DESIGN PATTERNS

Course 4

COURSE 3 - CONTENT

- Creational patterns
  - Singleton patterns
  - Builder pattern
  - Prototype pattern
  - Factory method pattern
  - Abstract factory pattern

CONTENT

- Structural patterns
  - Adapter
  - Bridge
  - Façade
  - Flyweight
- Behavioral patterns
  - Iterator

ADAPTER

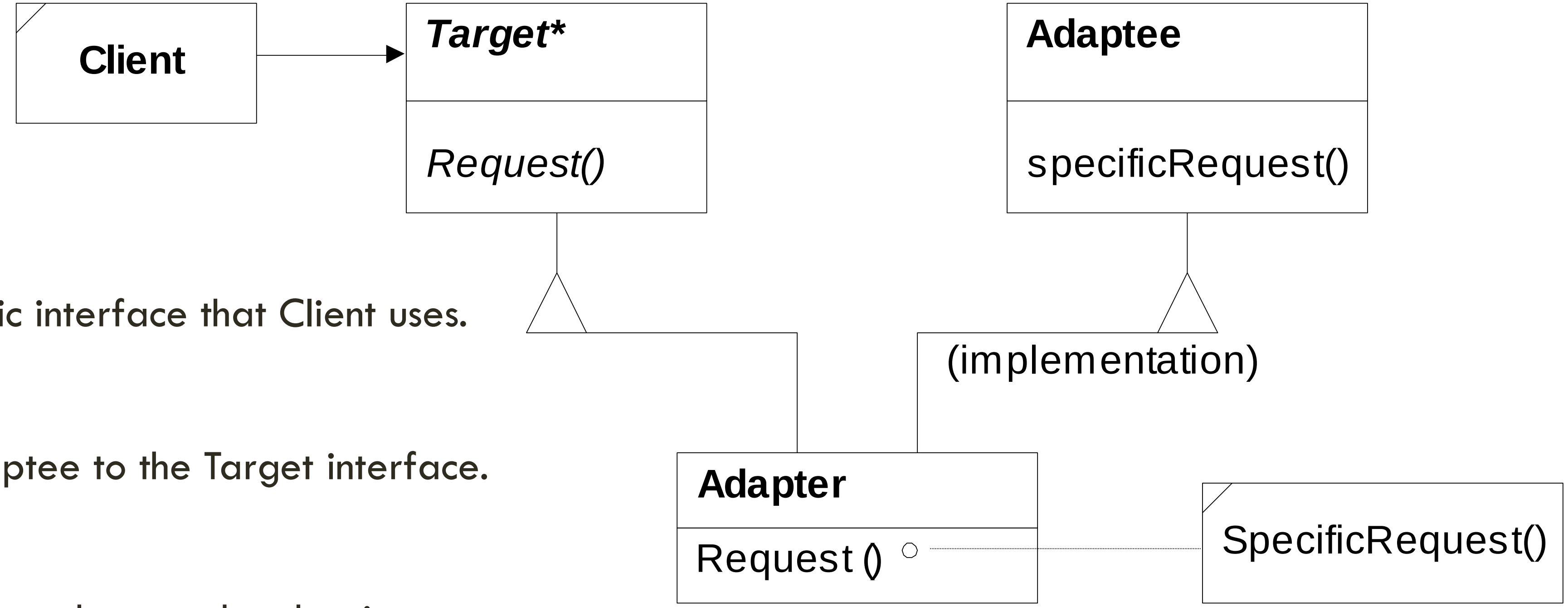
Indent

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Also Known As – Wrapper

Problem

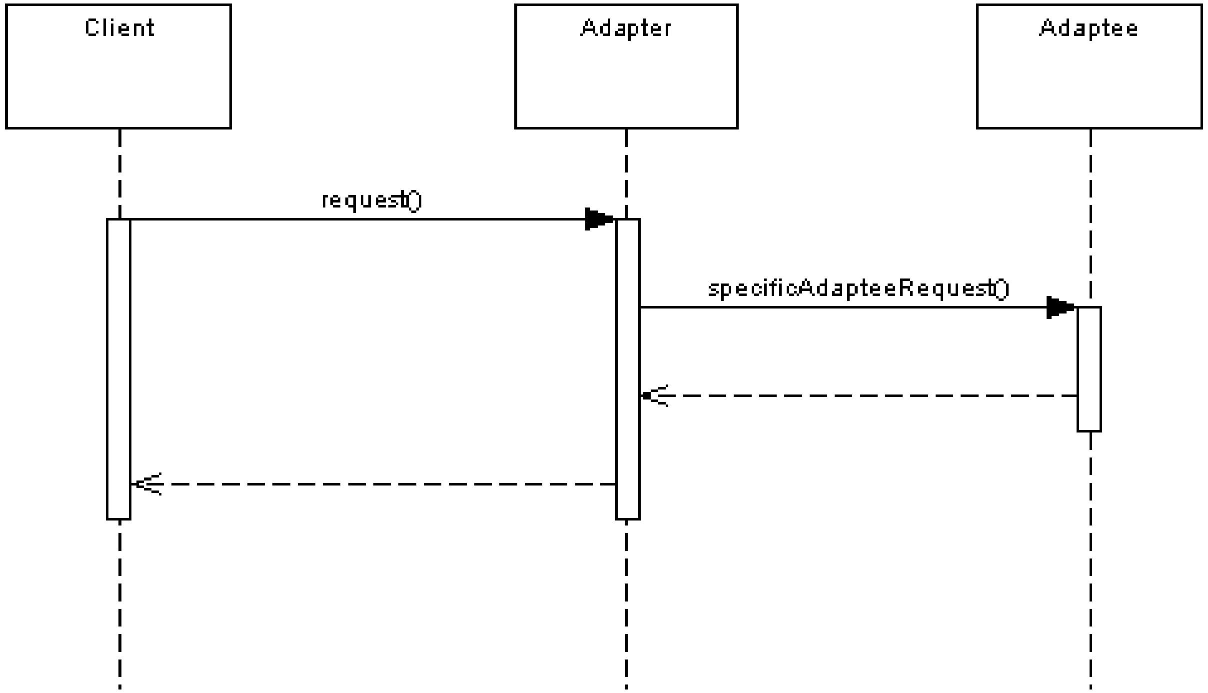
- Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
- We can not change the library interface, since we may not have its source code
- Even if we did have the source code, we probably should not change the library for each domain-specific application

ADAPTER



- Target
  - defines the domain-specific interface that Client uses.
- Adapter
  - adapts the interface Adaptee to the Target interface.
- Adaptee
  - defines an existing interface that needs adapting.
- Client
  - collaborates with objects conforming to the Target interface.

ADAPTER



ADAPTER

Applicability

- Use the Adapter pattern when
  - 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 classes with incompatible interfaces

Consequences

- Class adapter
  - Concrete Adapter class
  - Unknown Adaptee subclasses might cause problem
  - Overloads Adaptee behavior
  - Introduces only one object
- Object adapter
  - Adapter can service many different Adaptees
  - May require the creation of Adaptee subclasses and referencing those objects

ADAPTER

- How much adapting should be done?
  - Simple interface conversion that just changes operation names and order of arguments
  - Totally different set of operations

- When to use adapter?
  - 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.
  - you have several subclasses and would like to adapt some of their operations. Use Object Adapter to adapt their parent class instead of adapting all subclasses

ADAPTER. EXAMPLE

```java
public class CelciusReporter {
	double temperatureInC;
	public CelciusReporter() {
	}
	public double getTemperature() {
		return temperatureInC;
	}
	public void setTemperature(double temperatureInC) {
		this.temperatureInC = temperatureInC;
	}
}
```

```java
public interface TemperatureInfo {

	public double getTemperatureInF();

	public void setTemperatureInF(double temperatureInF);

	public double getTemperatureInC();

	public void setTemperatureInC(double temperatureInC);

}
```

ADAPTER. EXAMPLE

TemperatureClassReporter is a class adapter. It extends CelciusReporter (the adaptee) and implements TemperatureInfo (the target interface). If a temperature is in Celcius, TemperatureClassReporter utilizes the temperatureInC value from CelciusReporter. Fahrenheit requests are internally handled in Celcius.

```
// example of a class adapter
public class TemperatureClassReporter extends CelciusReporter
implements TemperatureInfo {
        @Override
        public double getTemperatureInC() {
                return temperatureInC;
        }
        @Override
        public double getTemperatureInF() {
                return cToF(temperatureInC);
        }
        @Override
        public void setTemperatureInF(double temperatureInF) {
                this.temperatureInC = fToC(temperatureInF);
        }
        private double fToC(double f) {
                return ((f - 32) * 5 / 9);
        }
        private double cToF(double c) {
                return ((c * 9 / 5) + 32);
        }
}
```

ADAPTER. EXAMPLE

TemperatureObjectReporter is an object adapter. It is similar in functionality to TemperatureClassReporter, except that it utilizes composition for the CelciusReporter rather than inheritance.

```
// example of an object adapter
public class TemperatureObjectReporter implements TemperatureInfo {
		CelciusReporter celciusReporter;
		public TemperatureObjectReporter() {
				celciusReporter = new CelciusReporter();
		}
		@Override
		public double getTemperatureInC() {
				return celciusReporter.getTemperature();
		}
		@Override
		public double getTemperatureInF() {
				return cToF(celciusReporter.getTemperature());
		}
		@Override
		public void setTemperatureInC(double temperatureInC) {
		celciusReporter.setTemperature(temperatureInC);
		}
		@Override
		public void setTemperatureInF(double temperatureInF) {
		celciusReporter.setTemperature(fToC(temperatureInF));
		}
		private double fToC(double f) {
				return ((f - 32) * 5 / 9);
		}
		private double cToF(double c) {
				return ((c * 9 / 5) + 32);
		}
}
```

ADAPTER. EXAMPLE

```
public class AdapterDemo {
	public static void main(String[] args) {
		// class adapter
		System.out.println("class adapter test");
		TemperatureInfo tempInfo = new
TemperatureClassReporter();
		testTempInfo(tempInfo);
		// object adapter
		System.out.println("\nobject adapter test");
		tempInfo = new TemperatureObjectReporter();
		testTempInfo(tempInfo);

	}
}
```

```
public static void testTempInfo(TemperatureInfo tempInfo) {
		tempInfo.setTemperatureInC(0);
		System.out.println("temp in C:" + tempInfo.getTemperatureInC());
		System.out.println("temp in F:" + tempInfo.getTemperatureInF());

		tempInfo.setTemperatureInF(85);
		System.out.println("temp in C:" + tempInfo.getTemperatureInC());
		System.out.println("temp in F:" + tempInfo.getTemperatureInF());
}
```

ITERATOR

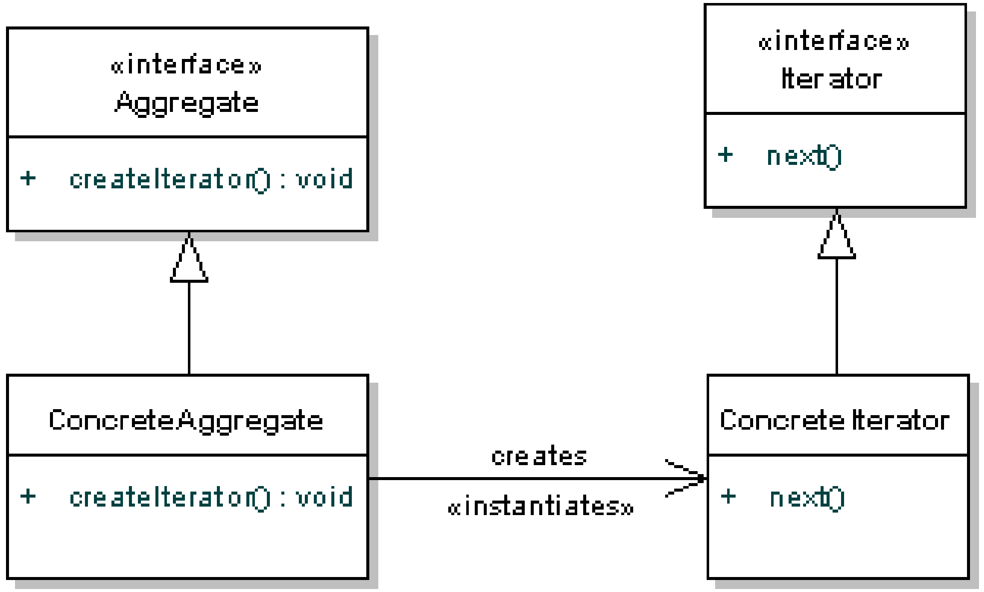
Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.

Problem

- An object that provides a standard way to examine all elements of any collection
- Uniform interface for traversing many different data structures without exposing their implementations
- Supports concurrent iteration and element removal
- Removes need to know about internal structure of collection or different methods to access data from different collections

ITERATOR



- Aggregate
  - defines an interface for the creation of the Iterator object.
- ConcreteAggregate
  - implements this interface, and returns an instance of the ConcreteIterator.
- Iterator
  - defines the interface for access and traversal of the elements
- ConcreteIterator
  - implements this interface while keeping track of the current position in the traversal of the Aggregate.

ITERATOR. EXAMPLE IN JAVA

```
public interface java.util.Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
}

public interface java.util.Collection {
  ...   // List, Set extend Collection
  public Iterator iterator();
}

public interface java.util.Map {
  ...
  public Set keySet();          // keys,values are Collections
  public Collection values();  // (can call iterator() on them)
}
```

ITERATOR. EXAMPLE IN JAVA

- all Java collections have a method iterator that returns an iterator for the elements of the collection
- can be used to look through the elements of any kind of collection (an alternative to for loop)

```
List list = new ArrayList();
... add some elements ...

for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

```
set.iterator()
map.keySet().iterator()
map.values().iterator()
```

BRIDGE

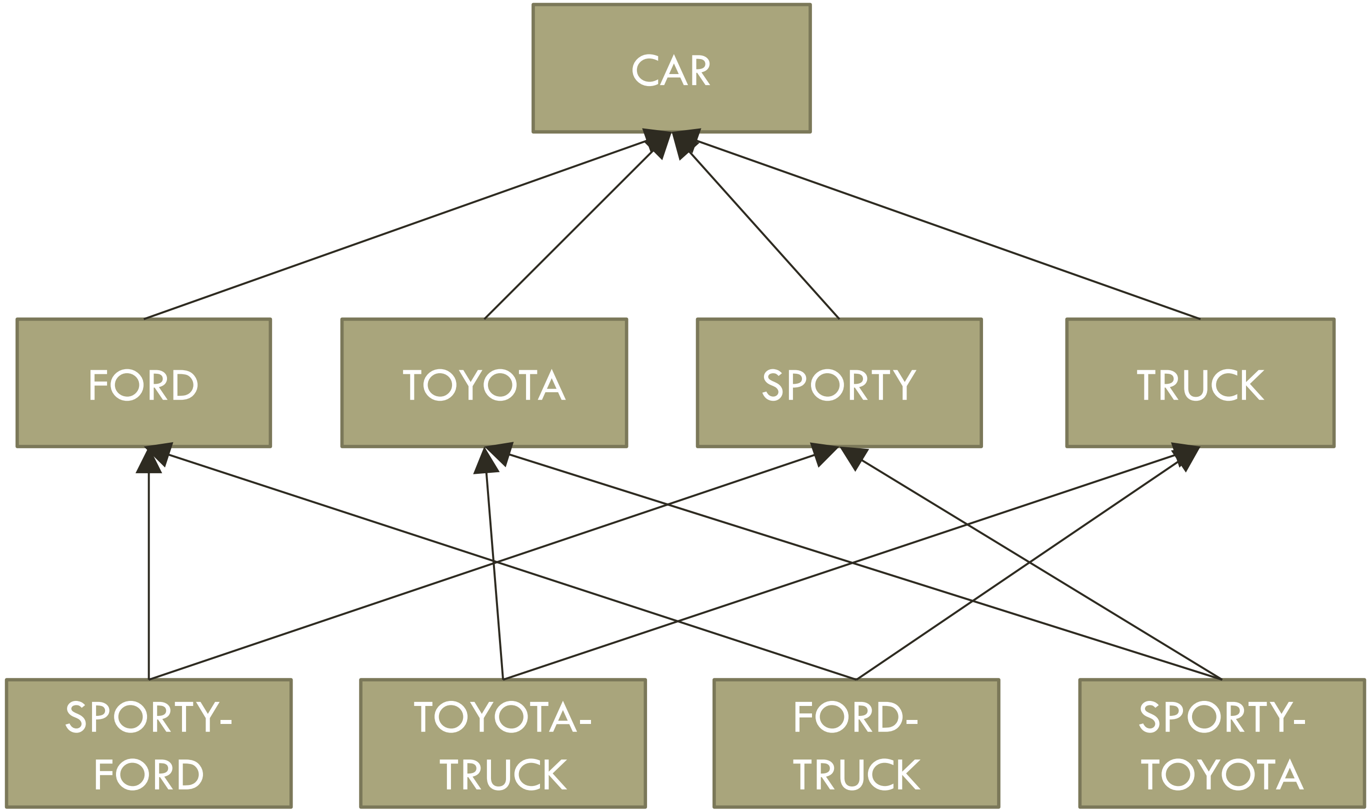
Intent

- Separate a (logical) abstraction interface from its (physical) implementation(s)
- Allows different implementations of an interface to be decided upon dynamically.

Applicability

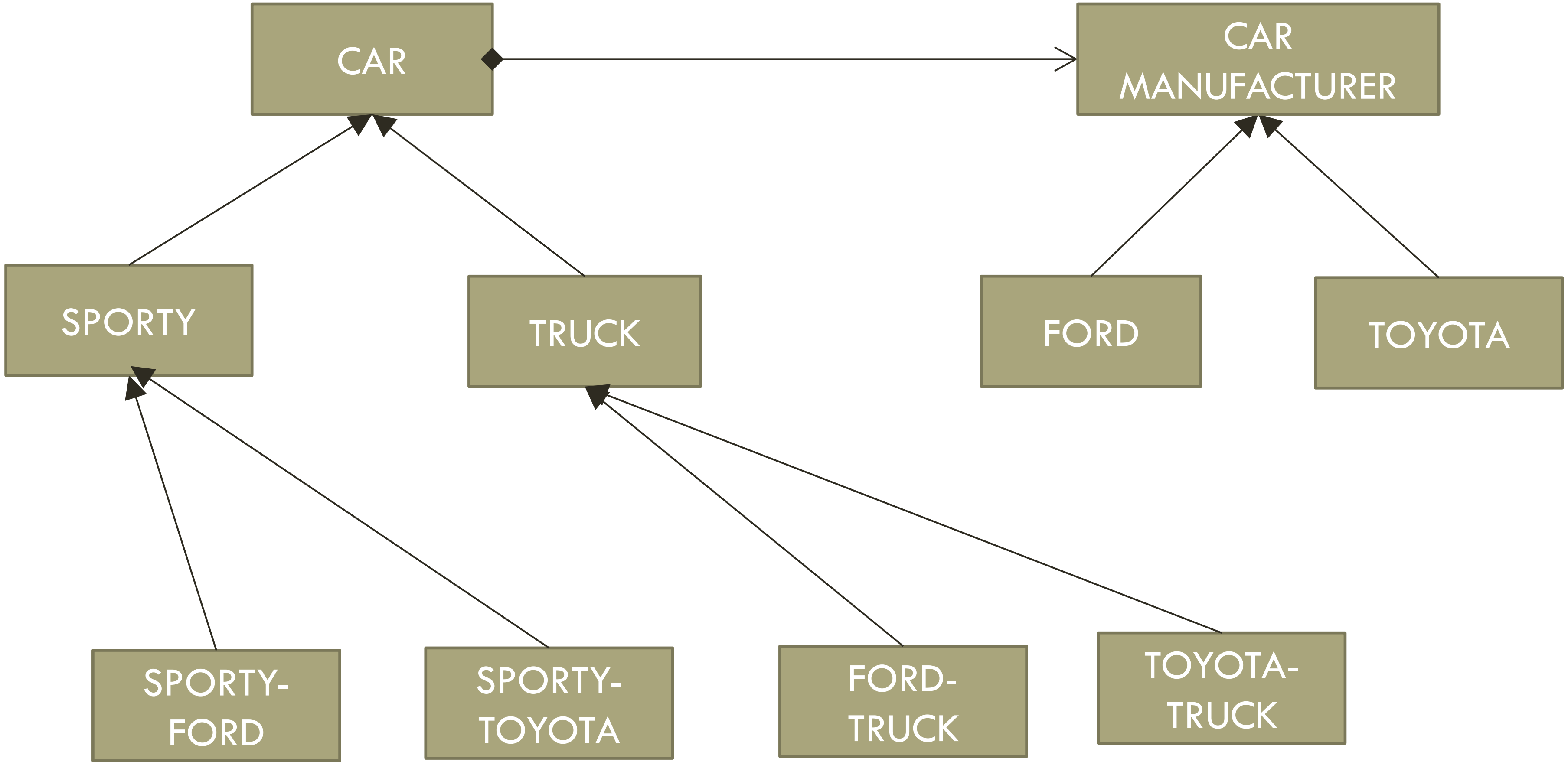
- When interface & implementation should vary independently
- Require a uniform interface to interchangeable class hierarchies

BRIDGE



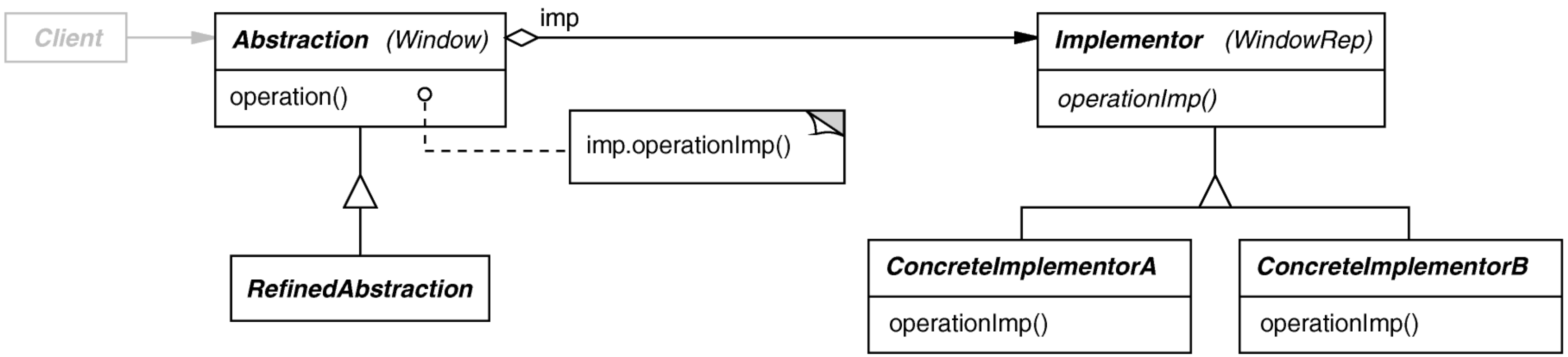
Can this hierarchy be simplified and easy to understand? How?

BRIGE



CAR
CAR MANUFACTURER
SPORTY
TRUCK
FORD
TOYOTA
SPORTY-FORD
SPORTY-TOYOTA
FORD-TRUCK
TOYOTA-TRUCK

BRIDGE



- Abstraction
  - defines the abstraction's interface
  - maintains a reference to the Implementor
- RefinedAbstraction
  - extends abstraction interface
- Implementor
  - defines interface for implementations
- ConcreteImplementor
  - implements Implementor interface, ie defines an implementation

BRIDGE. EXAMPLE

1. Graphical User Interface Frameworks.
   - Use the bridge pattern to separate abstractions from platform specific implementation.
   - GUI frameworks separate a Window abstraction from a Window implementation for Linux or Mac OS using the bridge pattern.
2. Object Persistence API.
   - Many implementations depending on the presence or absence of a relational database, a file system, as well as on the underlying operating system

BRIDGE. IMPLEMENTATION OF CHAR EXAMPLE

```
public abstract class Car {
   private CarManufator manufactor;
   public Car ( CarManufator manufactor) {
        this.manufactor = manufactor
  }
}

public interface CarManufactor{
  public void getManufactor();
}
```

```
public class Ford implements CarManufactor{
 public void getManufactor(){
    System.out.print("Ford producer");
 }
}

public class Toyota implements CarManufactor{
 public void getManufactor(){
    System.out.print("Toyota producer");
 }
}
```

BRIDGE. IMPLEMENTATION OF CHAR EXAMPLE

```
public class Sporty extends Car {
  public Sporty(CarManufator manufactor) {
     super(manufactor);
     System.out.println(manufactor.getManufactor() +" for Sporty car");
  }
}
public class Truck extends Car {
  public Truck(CarManufator manufactor) {
     super(manufactor);
     System.out.println(manufactor.getManufactor() + " for Truck car");
  }
}
```

```
public class Client {
  public static void main( String args[]){
   CarManufator mFord = new Ford();
   CarManufator mToyota = new Toyota();

   Car sportyFord = new Sporty(mFord);
   Car sportyToyota = new Sporty(mToyota);

   Car truckFord = new Truck(mFord);
   Car truckToyota = new Truck(mToyota);
    }
}
```

BRIDGE

- Decouples interface and implementation

  Decoupling Abstraction and Implementor also eliminates compile-time dependencies on implementation. Changing implementation class does not require recompile of abstraction classes.

- Improves extensibility

  Both abstraction and implementations can be extended independently

- Hides implementation details from clients
- More of a design-time pattern

BRIDGE

Disadvantages

- abstractions that have only one implementation
- creating the right Implementor
- sharing implementors
- use of multiple inheritance

Implementation Isues

How, where, and when to decide which implementer to instantiate?

- Depends:

  - If Abstraction knows about all concrete implementer, then it can instantiate in the constructor.

  - It can start with a default and change it later

  - Or it can delegate the decision to another object (to an abstract factory for example)

- Can't implement a true bridge using multiple inheritance

A class can inherit publicly from an abstraction and privately from an implementation, but since it is static inheritance it bind an implementation permanently to its interface

FAÇADE

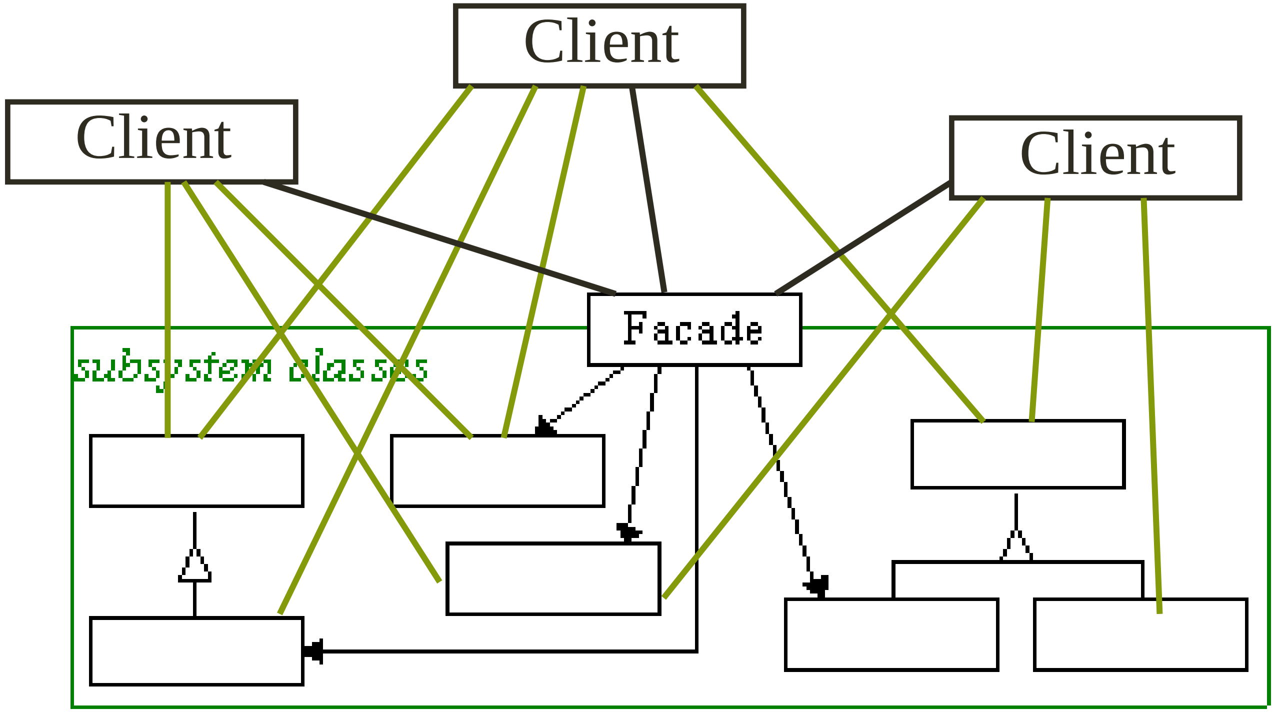
Intent

- To provide a unified interface to a set of interfaces in a subsystem
- To simplify an existing interface
- Defines a higher-level interface that makes the subsystem easier to use

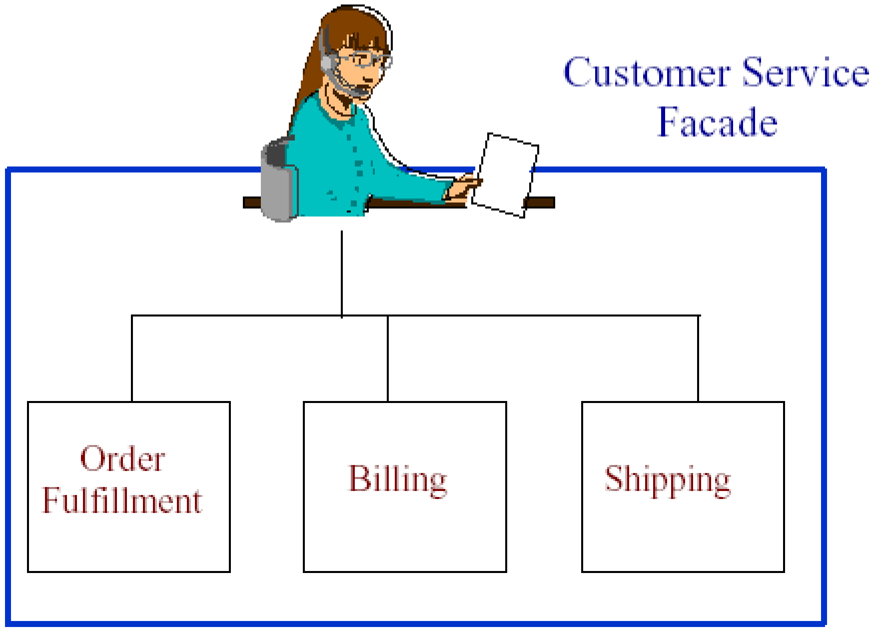
Problem

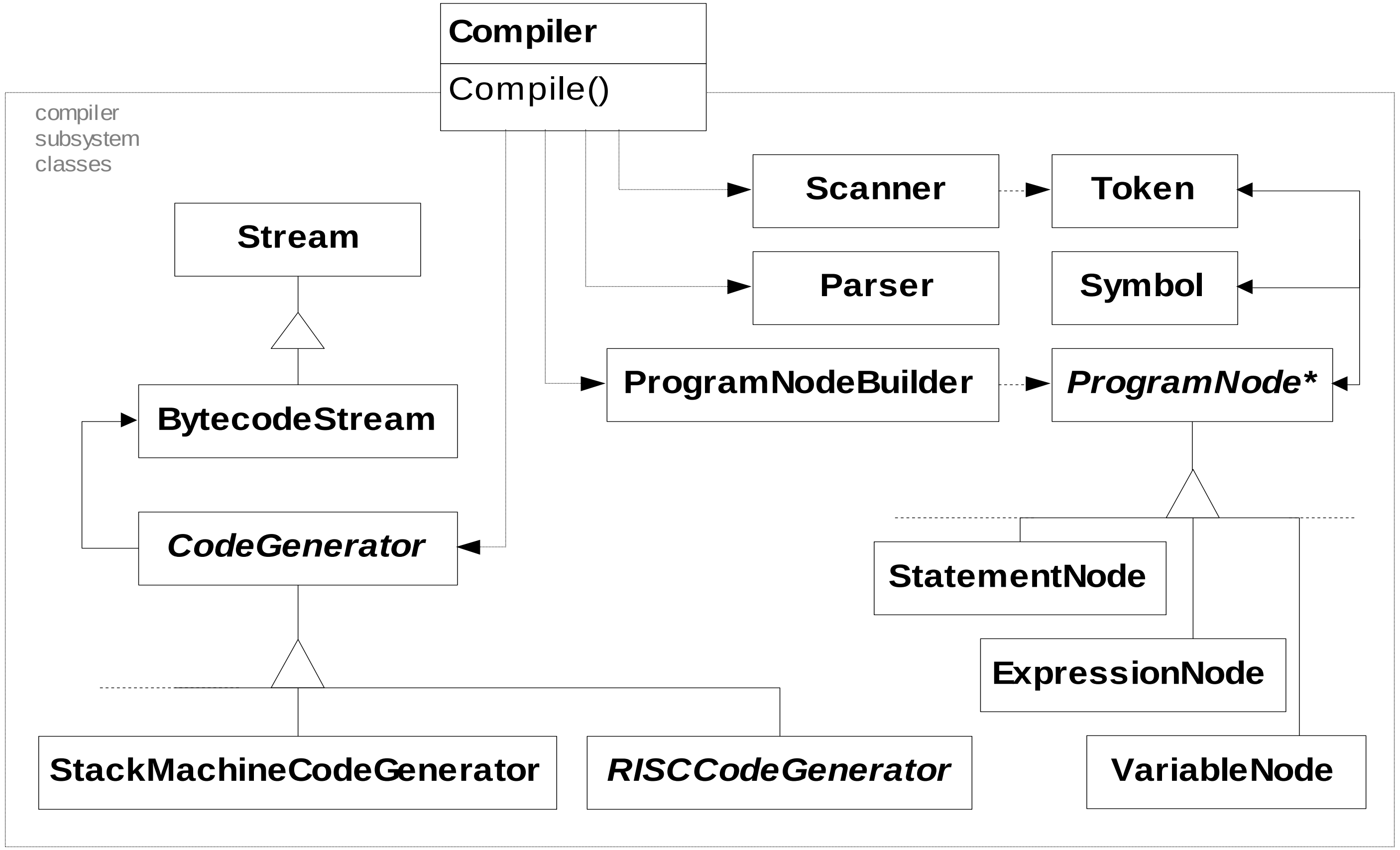
- Situation I: Wish to simplify a process for most clients
  - Subsystems are built for multiple applications
  - Most applications use only a small subset
  - Most applications interact in a predefined manner
- Situation II: Wish to reduce the number of dependencies between client and implementation classes
  - Requires an interface that allows a level of isolation
- Situation III: Wish to build a layered software design with all inter-layer communication between interfaces

FAÇADE. PATTERN DESCRIPTION

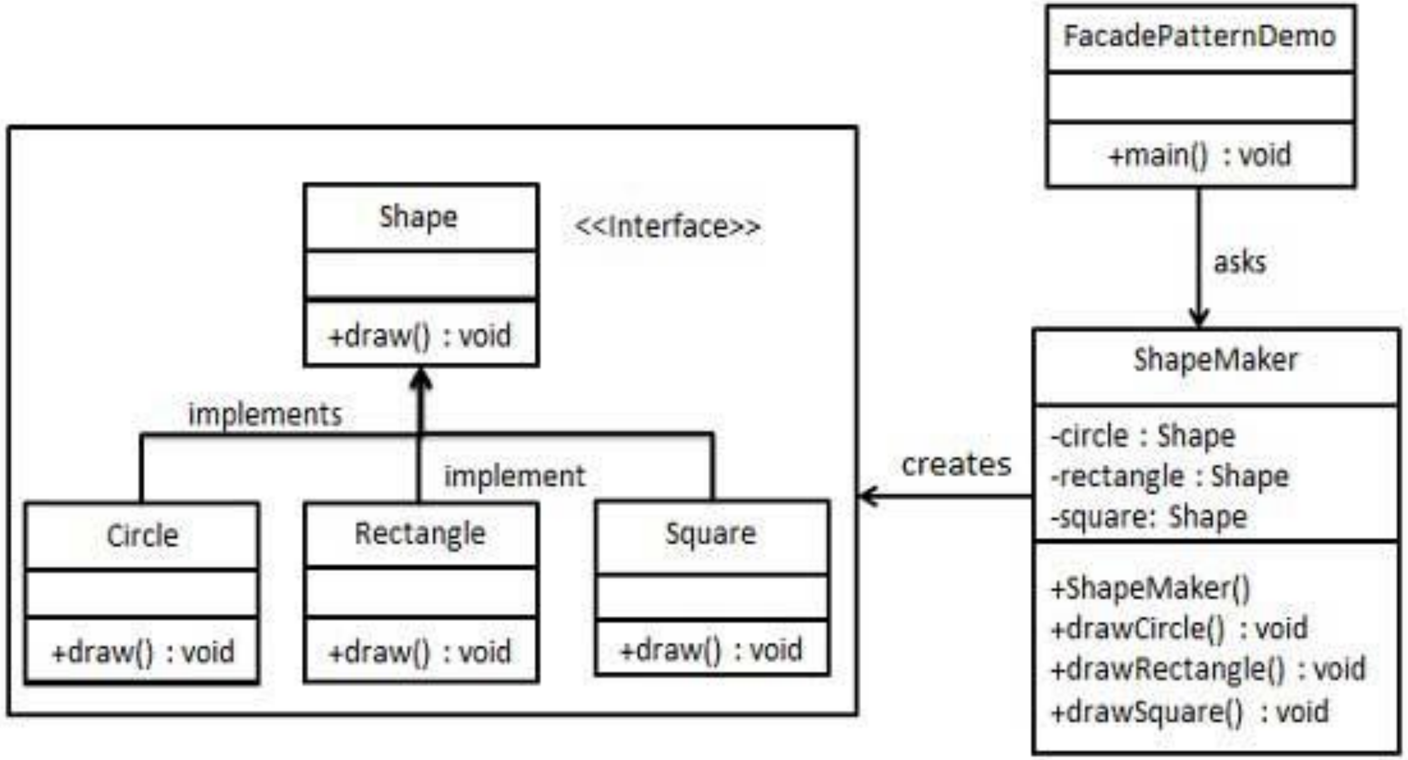


FAÇADE. EXAMPLE





FAÇADE. EXAMPLE



FAÇADE. EXAMPLE. IMPLEMENTATION

```
public interface Shape {
        void draw();
}

public class Rectangle implements Shape {
   @Override
    public void draw() {
        System.out.println("Rectangle::draw()");
    }
 }
```

```
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square::draw()");
     }
}
public class Circle implements Shape {
     @Override
      public void draw() {
          System.out.println("Circle::draw()");
      }
 }
```

FAÇADE. EXAMPLE. IMPLEMENTATION

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;
   public ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
  }
public void drawCircle(){
       circle.draw();
}
public void drawRectangle(){
            rectangle.draw();
 }
    public void drawSquare(){
        square.draw();
     }
}
public class FacadePatternDemo {
   public static void main(String[] args) {
    ShapeMaker shapeMaker = new ShapeMaker();
     shapeMaker.drawCircle();
     shapeMaker.drawRectangle();
     shapeMaker.drawSquare();
    }
}
```

FAÇADE

Consequences

- Shields clients from subsystem complexity

- Promotes weak coupling between clients and subsystems
  - Easier to swap out alternatives

- Allows more advanced clients to by-pass and have direct subsystem access

FAÇADE

Implementation Issues

- Can involve nontrivial manipulation of subsystem
  - May require several steps in sequence or composition
  - May require temporary storage
- Can completely hide subsystem
  - Place subsystem and façade in package
  - Make façade only public class
  - Can be difficult if subsystem objects returned to client
- Can implement Façade as abstract class
  - Allows different concrete facades under same interface

FLYWEIGHT

Intent

- "Use Sharing to support large numbers of fine-grained objects efficiently."
- Simply put, a method for storing a small number of complex objects that are used repeatedly.
- Flyweight factors the common properties of multiple instances of a class into a single object, saving space and maintenance of duplicate instances.

Problem

- Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

FLYWEIGHT. EXAMPLE

Flyweighted strings

- Java Strings are flyweighted by the compiler wherever possible

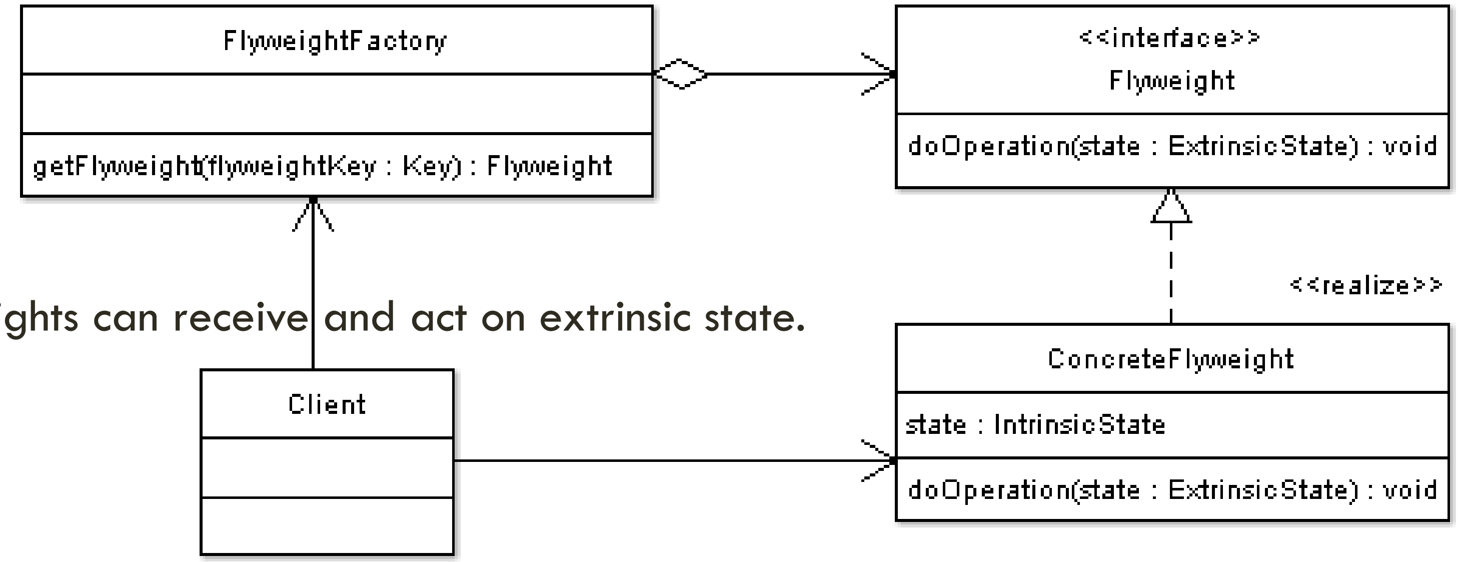
Flyweighting works best on *immutable* objects

```
public class StringTest {
  public static void main(String[] args) {
    String fly  = "fly", weight  = "weight";
    String fly2 = "fly", weight2 = "weight";

    System.out.println(fly == fly2);                      // true
    System.out.println(weight == weight2);                // true

    String distinctString = fly + weight;
    System.out.println(distinctString == "flyweight"); // false

    String flyweight = (fly + weight).intern();
    System.out.println(flyweight == "flyweight");      // true
  }
}
```

FLYWEIGHT. APPLICABILITY

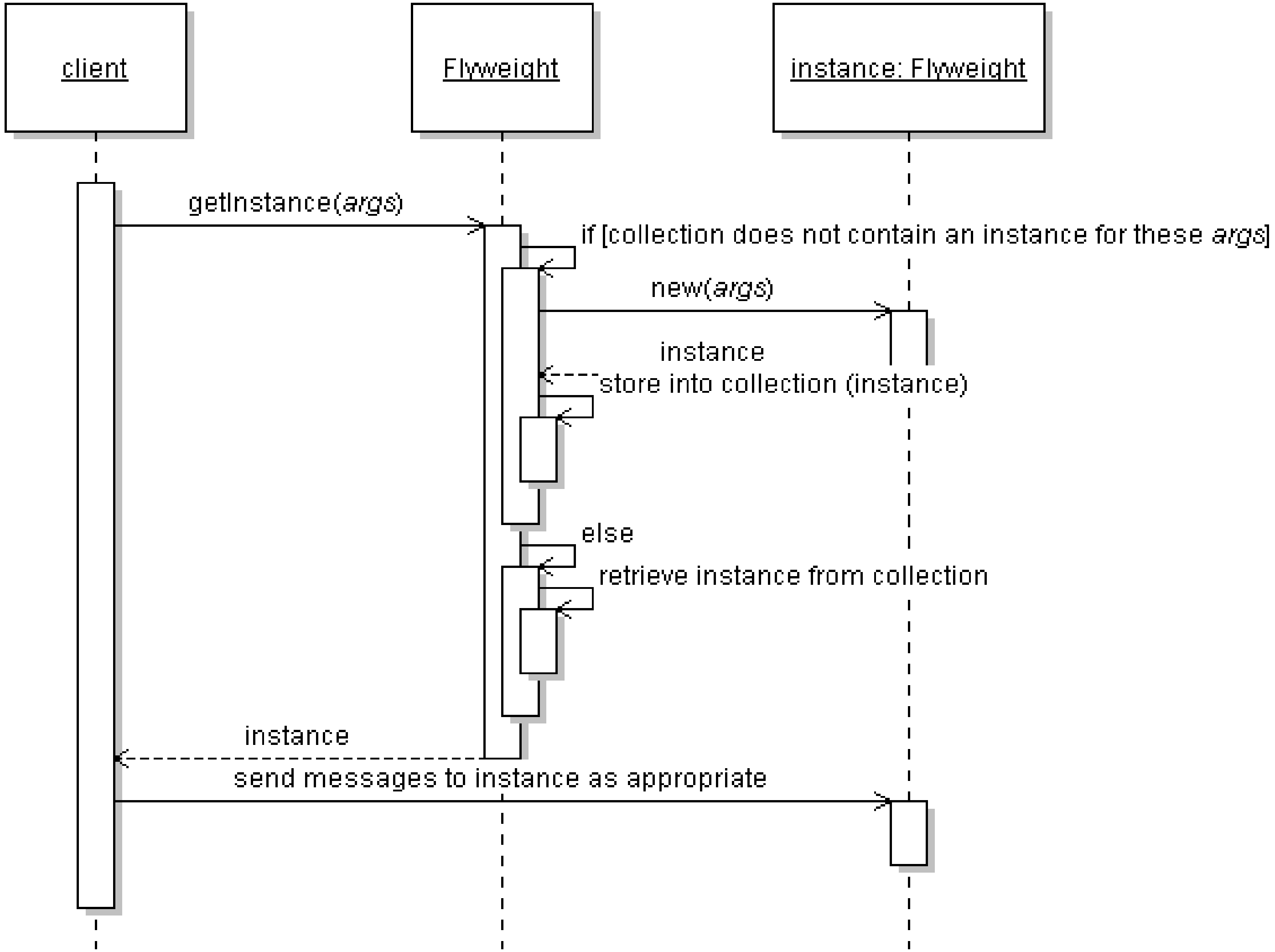
- Application has a large number of objects.
- Storage costs are high because of the large quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few once you remove their extrinsic state.
- The application doesn’t depend on object identity.

FLYWEIGHT. DESIGN



- Flyweight
  - Declares an interface through which flyweights can receive and act on extrinsic state.
- ConcreteFlyweight
  - Stores intrinsic state of the object.
  - Must be sharable.
  - Must maintain state that it is intrinsic to it, and must be able to manipulate state that is extrinsic.
- FlyweightFactory
  - The factory that creates and manages flyweight objects.
  - The factory ensures sharing of the flyweight objects.
  - The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.
- Client
  - A client maintains references to flyweights in addition to computing and maintaining extrinsic state

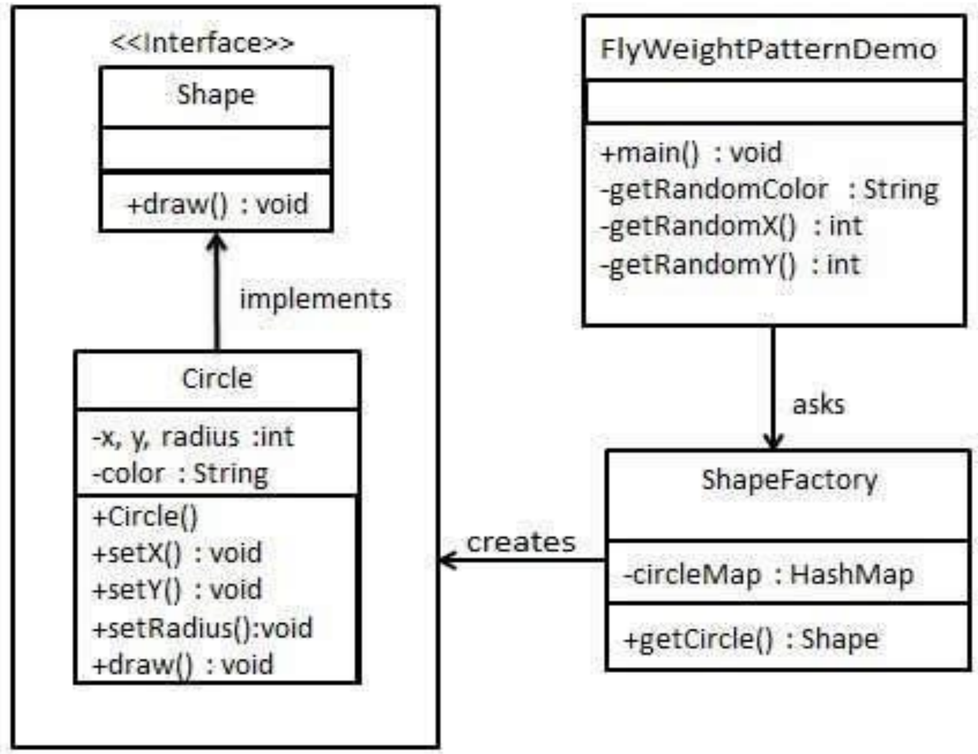
FLYWEIGHT.DESIGN



FLYWEIGHT. EXAMPLE

Drawing 20 circles of different locations but we will create only 5 objects.

- Only 5 colors are available so color property is used to check already existing Circle objects



FLYWEIGHT. EXAMPLE

```
public interface Shape {
  void draw();
}
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;
   public Circle(String color){
      this.color = color;
   }
```

```
public void setX(int x) {
    this.x = x;
 }
 public void setY(int y) {
     this.y = y;
}
public void setRadius(int radius) {
     this.radius = radius;
}
@Override
 public void draw() {
          System.out.println("Circle: Draw() [Color : " + color + ", x : " + x
+ ", y :" + y + ", radius :" + radius);
}
}
```

FLYWEIGHT. EXAMPLE

```
public class ShapeFactory {

   private static final HashMap<String, Shape> circleMap = new HashMap();


   public static Shape getCircle(String color) {

      Circle circle = (Circle)circleMap.get(color);


      if(circle == null) {

         circle = new Circle(color);

         circleMap.put(color, circle);

         System.out.println("Creating circle of color : " + color);

      }

      return circle;

   }

}
```

```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

FLYWEIGHT

Benefits

- If the size of the set of objects used repeatedly is substantially smaller than the number of times the object is logically used, there may be an opportunity for a considerable cost benefit
- When To Use Flyweight:
  - There is a need for many objects to exist that share some intrinsic, unchanging information
  - Objects can be used in multiple contexts simultaneously
  - Acceptable that flyweight acts as an independent object in each instance

Consequences

- Overhead to track state
  - Transfer
  - Search
  - Computation
- When Not To Use Flyweight:
  - If the extrinsic properties have a large amount of state information that would need passed to the flyweight (overhead)
  - Need to be able to be distinguished shared from non-shared objects