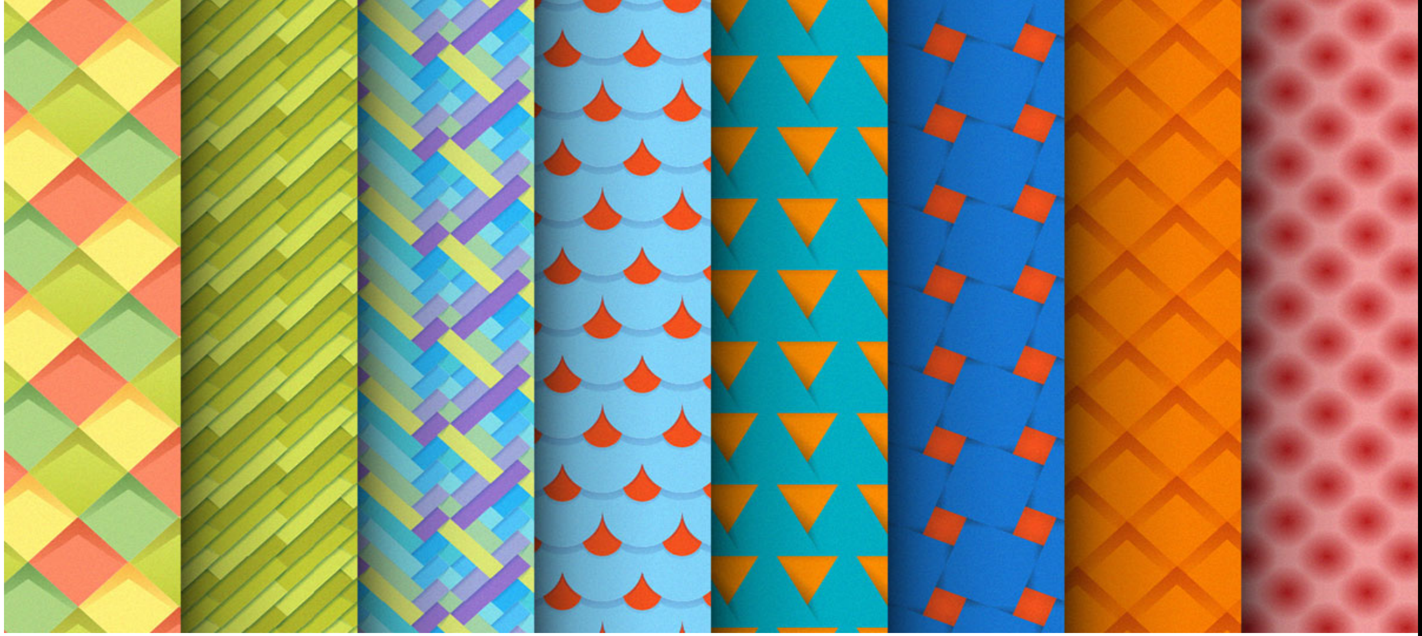


DESIGN PATTERNS

COURSE 4

PREVIOUS COURSE

- **Creational Patterns**
  - Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate
  - Abstract Factory provides an interface for creating families of related objects, without specifying concrete classes
  - Builder separates the construction of a complex object from its representation, so that the same construction process can create different representation
  - Prototype specifies the kind of objects to create using a prototypical instances
  - Singleton ensures that a class has only one instance, and provides a global point of access to that instance

CONTENT

- ❑ **Structural patterns**
  - ❑ Adapter
  - ❑ Bridge
  - ❑ Façade
  - ❑ Flyweight
  - ❑ Proxy
  - ❑ Composite
  - ❑ Decorator

STRUCTURAL PATTERNS

- **Help identify and describe relationships between entities**

- **Address how classes and objects are composed to form large structures**
  - Class-oriented patterns use inheritance to compose interfaces or implementations
  - Object-oriented patterns describe ways to compose objects to realize new functionality, possibly by changing the composition at run-time

- **Example**
  - Proxy in distributed programming
  - Bridge in JDBC drivers

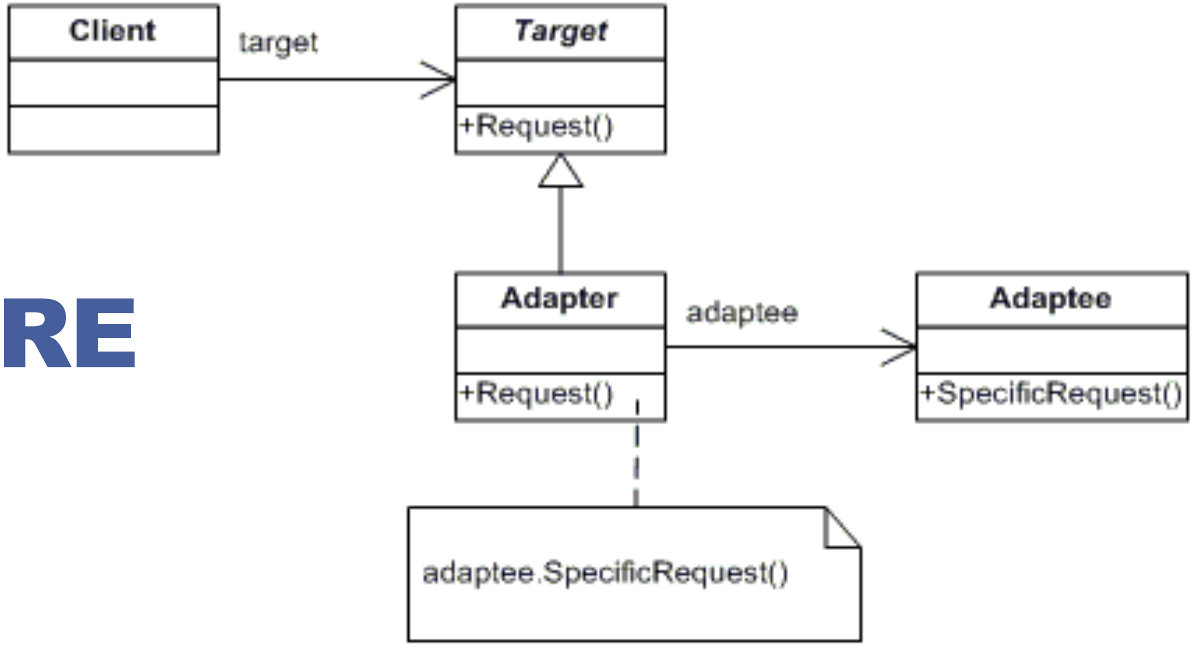
STRUCTURAL PATTERNS

- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

ADAPTER

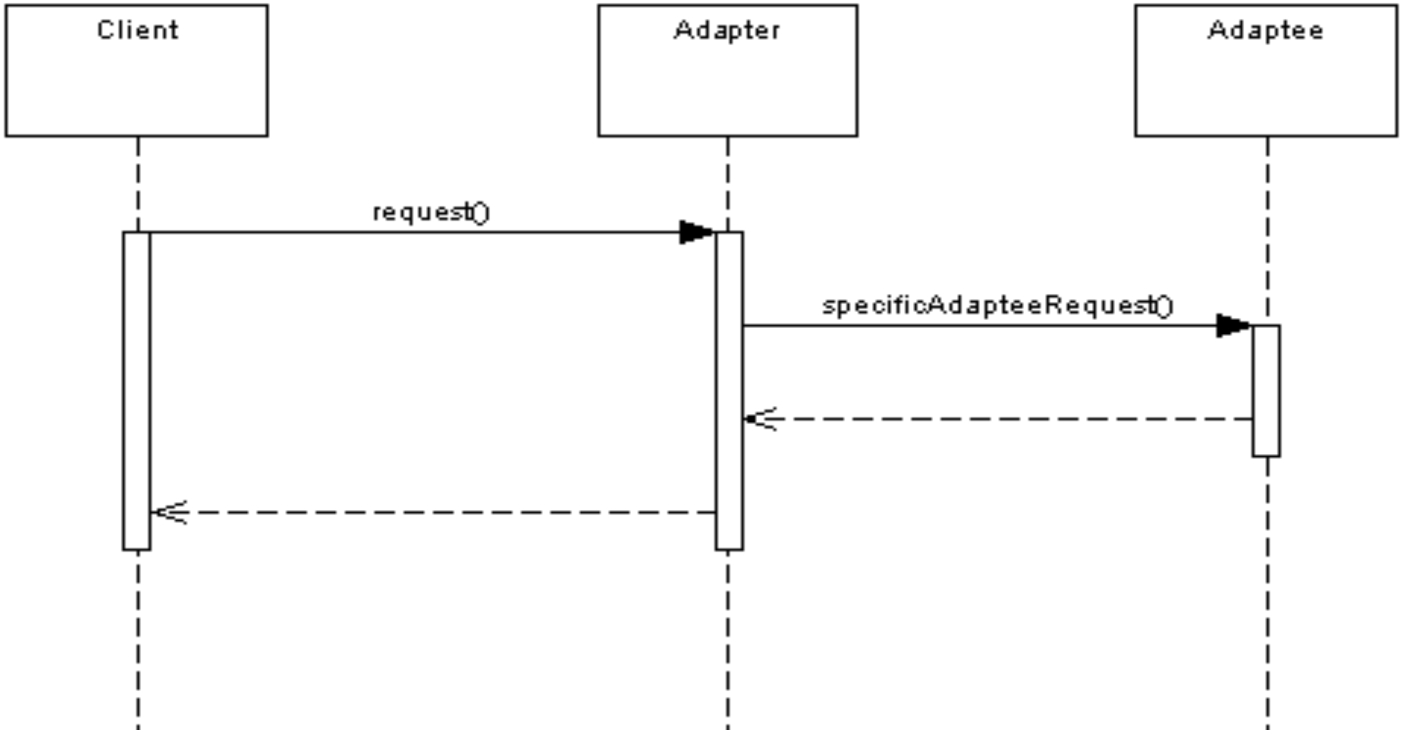
- **Indent**
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
- **Also Known As**
  - Wrapper
- **Problem**
  - Sometimes a toolkit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



- ❑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

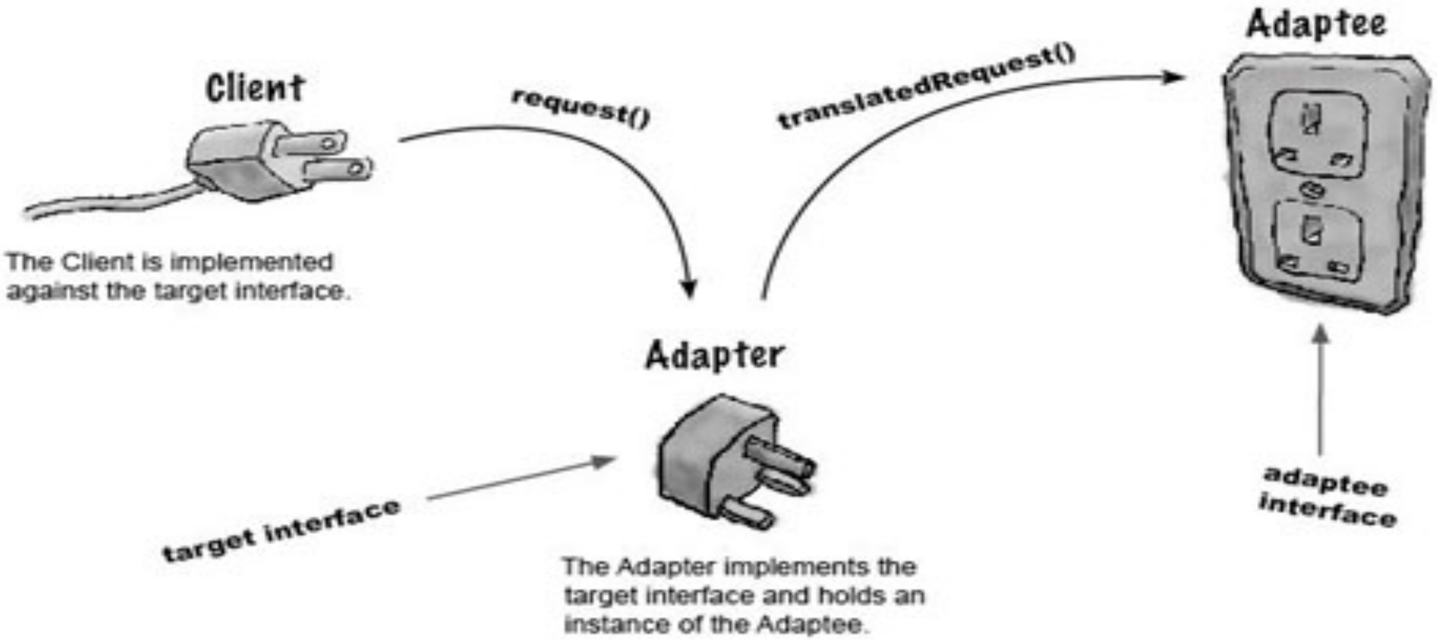
ADAPTER.



❑Client is concerned it's just calling the **request** method of the Target interface, which the Adapter has implemented.

❑In the background however, the Adapter knows that to return the right result, it needs to call a different method, **specificAdapteeRequest**, on the Adaptee.

ADAPTER. EXAMPLE


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.

- ❑ Eclipse plug-ins
  - ❑For a particular object to contribute to the Properties view, adapters are used display the objects data.
  - ❑The view itself doesn't need to know anything about the object the it is displaying properties for.

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

- **2 types of implementations**
  - Class adapter (suitable for programming languages that allow multiple inheritance)
    - 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 1

- **Consider that we have a third party library that provides print string functionality**
  - through PrintString class = adaptee

```
public class PrintString {
    public void print(String s)  {
            System.out.println(s);
    }
}
```

- **Client deals with `ArrayList<String>` but not with string.**
  - provided a PrintableList interface that expects the client input = target

```
public interface PrintableList {
     void printList(ArrayList<String> list);
}
```

  - Clients should see the printable list

ADAPTER EXAMPLE 1

- ❑ **Adapter pattern**

```
public class PrintableListAdapter
            implements PrintableList {
   public void printList(
             ArrayList<String> list) {
//Converting ArrayList<String> to
String so that
// we can pass String to adaptee class
  String listString = "";
  for (String s : list) {
      listString += s + "\t";
  }
  // instantiating adaptee class
  PrintString printString=new
                      PrintString();
  ps.print(listString);
 }
}
```

- ❑ **Client**

```
public class AdapterDPMain {

 public static void
        main(String[] args)
 {
  ArrayList<String> list=new
           ArrayList<String>();
  list.add("one");
  list.add("two");
  list.add("three");
  PrintableList pl=new
         PrintableListAdapter();
  pl.printList(list);
 }
}
```

ADAPTER EXAMPLE 2

- **We have the following 3th party library = adaptee**

```
public class CelciusReporter {

    double temperatureInC;

    public CelciusReporter() {
    }

    public double getTemperature() {
            return temperatureInC;
    }

    public void setTemperature(double temperatureInC) {
            this.temperatureInC = temperatureInC;
    }

}
```

ADAPTER EXAMPLE 2

- Target interface

```
public interface TemperatureInfo {

    public double getTemperatureInF();

    public void setTemperatureInF(double temperatureInF);

    public double getTemperatureInC();

    public void setTemperatureInC(double temperatureInC);

}
```

ADAPTER EXAMPLE 2

- **Propose a way to create an adapter using**
  - inheritance
  - composition

- **Hellper methos that allows transormation from celcius in farenheit**

```
private double fToC(double f) {
        return ((f - 32) * 5 / 9);
}

private double cToF(double c) {
        return ((c * 9 / 5) + 32);
}
```

STRUCTURAL PATTERNS

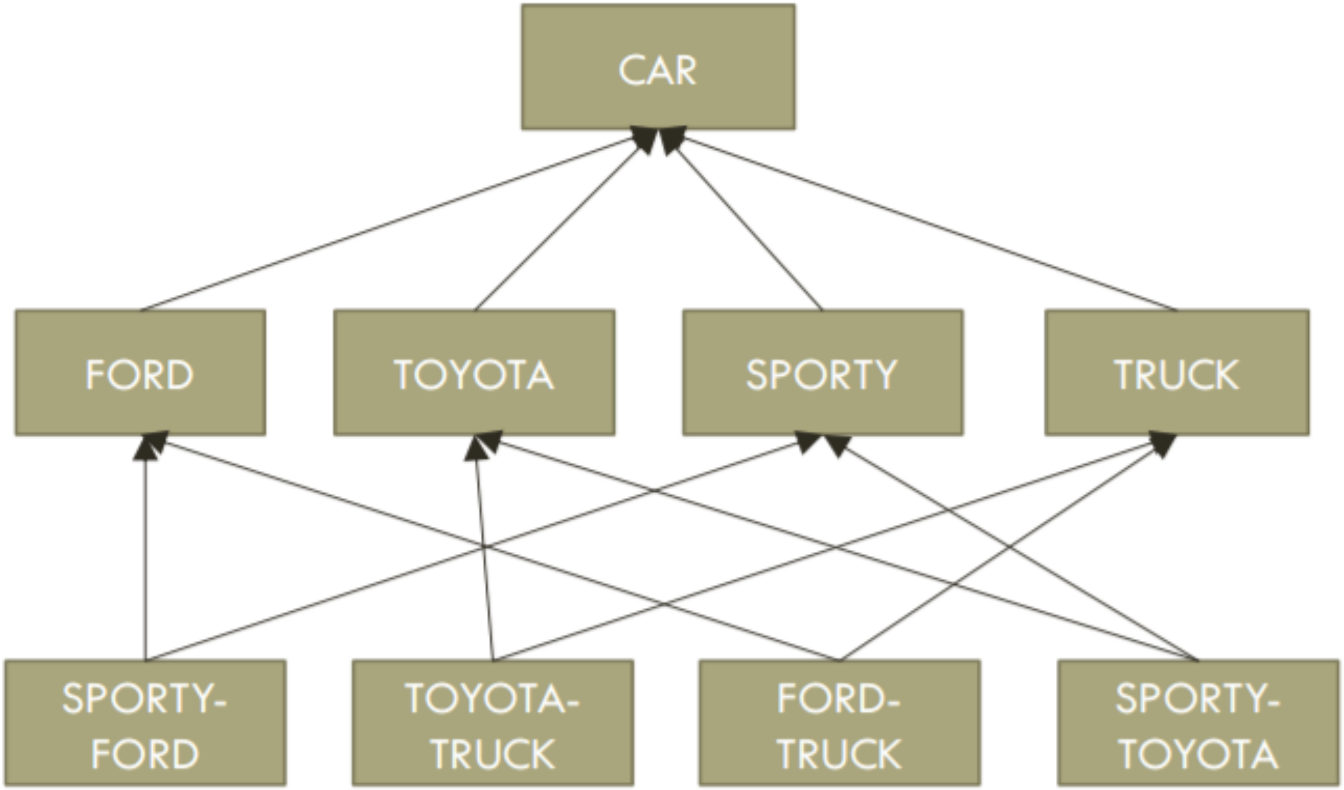
- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

BRIDGE

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

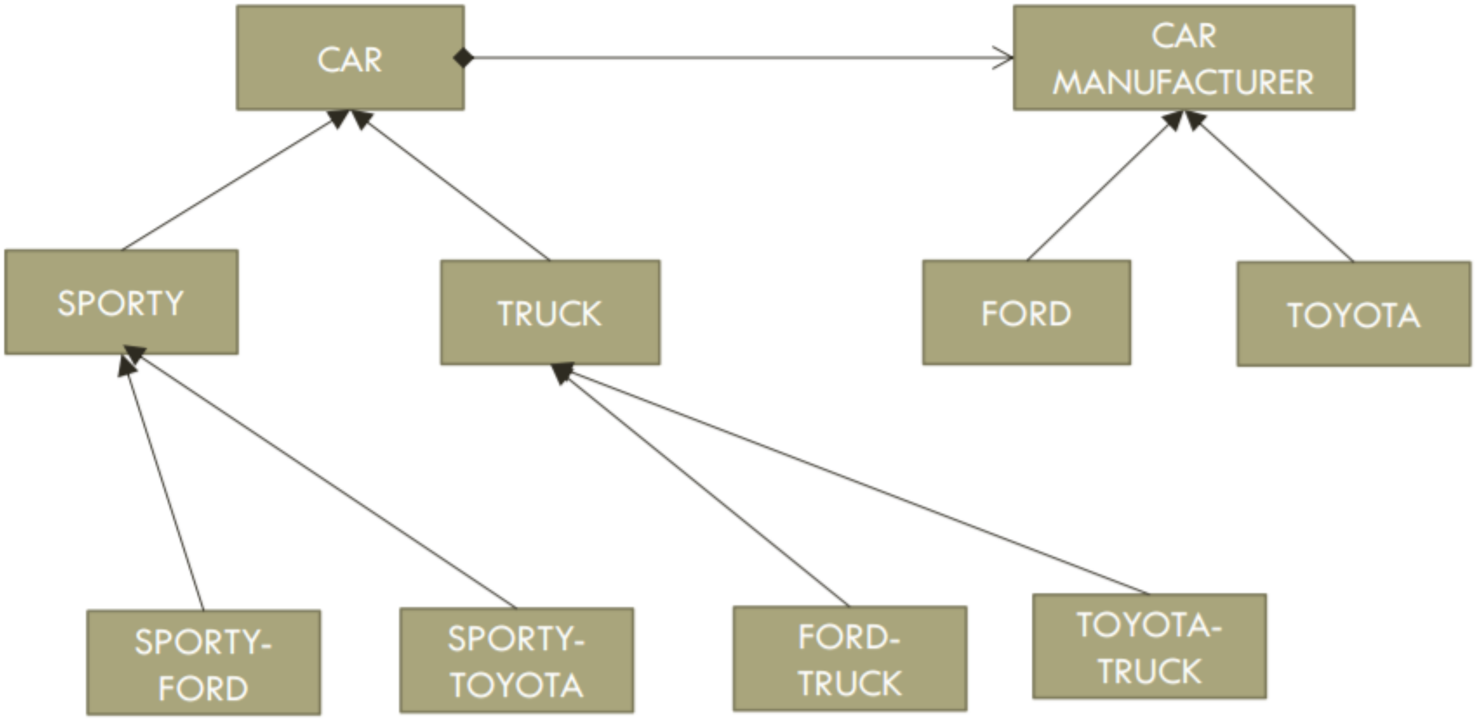
- **Applicability**
  - When interface & implementation should vary independently
  - Require a uniform interface to interchangeable class hierarchies

BRIDGE



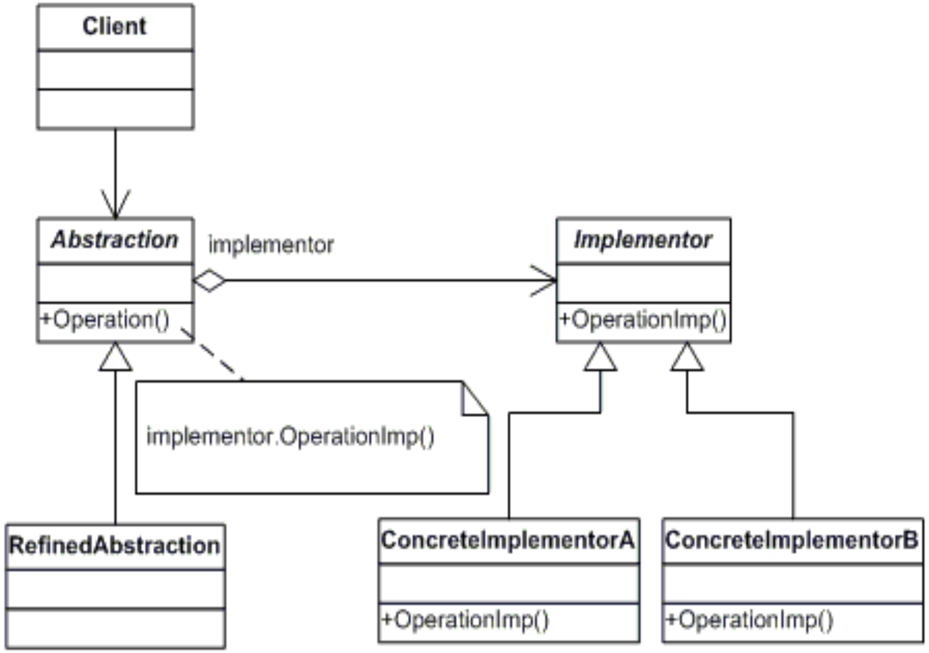
Can this hierarchy be simplified and easy to understand? How?

BRIDGE


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

BRIDGE. STRUCTURE

- **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

- **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.

- **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. EXAMPLE IMPLEMENTATION

```
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. EXAMPLE IMPLEMENTATION

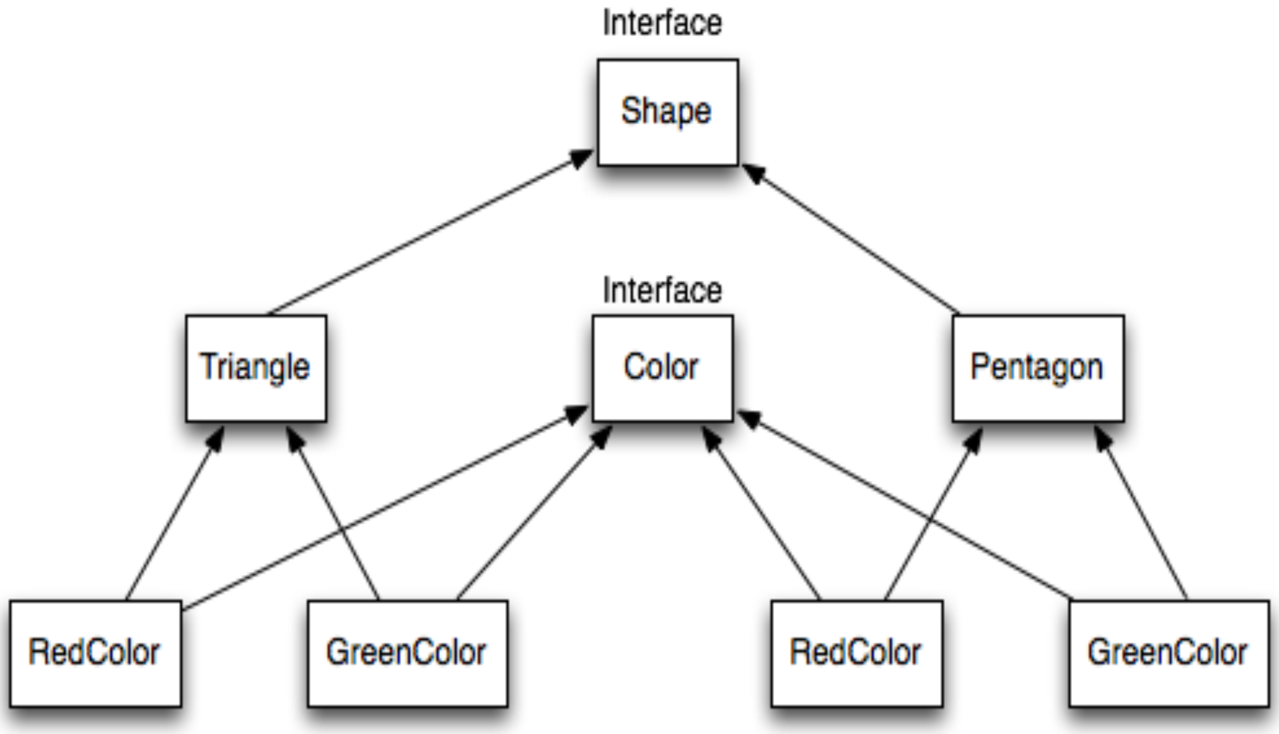
```
public class Sporty extends Car {
 public Sporty(CarManufator manufactor){
     super(manufactor);
     System.out.println(manufactor.
        getManufactor()+“ for Sporty car”);
 }
}
public class Truck extends Car {
 public Truck(CarManufator manufactor) {
     super(manufactor);
     System.out.println(manufactor.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



How you will refactor the following class hierarchy in order to follow bridge pattern?

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 Issues**
  - 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

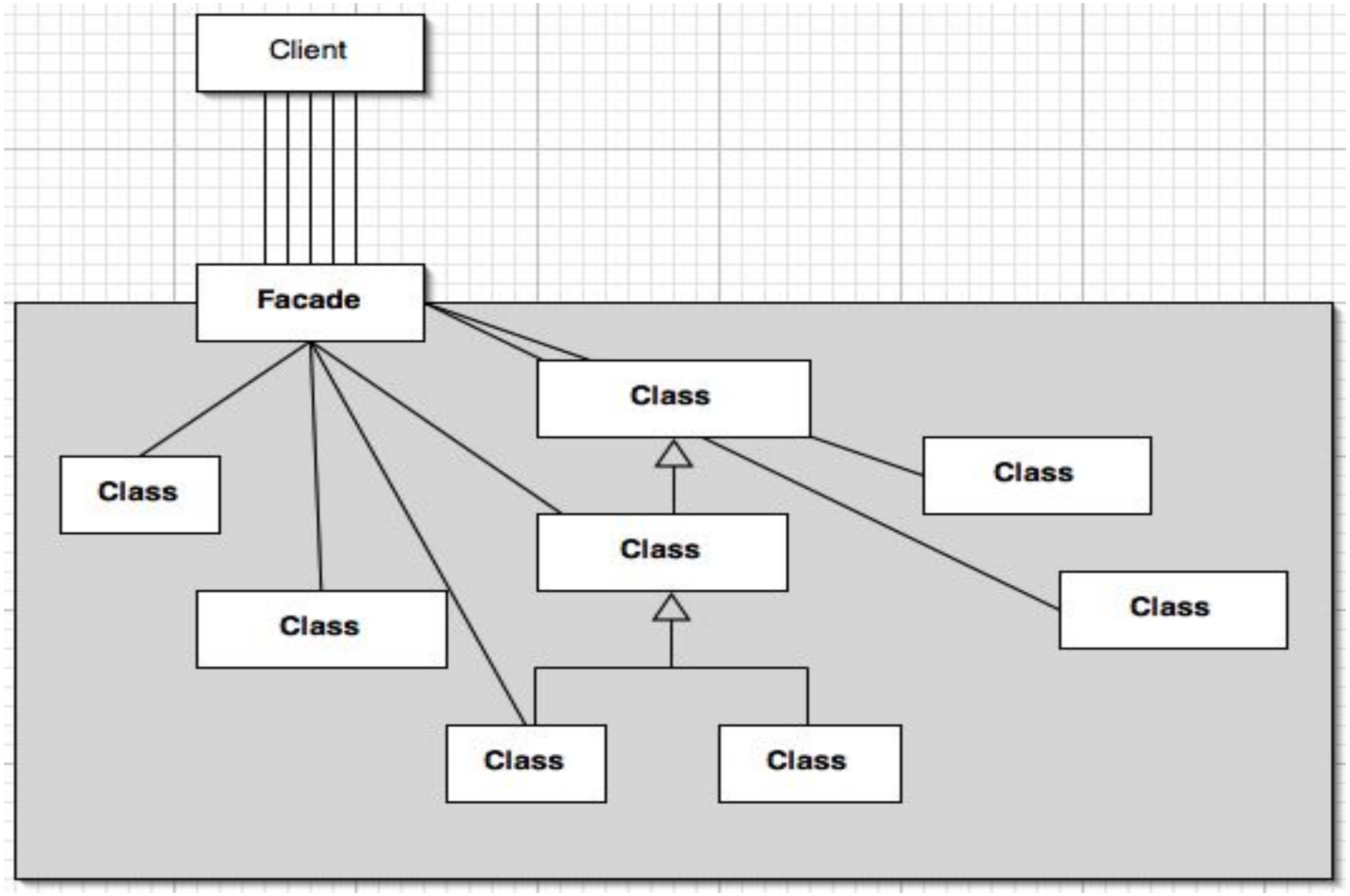
STRUCTURAL PATTERNS

- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

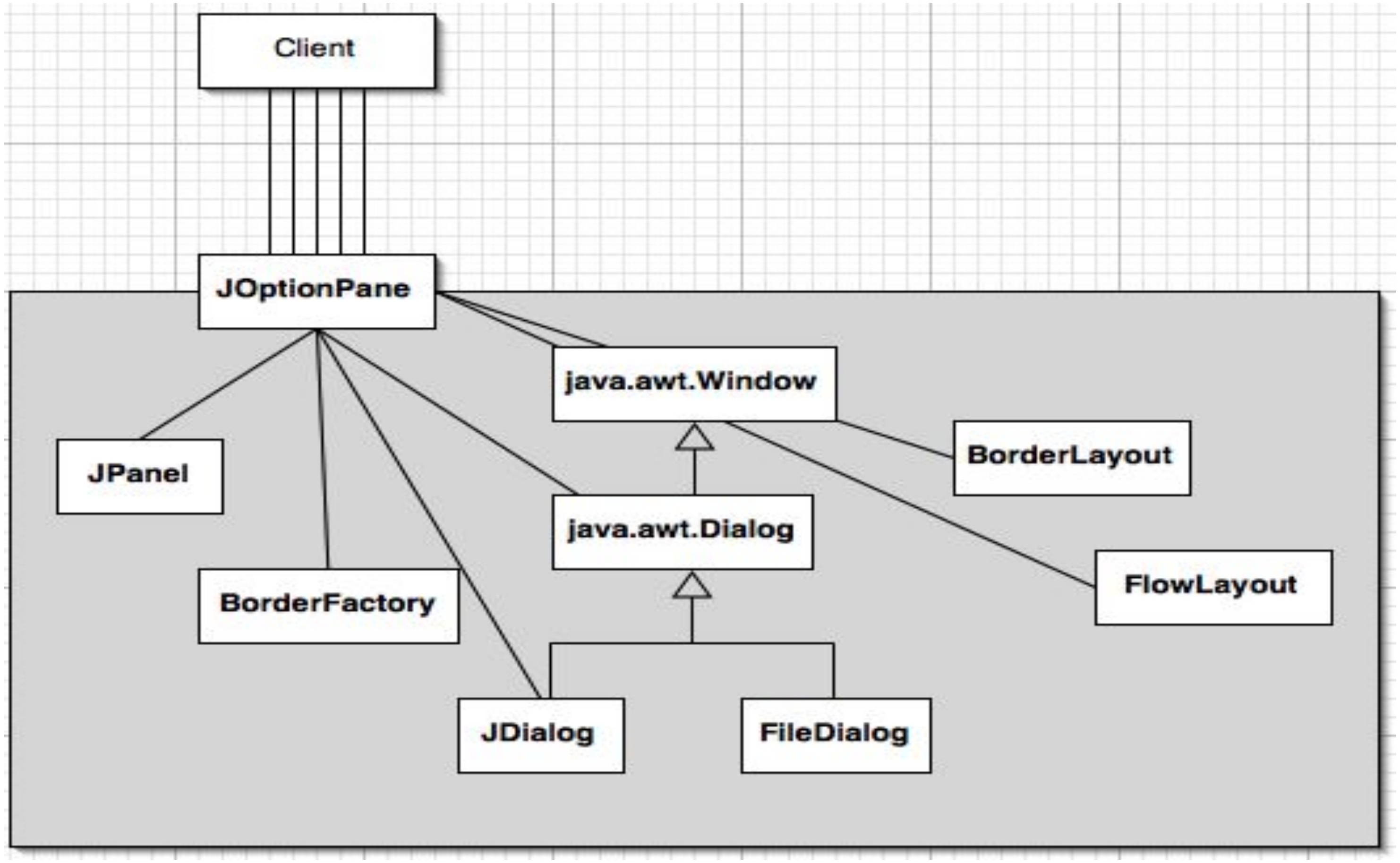
FACADE

- **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

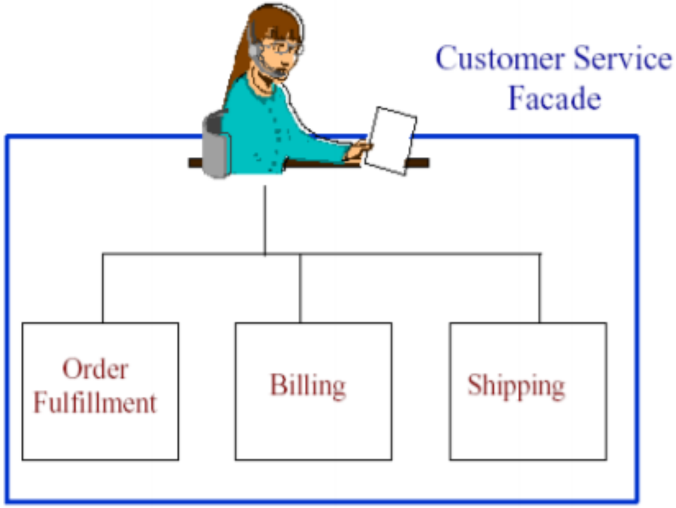
FACADE. STRUCTURE

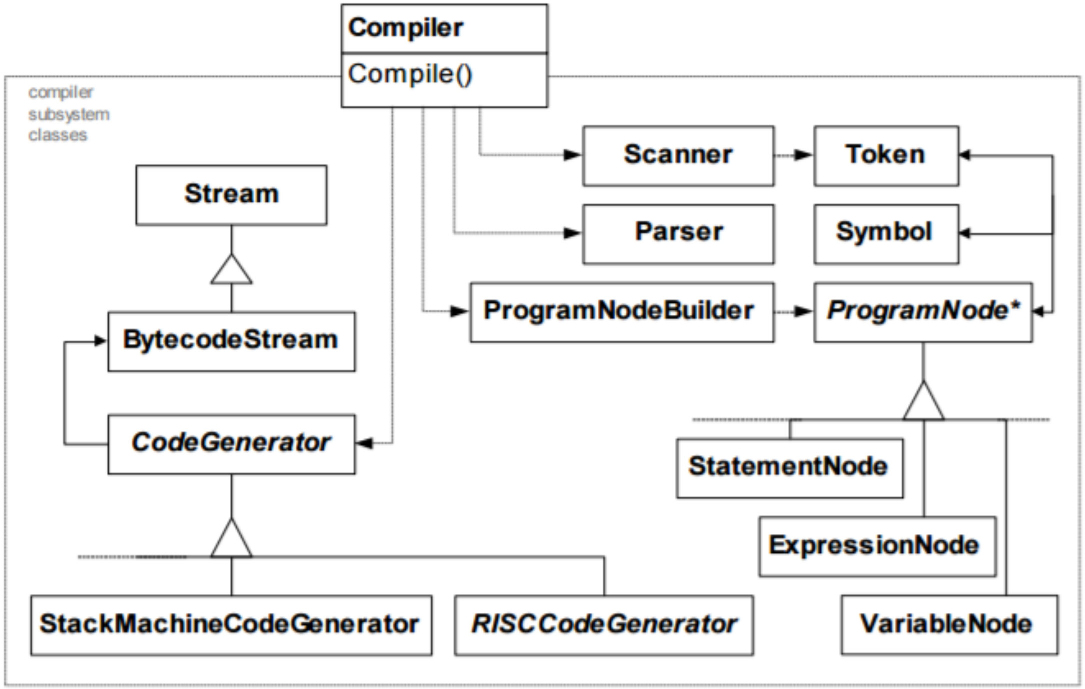


FACADE. EXAMPLE

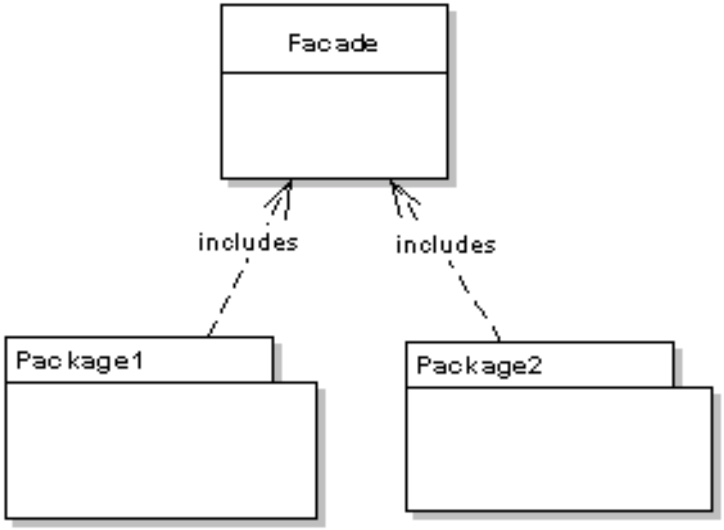


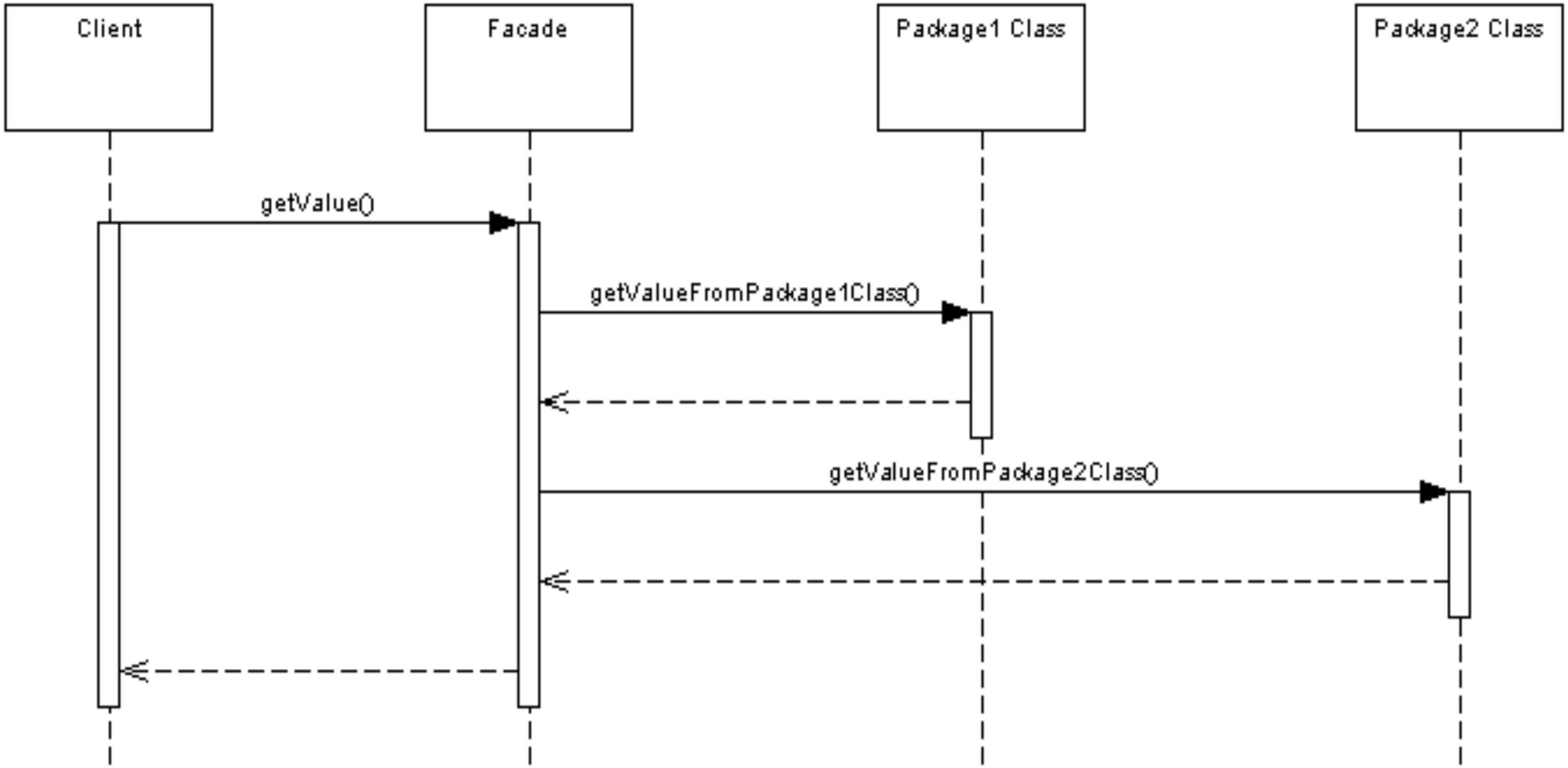
FACADE. EXAMPLE





FACADE. STRUCTURE





FACADE. EXAMPLE

- **Travel agent site that allows you to book hotels and flights**
  - we have 2 agents
    - HotelBooker
    - FlightBroker
  - HotelBooker

```
public class HotelBooker{
 public ArrayList<Hotel> getHotelNamesFor(Date from,Date to){
  //returns hotels available in the particular date range
 }
}
```

  - FlightBooker

```
public class FlightBooker{
 public ArrayList<Flight> getFlightsFor(Date from, Date to){
   //returns flights available in the particular date range
 }
}
```

FACADE. EXAMPLE

- **TravelFacade class allows the user to get their Hotel and Flight information in one call**

```
public class TravelFacade{
   private HotelBooker hotelBooker;
   private FlightBooker flightBooker;

   public void getFlightsAndHotels(Date from, Data to)  {
      ArrayList<Flight> flights =
                         flightBooker.getFlightsFor(from, to);
      ArrayList<Hotel> hotels =
                          hotelBooker.getHotelsFor(from, to);
      //process and return
   }
}
```

- **Client**

```
public class Client{
   public static void main(String[] args)   {
      TravelFacade facade = new TravelFacade();
      facade.getFlightsAndHotels(from, to);
   }
}
```

FACADE

- **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

FACADE

- **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

STRUCTURAL PATTERNS

- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

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

- ❑ **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);
      System.out.println(weight == weight2);
      String distinctString = fly + weight;
      System.out.println(distinctString ==
                                  "flyweight");
      String flyweight = (fly + weight).intern();
      System.out.println(flyweight ==
                                   "flyweight");
  }
}
```

FLYWEIGHT

- **Flyweighted strings**
  - Java Strings are flyweighted by the compiler wherever possible
- **Flyweighting works best on immutable objects**

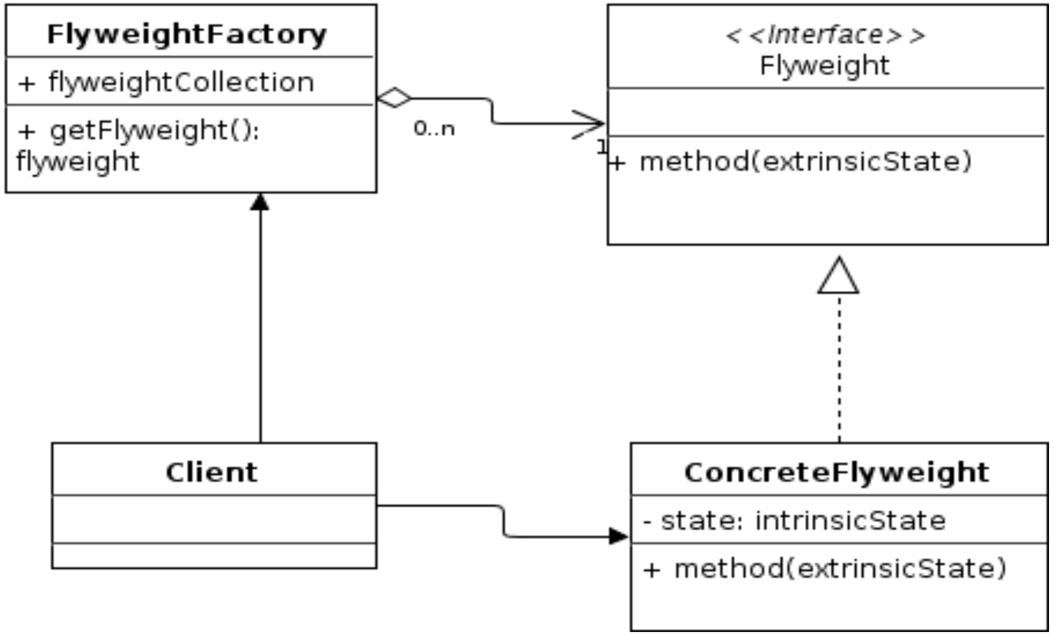
```
public class StringTest {
   public static void main(String[] args) {
      String fly = "fly", weight = "weight";
      String fly2 = "fly", weight2 = "weight";
      System.out.println(fly == fly2); //true
      System.out.println(weight == weight2); //true
      String distinctString = fly + weight;
      System.out.println(distinctString ==
                                 "flyweight"); //false

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

FLYWEIGH. 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



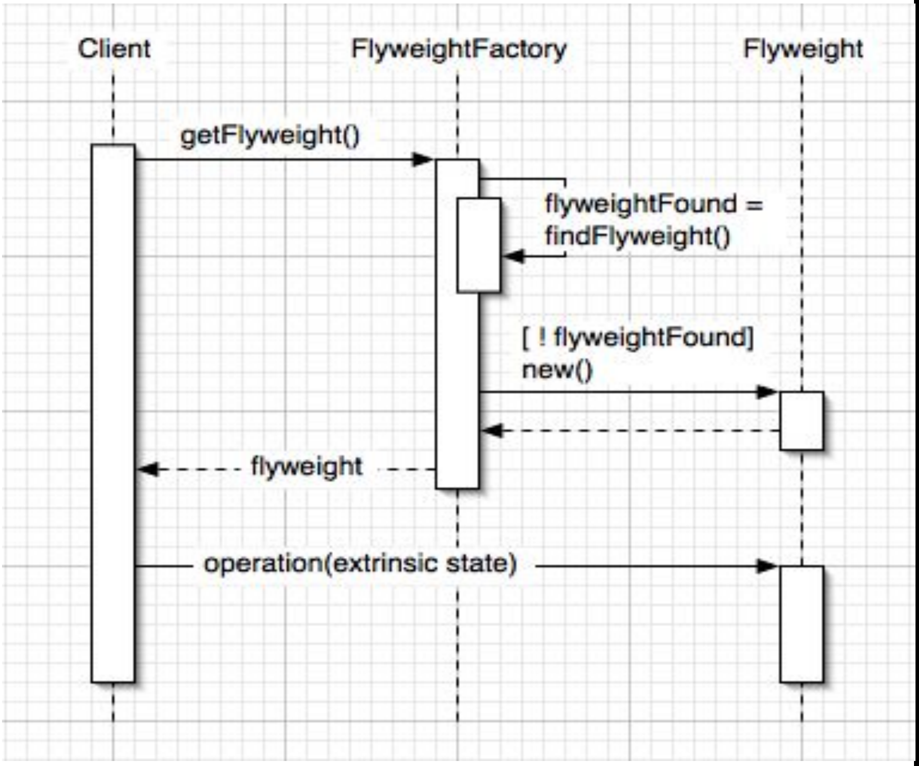
- ❑ **Flyweight**
  - ❑ Declares an interface through which flyweights can receive and act on extrinsic state.
- ❑ **ConcreteFlyweight**
  - ❑ Stores intrinsic state of the object.
  - ❑ Must be sharable.
  - ❑ Must maintain state that it is intrinsic to it, and must be able to manipulate state that 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

- ❑ Clients don't directly instantiate flyweights; instead they get them from a factory.

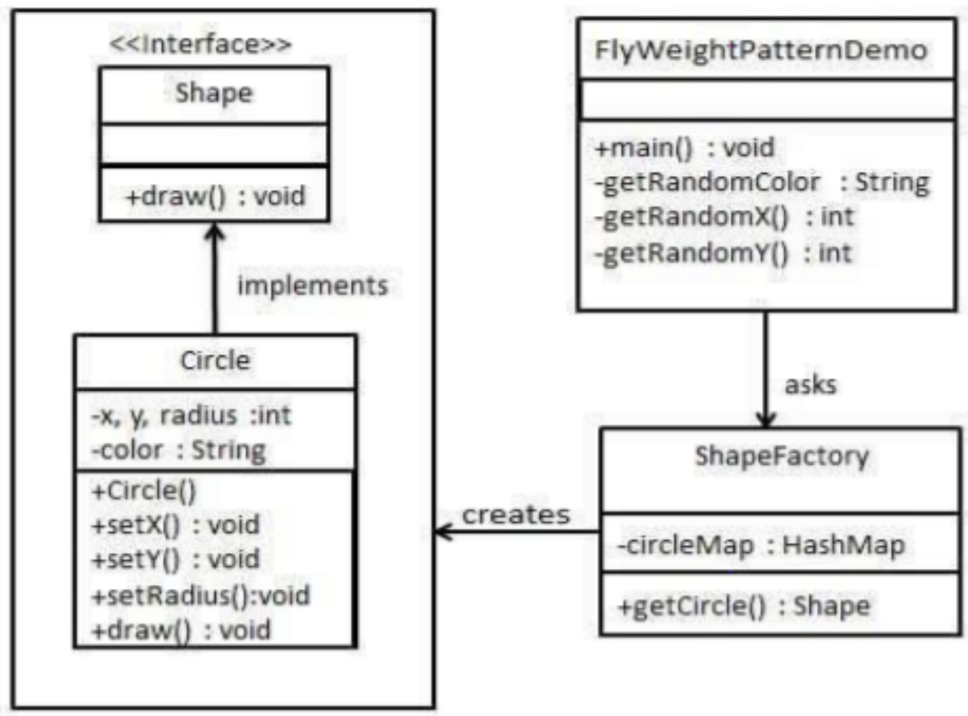
- ❑ The factory first checks to see if it has a flyweight that fits specific criteria (e.g., a blue or white line); if so, the factory returns a reference to the flyweight.

- ❑ If the factory can't locate a flyweight for the specified criteria, it instantiates one, adds it to the pool, and returns it to the client



FLYWEIGHT. EXAMPLE

- **Draw 20 circles of different locations but using only 5 objects.**
  - Only 5 objects because we have only 5 colors to draw



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;
       }
}
```

FLYWEIGHT. EXAMPLE

```
public class FlyweightPatternDemo {
  private static String getRandomColor(){
        return colors[(int)(Math.random()*colors.length)];
  }
  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 int getRandomY() { return (int)(Math.random()*100); }
   private static int getRandomX() { 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