject {
        List<Integer> state;

        @Override
        List<Integer> getState() {
                return state;
        }

        void setState(List<Integer> state) {
                this.state = state;
                notifyAllObservers();
        }
}

/**
 * Concrete Observer implementations. Maintains a reference to a ConcreteSubject
 * object. Stores state that should stay consistent with the subject's.
 * Implements the Observer updating interface to keep its state consistent with
 * the subject's.
 */
class BarChartObserver extends Observer {
        ConcreteSubject conSubject;
        List<Integer> state;

        public BarChartObserver(ConcreteSubject subject) {
                this.conSubject = subject;
                subject.attach(this);
        }

        @Override
        public void update() {
                state = this.conSubject.getState();
                for (Integer i : state) {
                        System.out.println("\tBarChart: " + i);
                }
        }
}
```

Lab 2

- Draw a class diagram using the provided source code

Factory Pattern

- Factory pattern defines an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory patterns lets a class defer instantiation to subclasses.
- Frameworks use abstract classes to define and maintain relationships between objects.
- A framework is often responsible for creating these objects as well.

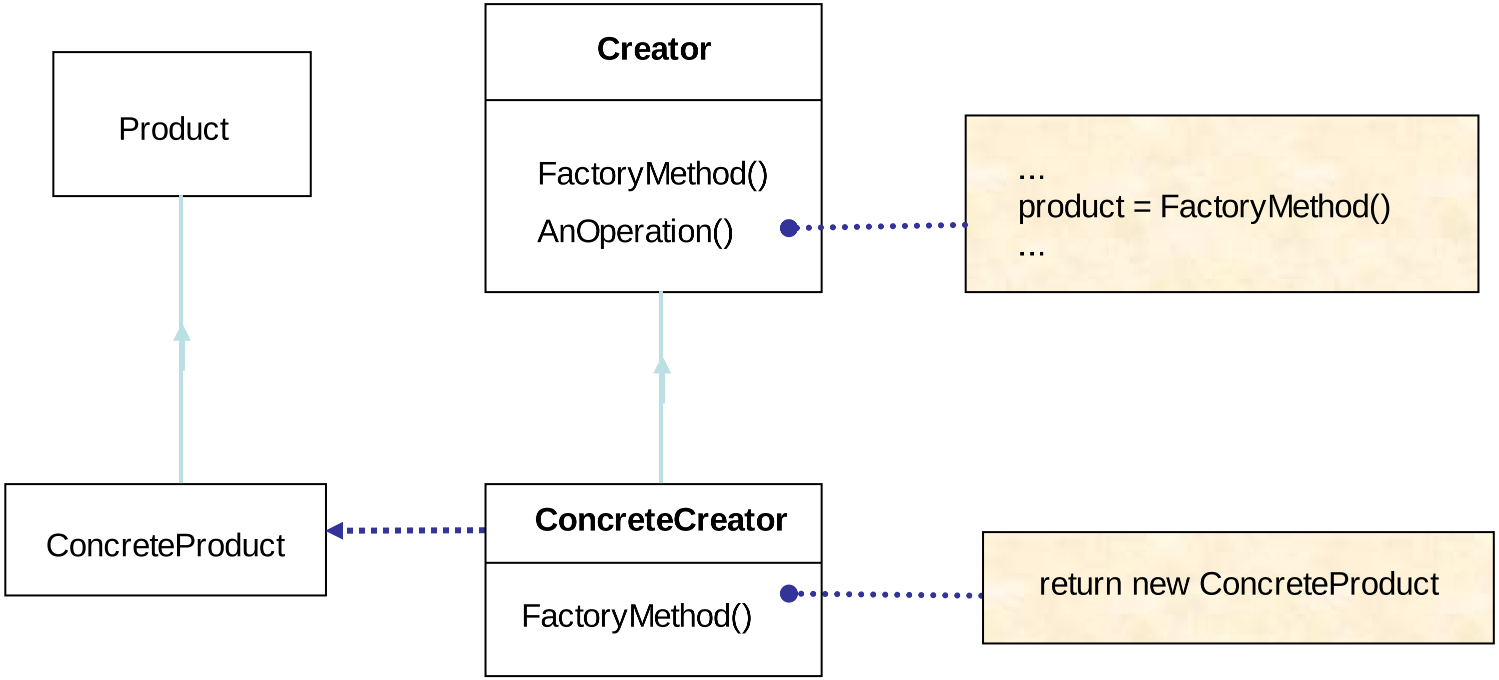
Factory Pattern

- Example
  - Car Factory produces different Car objects
  - Original
    - Different classes implement Car interface
    - Directly instantiate car objects
    - Need to modify client to change cars
  - Using pattern
    - Use CarFactory class to produce car objects
    - Can change cars by changing CarFactory

Factory Method Pattern

- Factory pattern defines an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory patterns lets a class defer instantiation to subclasses.
- Frameworks use abstract classes to define and maintain relationships between objects.
- A framework is often responsible for creating these objects as well.

Factory Method Pattern



Product

Creator

FactoryMethod()

AnOperation()

...
product = FactoryMethod()
...

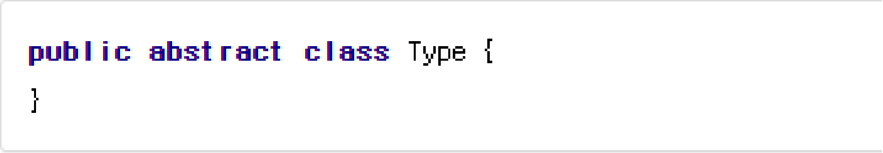
ConcreteProduct

ConcreteCreator

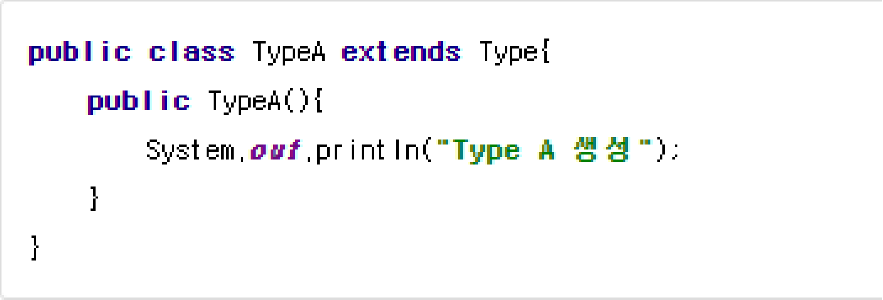
FactoryMethod()

return new ConcreteProduct

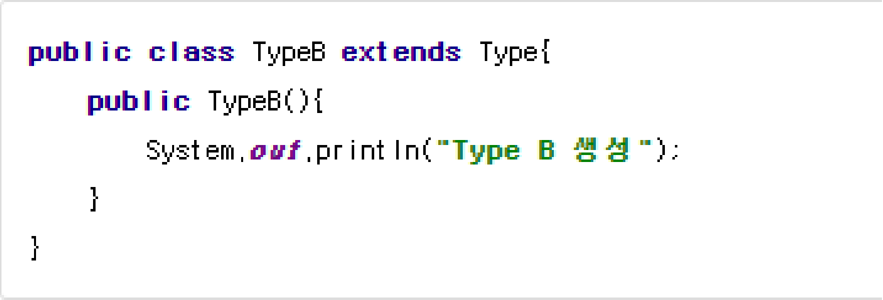
Factory Method Example



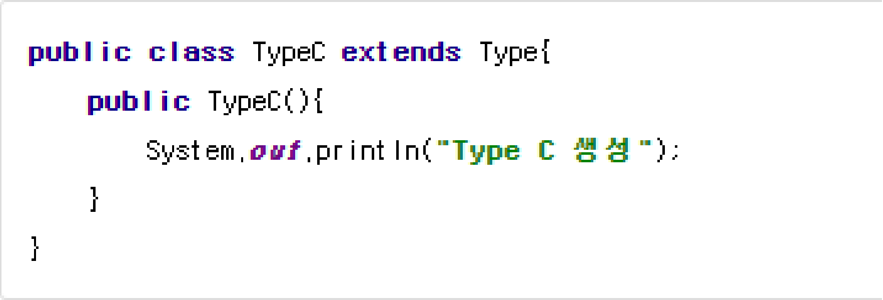
```java
public abstract class Type {
}
```



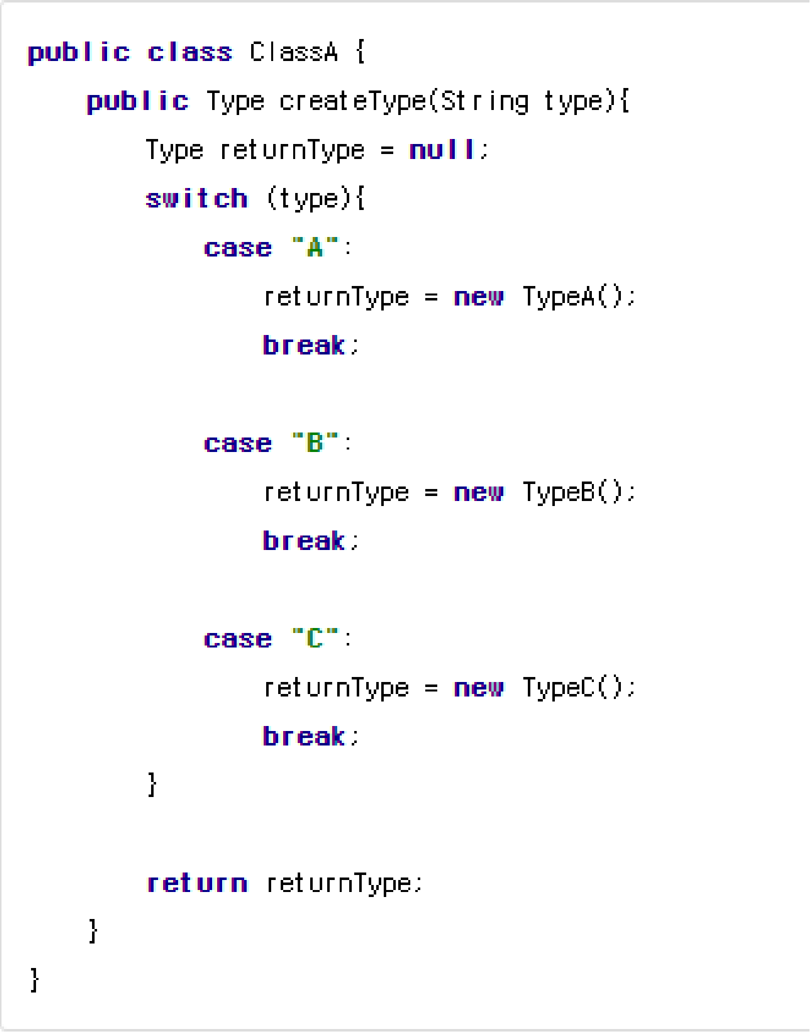
```java
public class TypeA extends Type{
    public TypeA(){
        System.out.println("Type A 생성");
    }
}
```



```java
public class TypeB extends Type{
    public TypeB(){
        System.out.println("Type B 생성");
    }
}
```

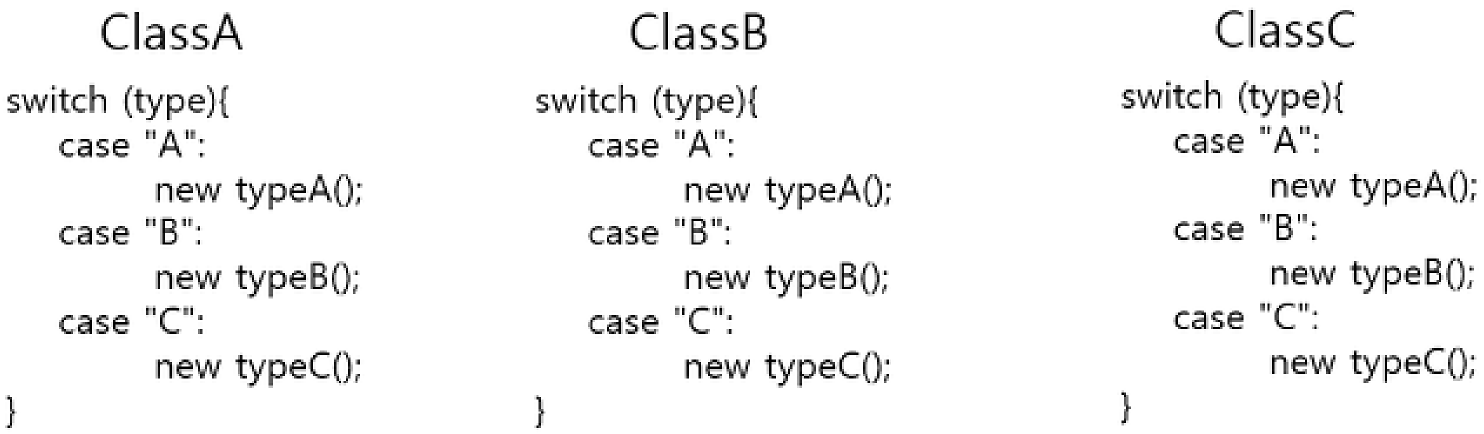


```java
public class TypeC extends Type{
    public TypeC(){
        System.out.println("Type C 생성");
    }
}
```



```java
public class ClassA {
    public Type createType(String type){
        Type returnType = null;
        switch (type){
            case "A":
                returnType = new TypeA();
                break;

            case "B":
                returnType = new TypeB();
                break;

            case "C":
                returnType = new TypeC();
                break;
        }

        return returnType;
    }
}
```

Factory Method Example



ClassA

```
switch (type){
    case "A":
          new typeA();
    case "B":
          new typeB();
    case "C":
          new typeC();
}
```

ClassB

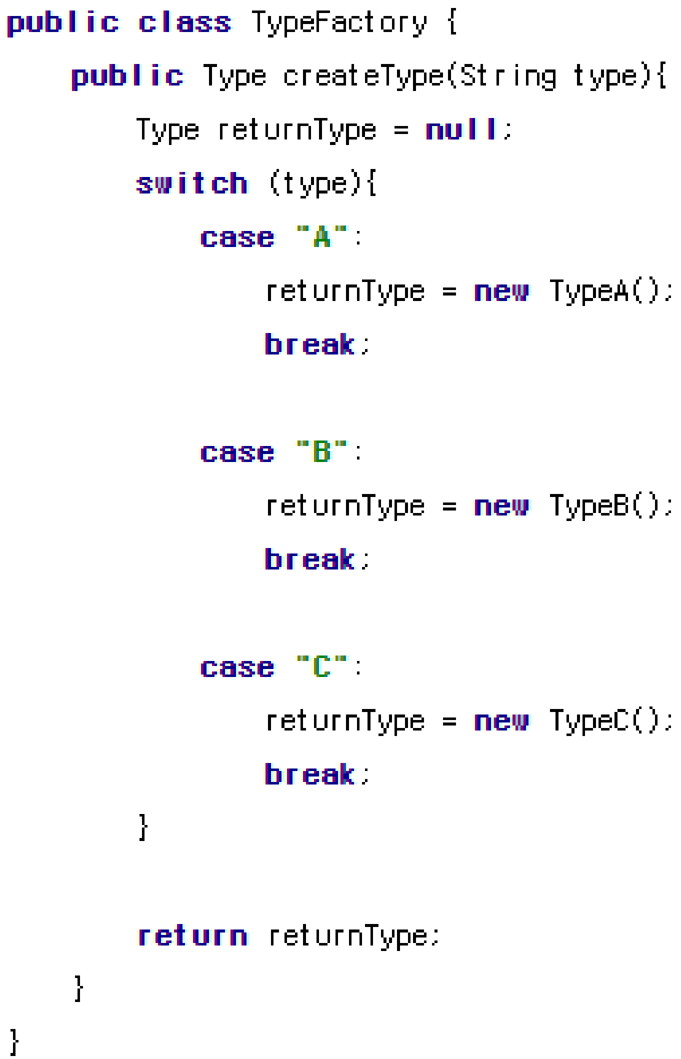
```
switch (type){
    case "A":
          new typeA();
    case "B":
          new typeB();
    case "C":
          new typeC();
}
```

ClassC

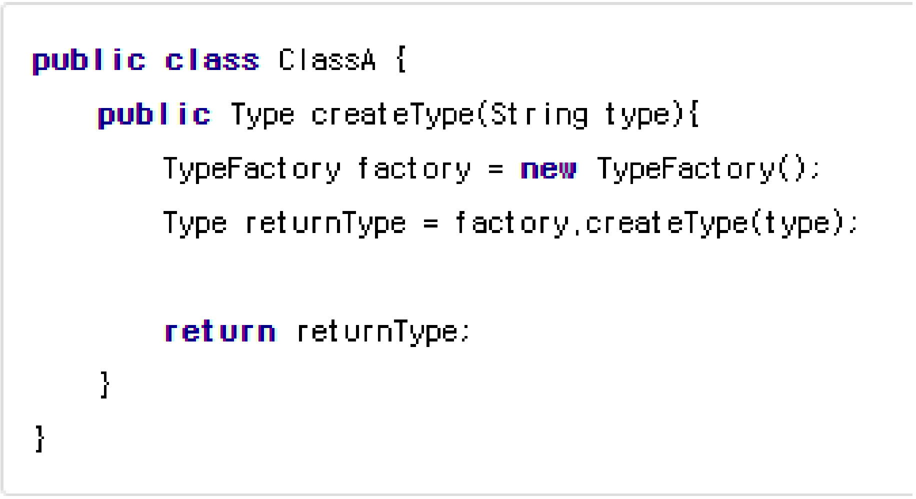
```
switch (type){
    case "A":
          new typeA();
    case "B":
          new typeB();
    case "C":
          new typeC();
}
```

- The problem is high coupling

Factory Method Example



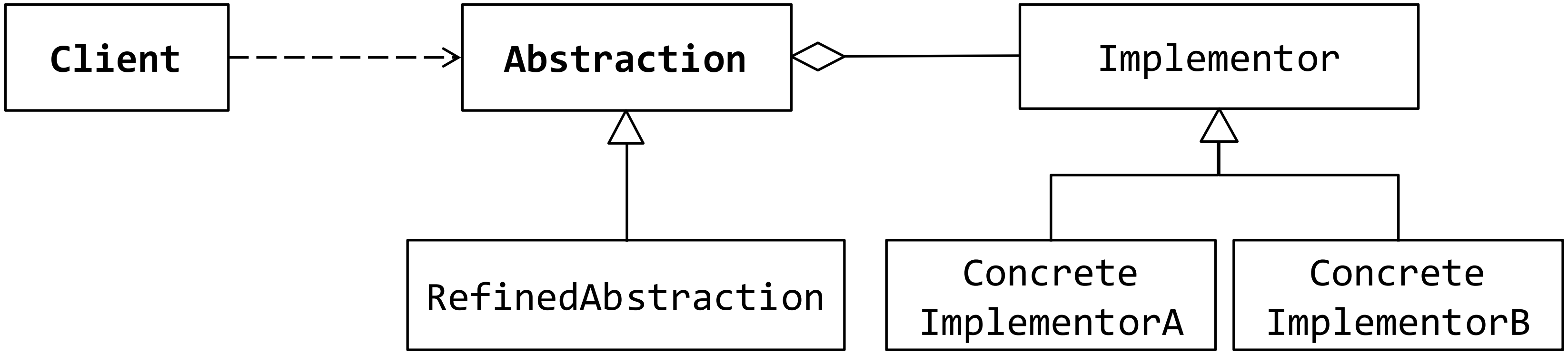
```
public class TypeFactory {
    public Type createType(String type){
        Type returnType = null;
        switch (type){
            case "A":
                returnType = new TypeA();
                break;

            case "B":
                returnType = new TypeB();
                break;

            case "C":
                returnType = new TypeC();
                break;
        }

        return returnType;
    }
}
```



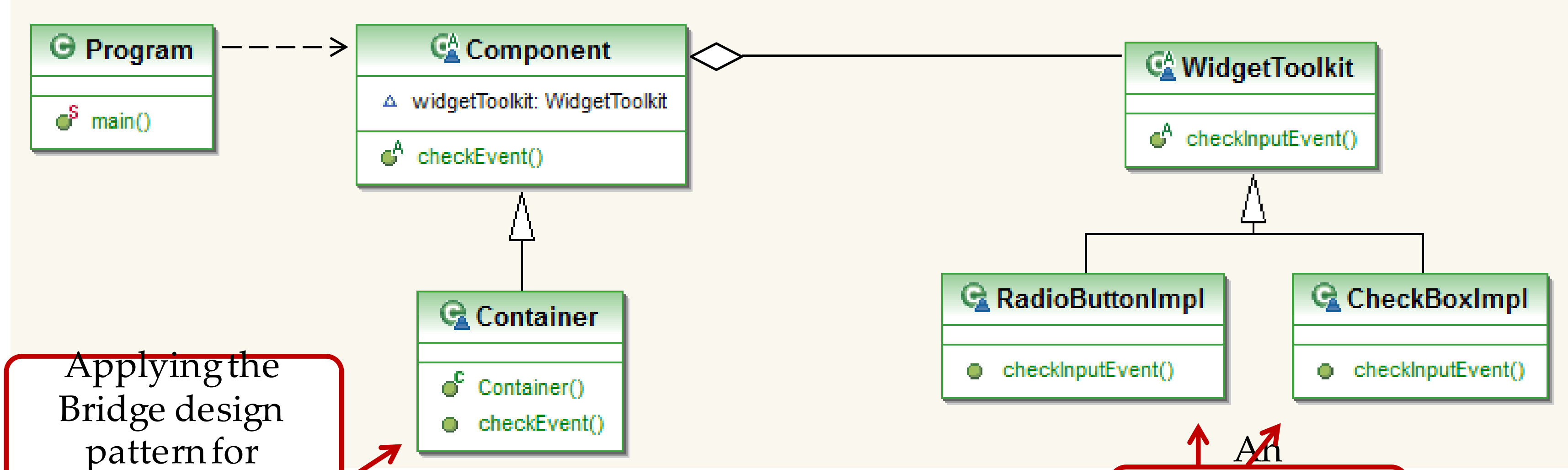
```
public class ClassA {
    public Type createType(String type){
        TypeFactory factory = new TypeFactory();
        Type returnType = factory.createType(type);

        return returnType;
    }
}
```

Bridge Pattern

- Decouple an interface from an implementation so that implementations can be, possibly substitute at runtime.
- The **Abstraction** class defines the interface visible to the client.
- The **Implementor** is an abstract class that declares the lower-level methods available to **Abstraction**.
- An **Abstraction** instance maintains a reference to its corresponding **Implementor** instance.
- **Abstraction** and **Implementor** can be refined independently.

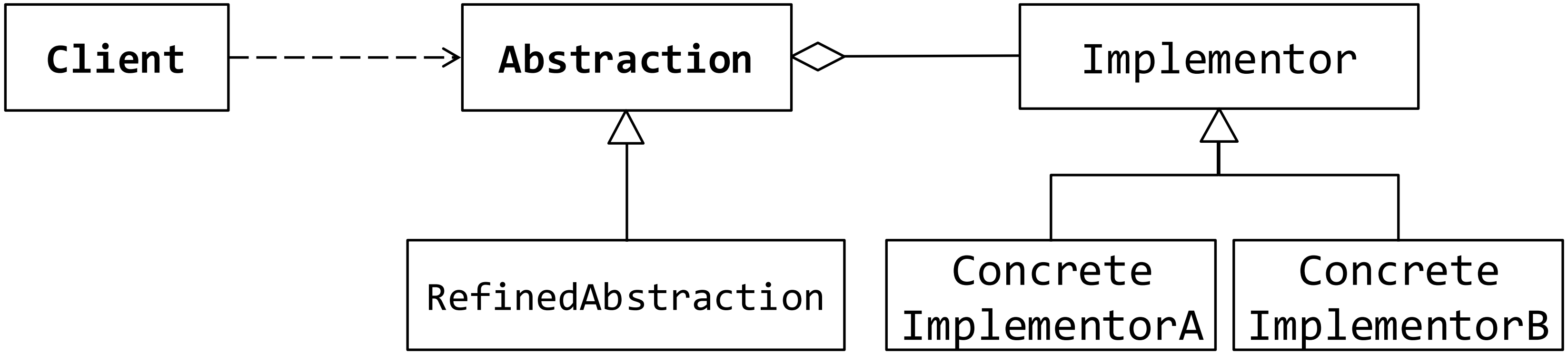


A Widget Example Using Bridge Pattern

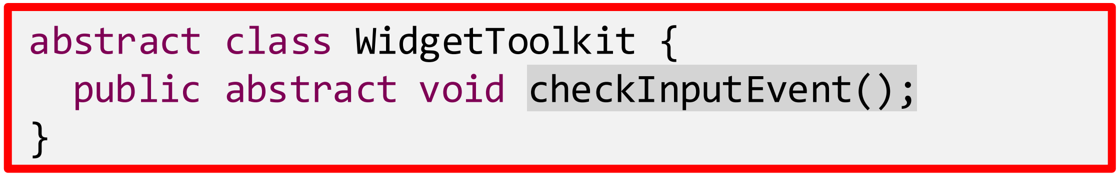


Applying the Bridge design pattern for abstracting event implementations

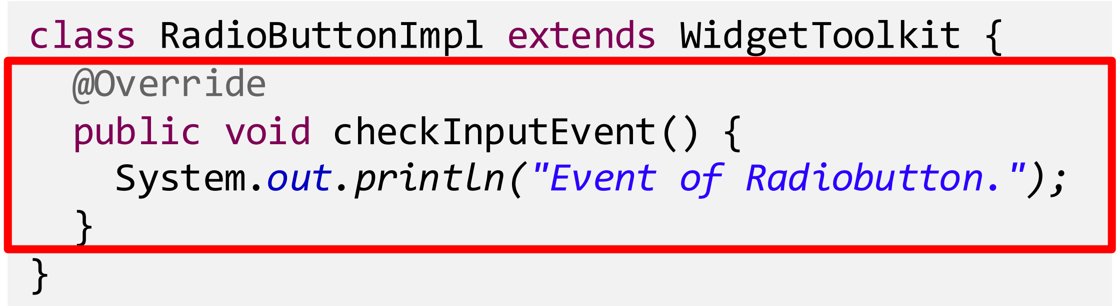
An independent implementation



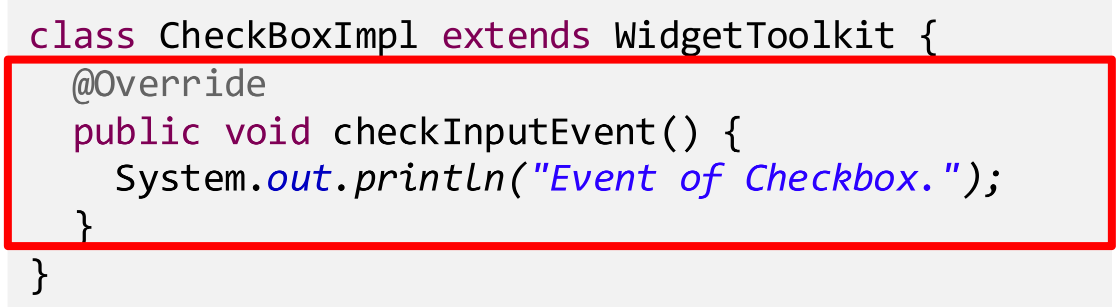
A Widget Example Using Bridge Pattern



```
abstract class WidgetToolkit {
  public abstract void checkInputEvent();
}
```



```
class RadioButtonImpl extends WidgetToolkit {
  @Override
  public void checkInputEvent() {
    System.out.println("Event of Radiobutton.");
  }
}
```



```
class CheckBoxImpl extends WidgetToolkit {
  @Override
  public void checkInputEvent() {
    System.out.println("Event of Checkbox.");
  }
}
```

A Widget Example Using Bridge Pattern

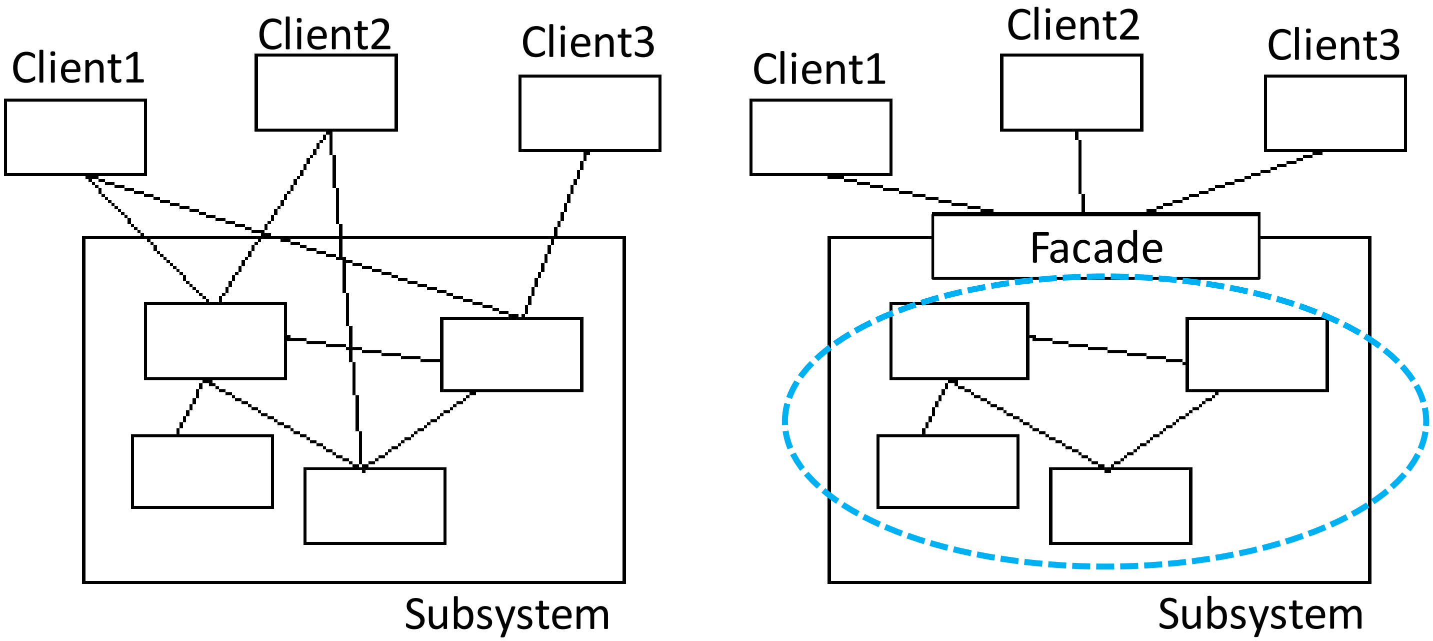
```
abstract class Component {
  WidgetToolkit widgetToolkit;

  protected Component(WidgetToolkit wd) {
    this.widgetToolkit = wd;
  }

  public abstract void checkEvent();
}
```

```
class Container extends Component {
  public Container(WidgetToolkit wd) {
    super(wd);
  }

  public void checkEvent() {
    widgetToolkit.checkInputEvent();
  }
}
```

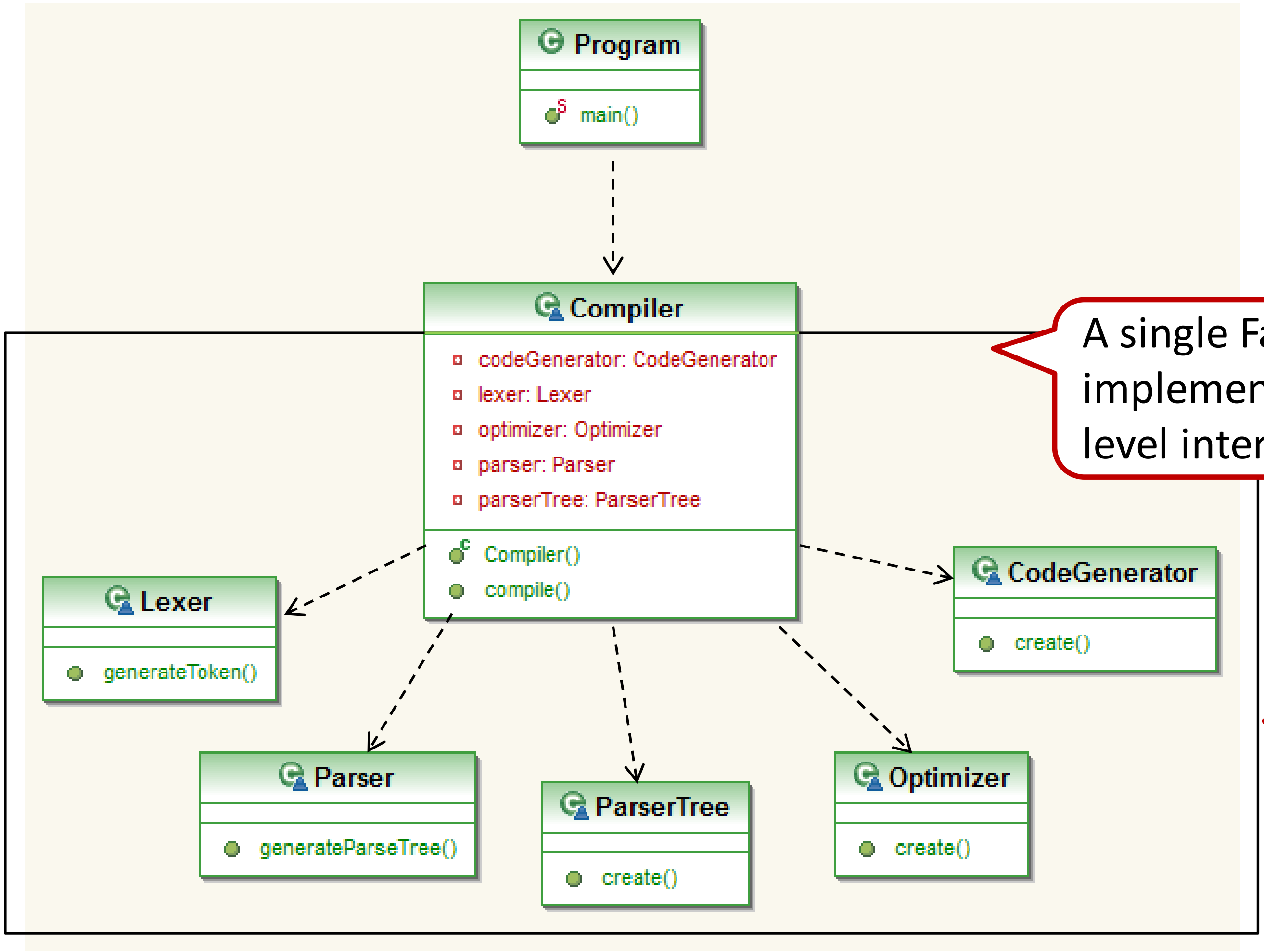
```
public class Program {
  public static void main(String[] args) {
    Component radioButton = new Container(new RadioButtonImpl());
    Component checkBox    = new Container(new CheckBoxImpl());

    radioButton.checkEvent();
    checkBox.checkEvent();
  }
}
```

Abstraction and Implementor can be refined independently.

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface
- 

An Example Application Using Façade Pattern



Program

main()

Compiler

codeGenerator: CodeGenerator

lexer: Lexer

optimizer: Optimizer

parser: Parser

parserTree: ParserTree

Compiler()

compile()

Lexer

generateToken()

CodeGenerator

create()

Parser

generateParseTree()

ParserTree

create()

Optimizer

create()

A single Façade class implements a high-level interface

Subsystems

An Example Application Using Façade Pattern

```
/* Complex parts */
class Lexer {
  public void generateToken() { /*..*/ }
}

class Parser {
  public void generateParseTree() { /*..*/ }
}

class ParserTree {
  public void create() { /*..*/ }
}
```

```
class CodeGenerator {
  public void create() { /*..*/ }
}

class Optimizer {
  public void create() { /*..*/ }
}
```

An Example Application Using Façade Pattern

```java
/* Subsystem Encapsulation */
class Compiler {
  private Lexer lexer;
  private Parser parser;
  private ParserTree parserTree;
  private CodeGenerator codeGenerator;
  private Optimizer optimizer;

  public Compiler() {
    this.lexer = new Lexer();
    this.parser = new Parser();
    this.parserTree = new ParserTree();
    this.codeGenerator = new CodeGenerator();
    this.optimizer = new Optimizer();
  }

  public void compile() {
    lexer.getToken();
    parser.generateParseTree();
    parserTree.create();
    codeGenerator.create();
    optimizer.create();
  }
}
```

```java
public class ProgramFacadePattern {
  public static void main(String[] args) {
    Compiler compiler = new Compiler();
    compiler.compile();
  }
}
```

A caller does not access the lower-level classes directly.

A single Façade class implements a high-level interface by invoking the methods of lower-level classes.