Design Patterns

MAI XUAN TRANG, PH.D.

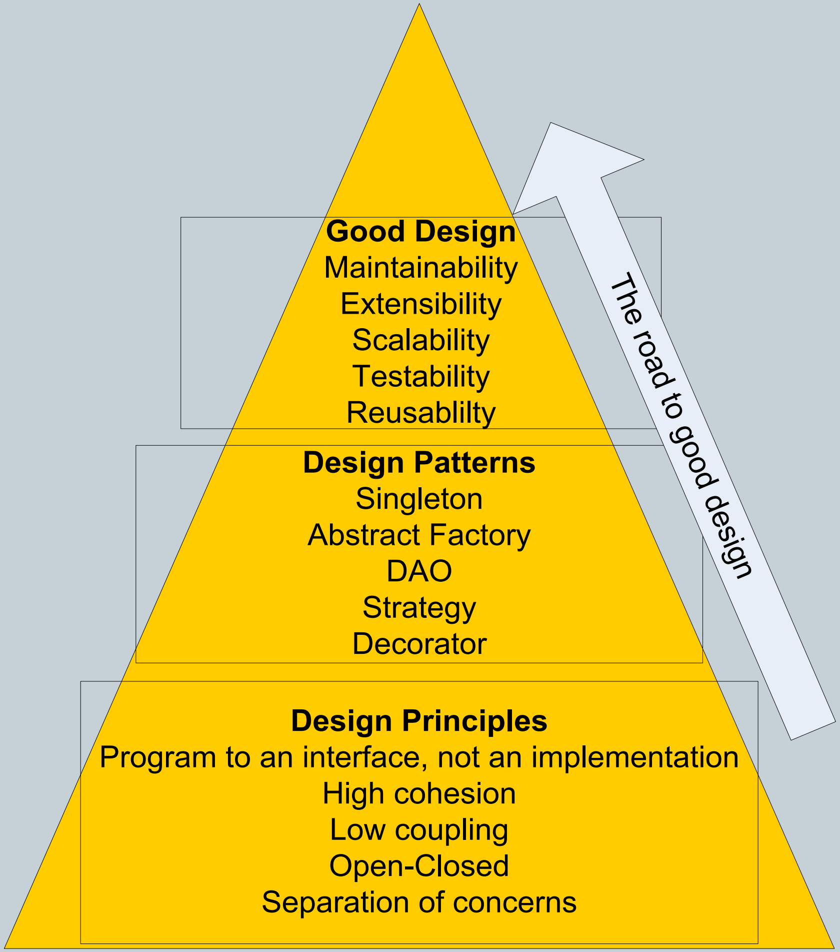
Contents

- Introduction to Design Patterns
  - What are Design Patterns?
  - Why use Design Patterns?
  - Elements of a Design Pattern
  - Design Patterns Classification
  - Pros/Cons of Design Patterns
- Popular Design Patterns
- Conclusion
- References

What are Design Patterns?

- What Are Design Patterns?
  - Wikipedia definition
    - “a design pattern is a general repeatable solution to a commonly occurring problem in software design”
  - Quote from Christopher Alexander
    - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995)

Why use Design Patterns?



Good Design
Maintainability
Extensibility
Scalability
Testability
Reusablilty

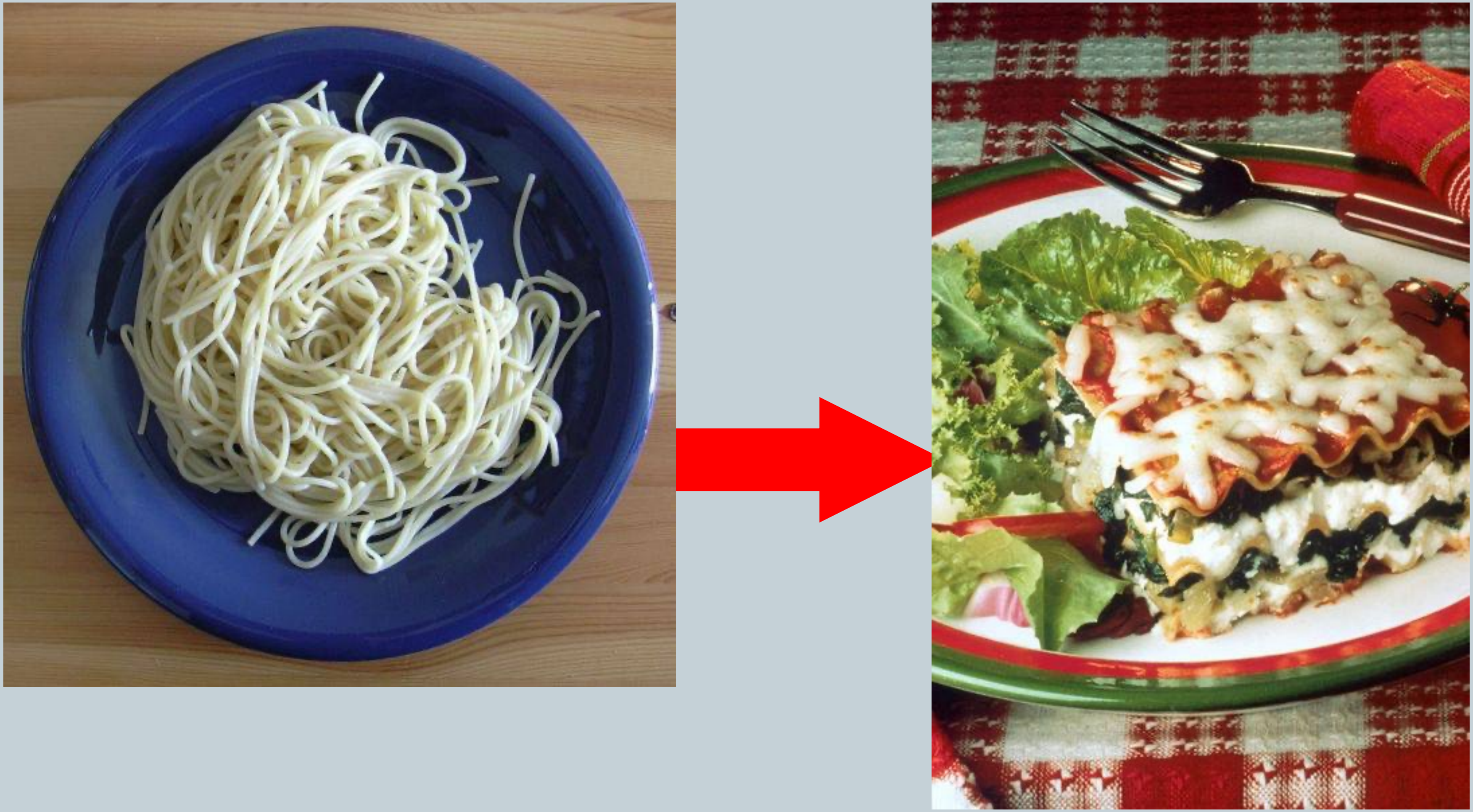
Design Patterns
Singleton
Abstract Factory
DAO
Strategy
Decorator

Design Principles
Program to an interface, not an implementation
High cohesion
Low coupling
Open-Closed
Separation of concerns

Why use Design Patterns?

- Design Objectives
  - Good Design (the “ilities”)
    - High readability and maintainability
    - High extensibility
    - High scalability
    - High testability
    - High reusability

Why use Design Patterns?



Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
  - Results of applying the pattern
  - Benefits and Costs
  - Subjective depending on concrete scenarios

Design Patterns Classification

A Pattern can be classified as

- Creational
- Structural
- Behavioral

Pros/Cons of Design Patterns

- Pros
  - Add **consistency** to designs by solving similar problems the same way, independent of language
  - Add **clarity** to design and design communication by enabling a common vocabulary
  - Improve **time** to solution by providing templates which serve as foundations for good design
  - Improve **reuse** through composition

Pros/Cons of Design Patterns

- Cons
  - Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
  - Consequences are subjective depending on concrete scenarios
  - Patterns are subject to different interpretations, misinterpretations, and philosophies
  - Patterns can be overused and abused → Anti-Patterns

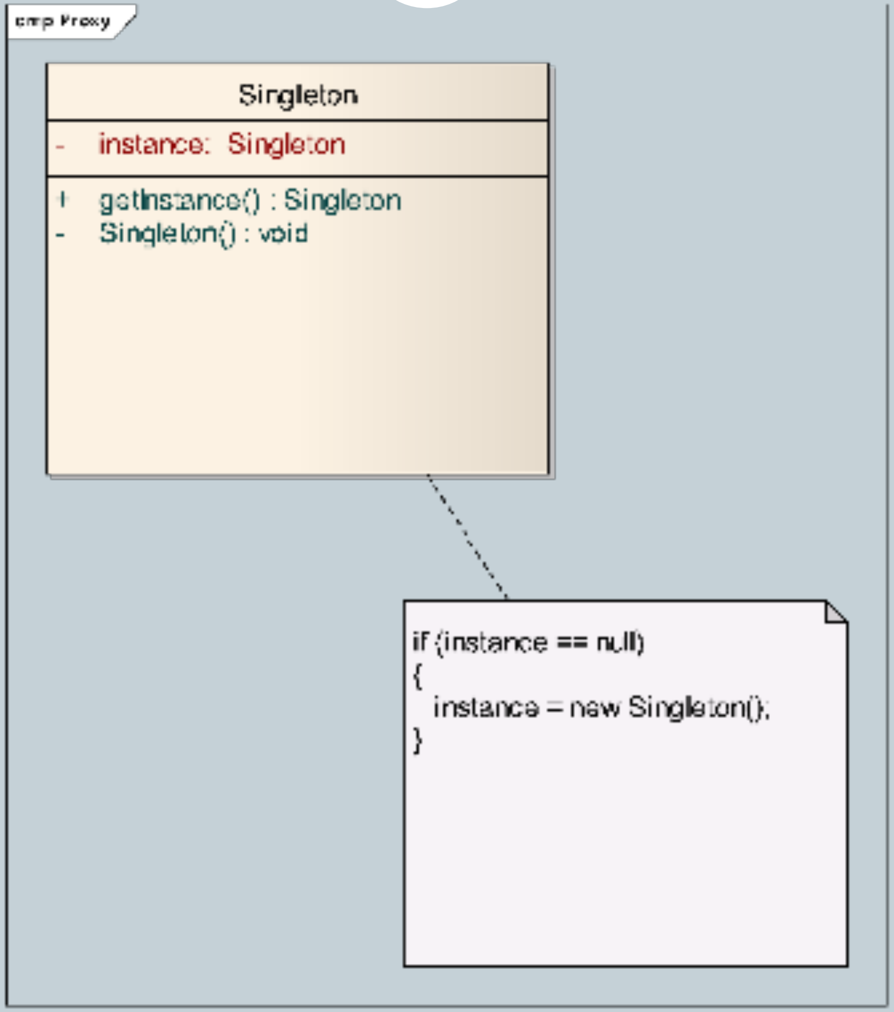
Popular Design Patterns

- Let’s take a look
  - Singleton
  - Strategy
  - Observer
  - Decorator
  - Proxy
  - Façade
  - Adapter

Singleton Definition

Ensure a class only has one instance and provide a global point of access to it.

Singleton - Class diagram



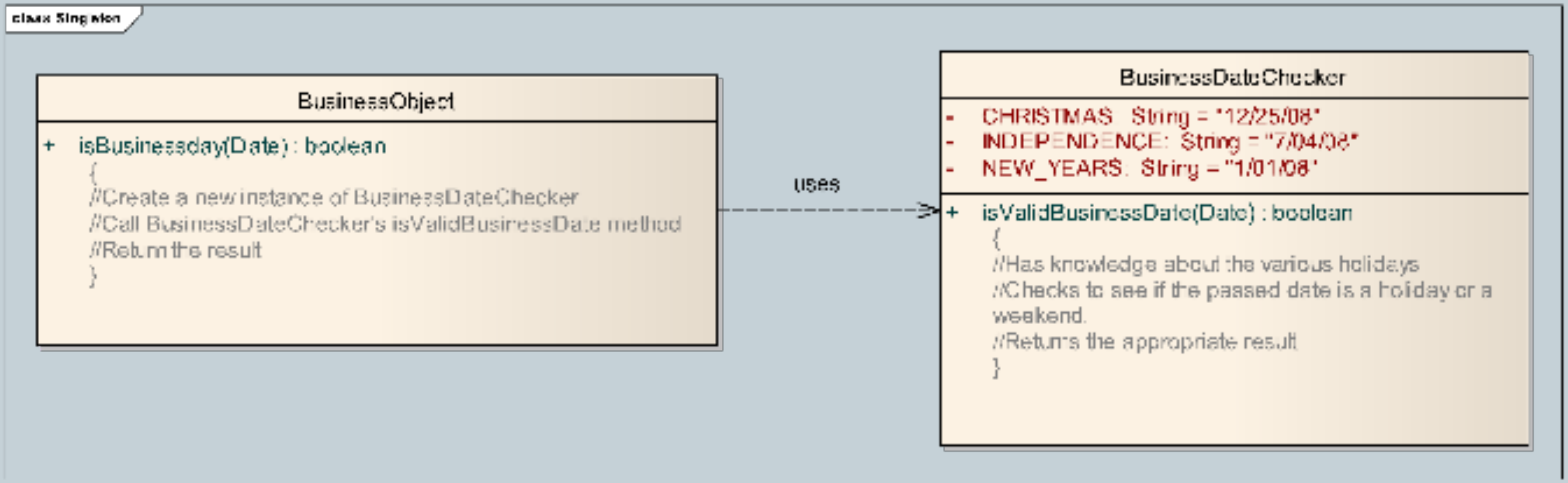
Singleton

- instance: Singleton

+ getInstance() : Singleton
- Singleton() : void

```
if (instance == null)
{
  instance = new Singleton();
}
```

Singleton - Problem



class Singleton

BusinessObject

```
+ isBusinessday(Date) : boolean
    {
    //Create a new instance of BusinessDateChecker
    //Call BusinessDateChecker's isValidBusinessDate method
    //Return the result
    }
```

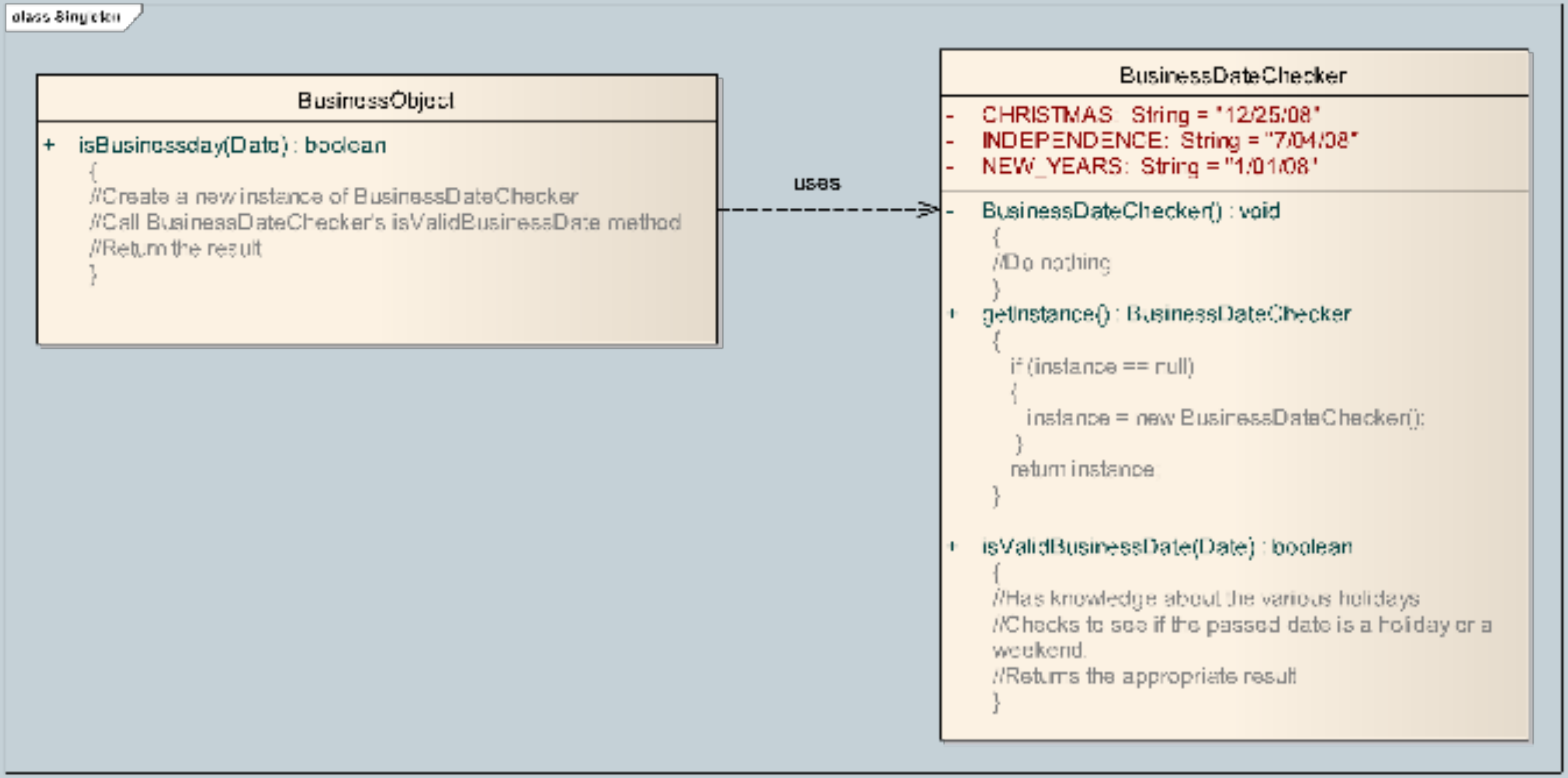
uses

BusinessDateChecker

```
- CHRISTMAS: String = "12/25/08"
- INDEPENDENCE: String = "7/04/08"
- NEW_YEARS: String = "1/01/08"
```

```
+ isValidBusinessDate(Date) : boolean
    {
    //Has knowledge about the various holidays
    //Checks to see if the passed date is a holiday or a
    weekend.
    //Returns the appropriate result
    }
```

Singleton - Solution



BusinessObject

```
+   isBusinessday(Date) : boolean
      {
      //Create a new instance of BusinessDateChecker
      //Call BusinessDateChecker's isValidBusinessDate method
      //Return the result
      }
```

BusinessDateChecker

```
-   CHRISTMAS:  String = "12/25/08"
-   INDEPENDENCE:  String = "7/04/08"
-   NEW_YEARS:  String = "1/01/08"

-   BusinessDateChecker() : void
      {
      //Do nothing
      }
+   getInstance() : BusinessDateChecker
      {
        if (instance == null)
        {
          instance = new BusinessDateChecker();
        }
        return instance
      }

+   isValidBusinessDate(Date) : boolean
      {
      //Has knowledge about the various holidays
      //Checks to see if the passed date is a holiday or a
      weekend
      //Returns the appropriate result
      }
```

Singleton

- Pros
  - Increases performance
  - Prevents memory wastage
  - Increases global data sharing

- Cons
  - Results in multithreading issues

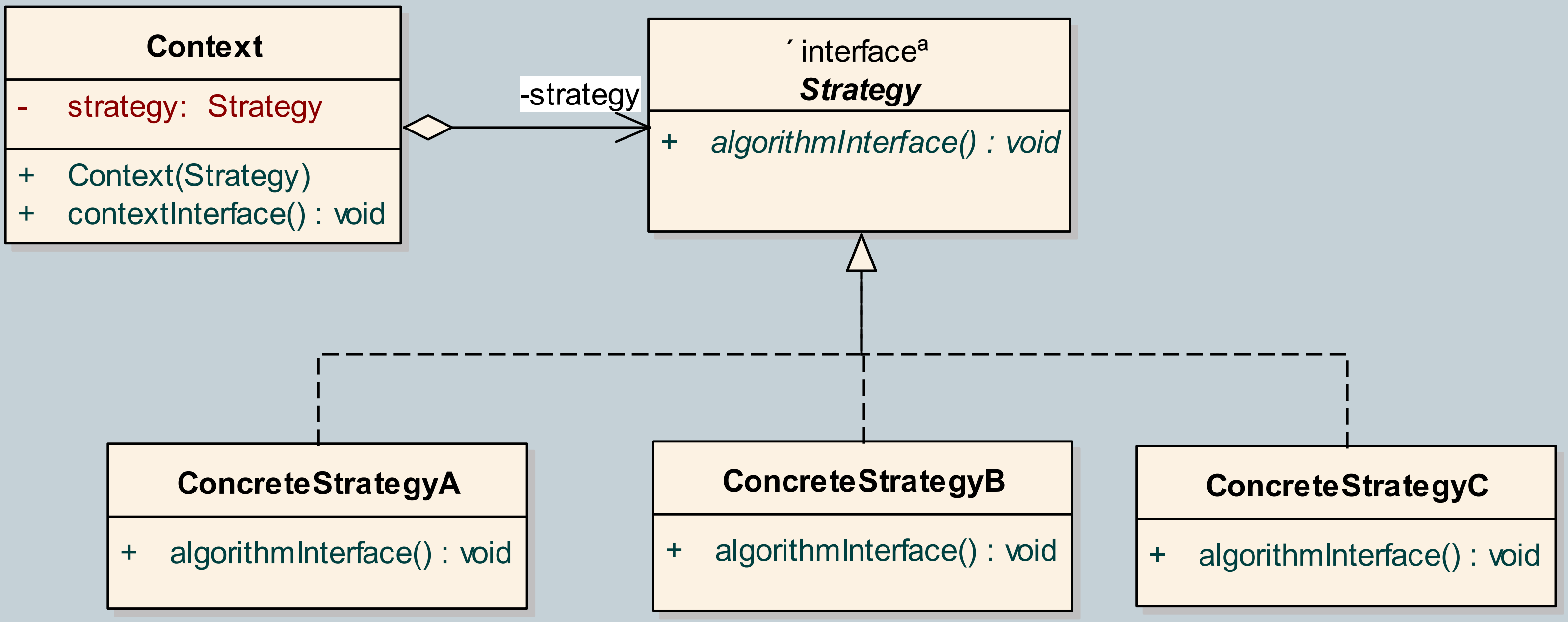
Strategy Definition

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance

Strategy - Class diagram



Context

- strategy: Strategy

+ Context(Strategy)

+ contextInterface() : void

-strategy

´ interfaceª

Strategy

+ *algorithmInterface() : void*

ConcreteStrategyA

+ algorithmInterface() : void

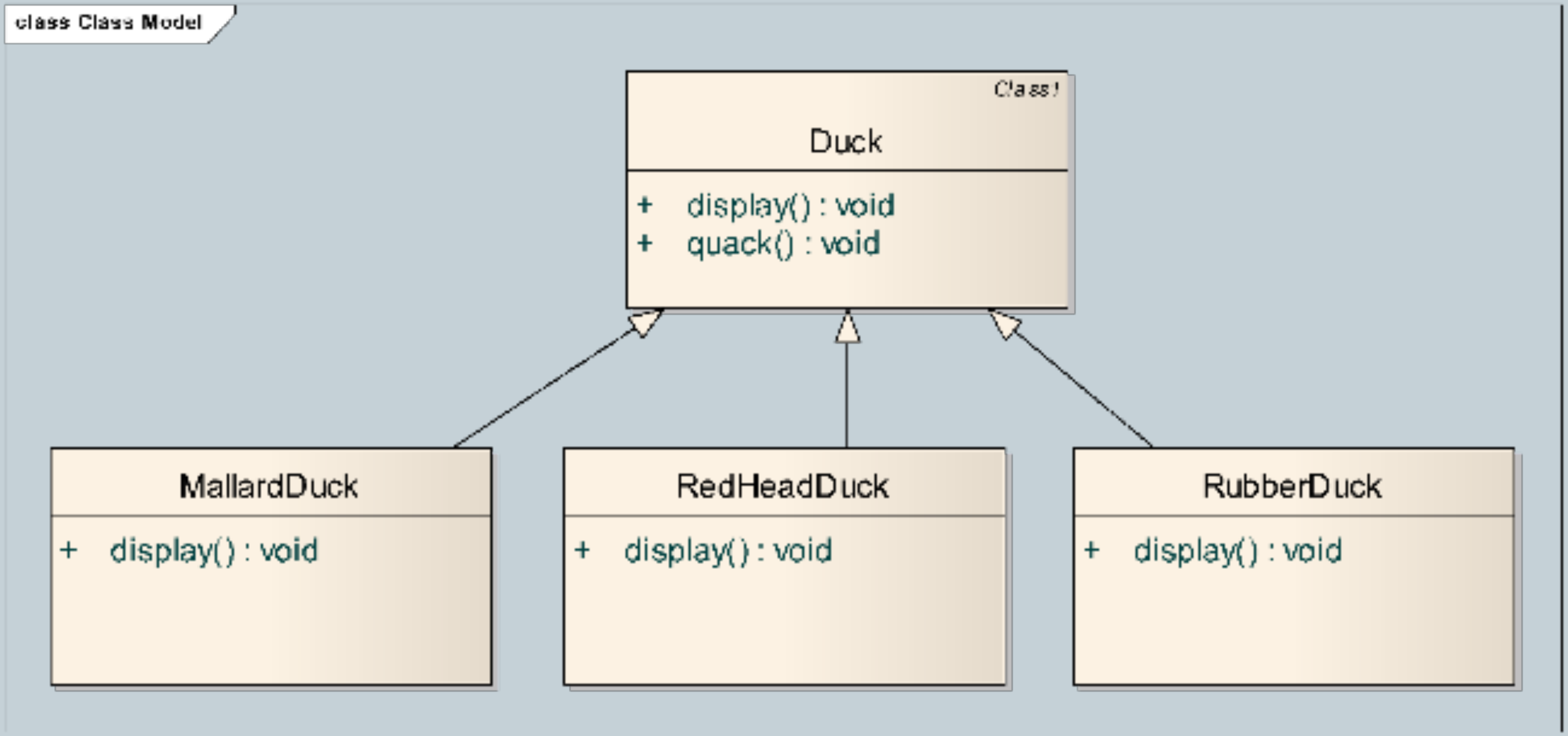
ConcreteStrategyB

+ algorithmInterface() : void

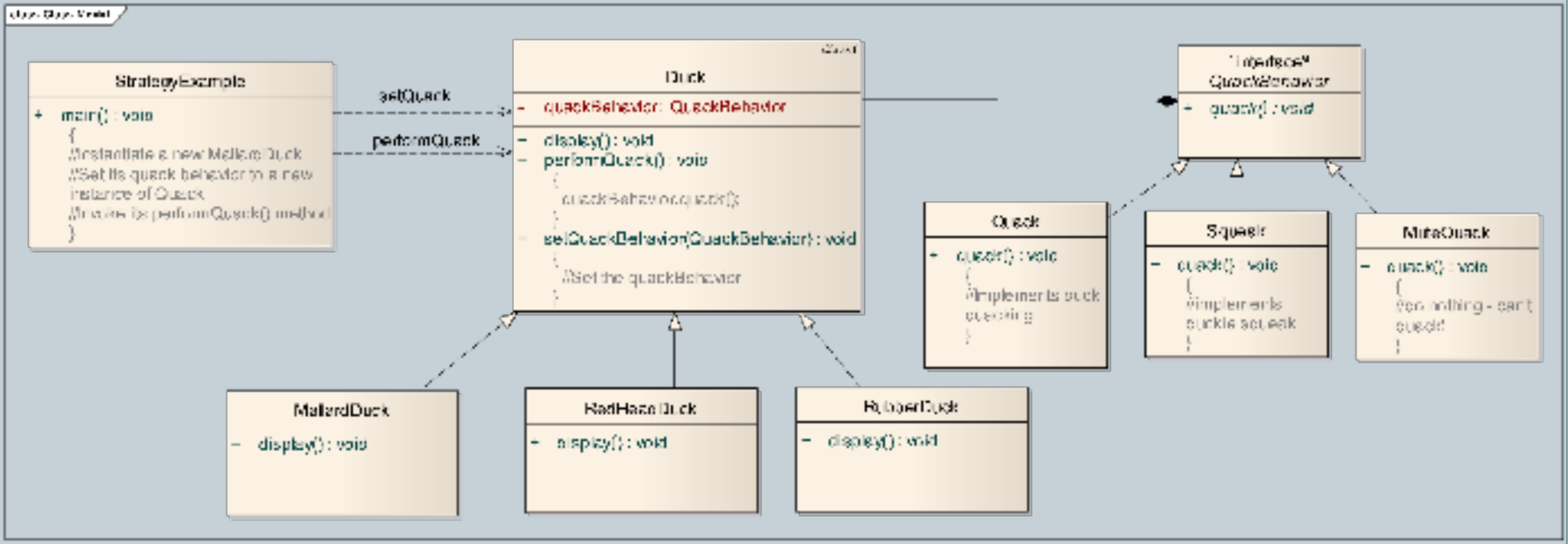
ConcreteStrategyC

+ algorithmInterface() : void

Strategy - Problem



Strategy - Solution



Strategy

- Pros
  - Provides encapsulation
  - Hides implementation
  - Allows behavior change at runtime
- Cons
  - Results in complex, hard to understand code if overused

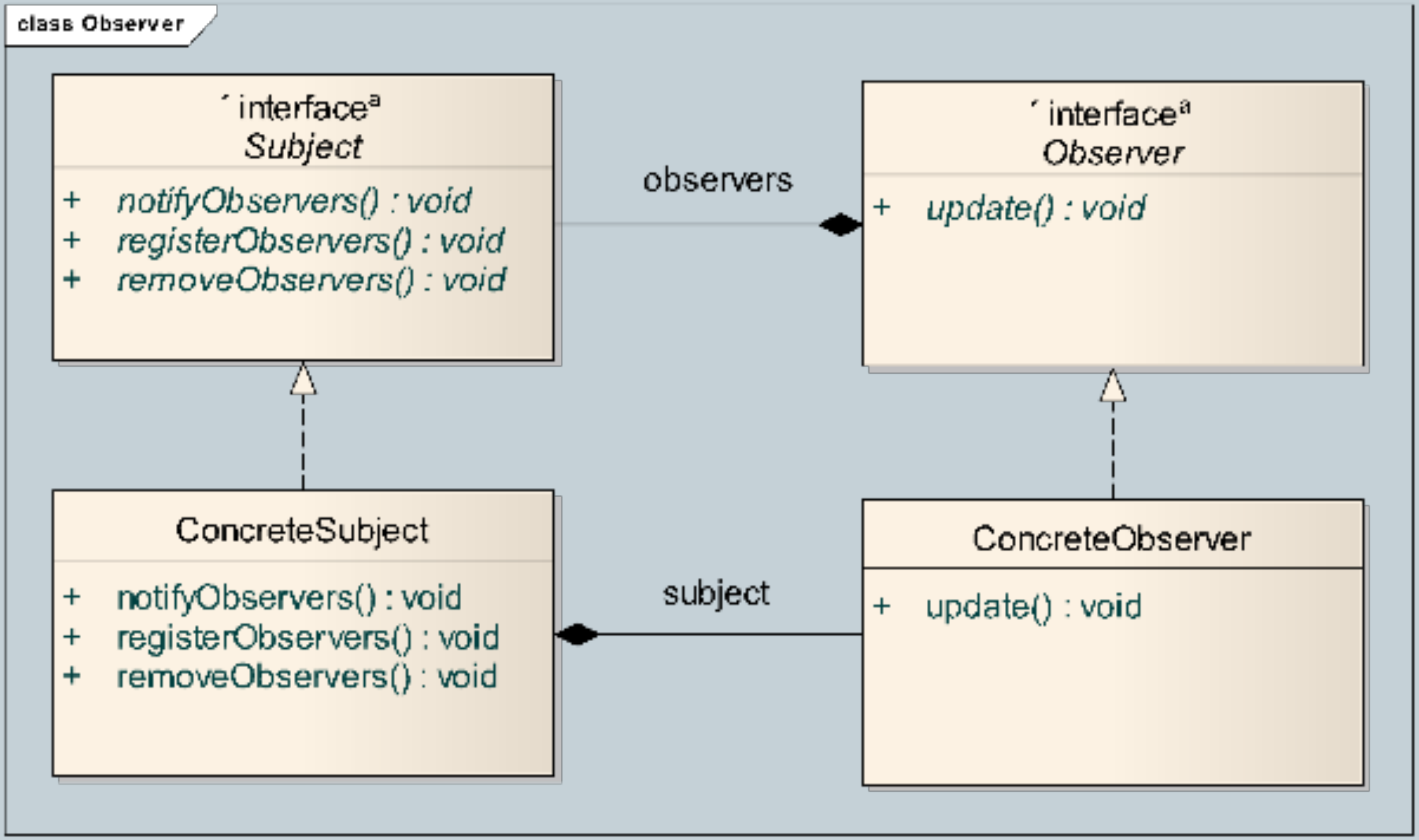
Observer Definition

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

Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact

Observer - Class diagram



class Observer

´ interfaceª
Subject

+ *notifyObservers() : void*
+ *registerObservers() : void*
+ *removeObservers() : void*

observers

´ interfaceª
Observer

+ *update() : void*

ConcreteSubject

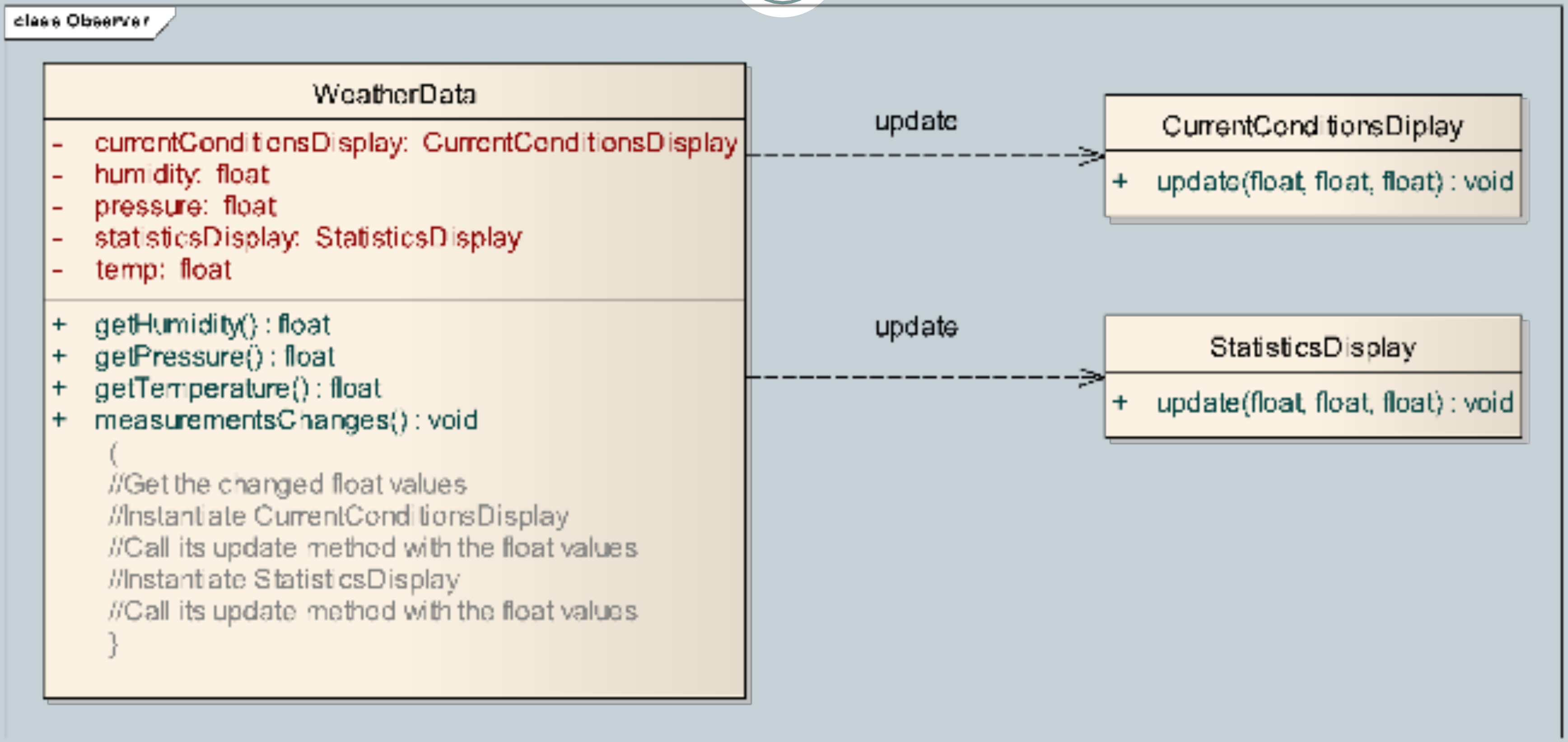
+ notifyObservers() : void
+ registerObservers() : void
+ removeObservers() : void

subject

ConcreteObserver

+ update() : void

Observer - Problem



class Observer

WeatherData

```
- currentConditionsDisplay: CurrentConditionsDisplay
- humidity: float
- pressure: float
- statisticsDisplay: StatisticsDisplay
- temp: float

+ getHumidity() : float
+ getPressure() : float
+ getTemperature() : float
+ measurementsChanges() : void
    {
    //Get the changed float values
    //Instantiate CurrentConditionsDisplay
    //Call its update method with the float values
    //Instantiate StatisticsDisplay
    //Call its update method with the float values
    }
```

update

CurrentConditionsDiplay

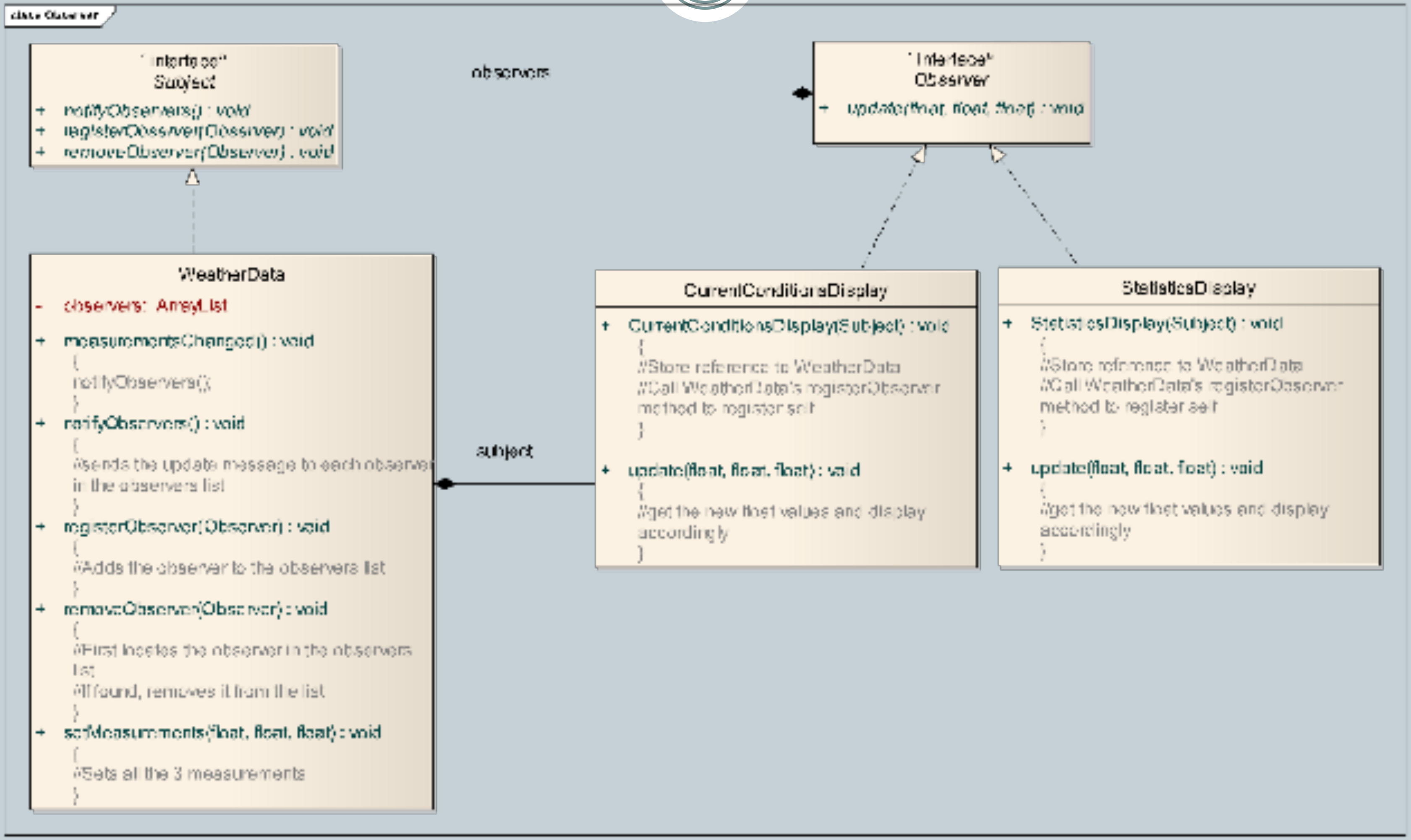
```
+ update(float, float, float) : void
```

update

StatisticsDisplay

```
+ update(float, float, float) : void
```

Observer - Solution



class Observer

«interface»
Subject

+ notifyObservers() : void
+ registerObserver(Observer) : void
+ removeObserver(Observer) : void

observers

«interface»
Observer

+ update(float, float, float) : void

WeatherData

- observers: ArrayList

+ measurementsChanged() : void
{
notifyObservers();
}

+ notifyObservers() : void
{
//sends the update message to each observer in the observers list
}

+ registerObserver(Observer) : void
{
//Adds the observer to the observers list
}

+ removeObserver(Observer) : void
{
//First locates the observer in the observers list
//If found, removes it from the list
}

+ setMeasurements(float, float, float) : void
{
//Sets all the 3 measurements
}

subject

CurrentConditionsDisplay

+ CurrentConditionsDisplay(Subject) : void
{
//Store reference to WeatherData
//Call WeatherData's registerObserver method to register self
}

+ update(float, float, float) : void
{
//get the new float values and display accordingly
}

StatisticsDisplay

+ StatisticsDisplay(Subject) : void
{
//Store reference to WeatherData
//Call WeatherData's registerObserver method to register self
}

+ update(float, float, float) : void
{
//get the new float values and display accordingly
}

Observer

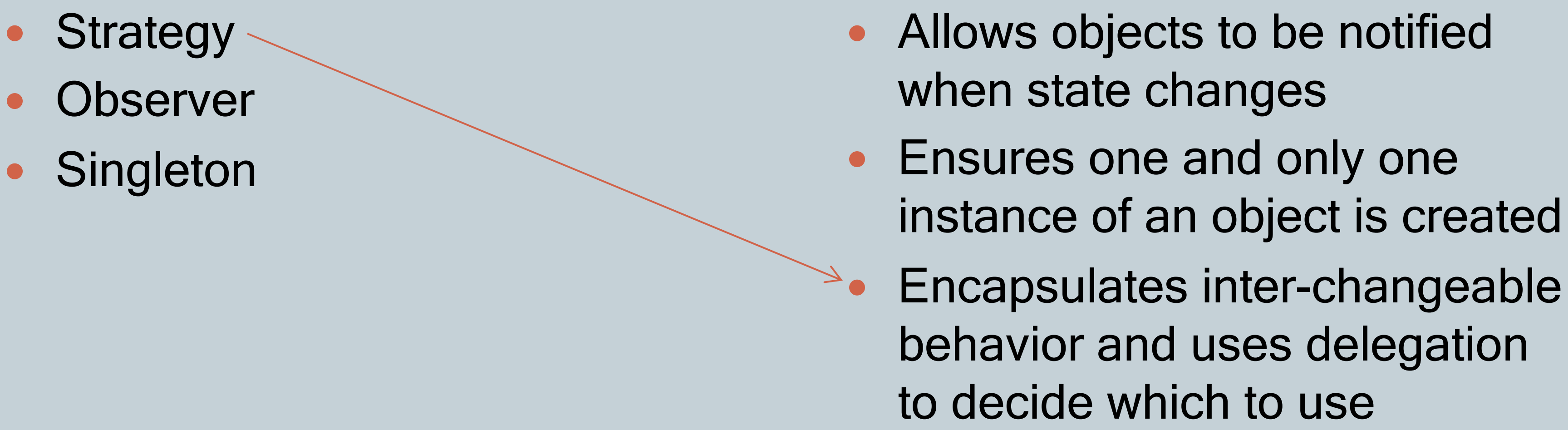
- Pros
  - Abstracts coupling between Subject and Observer
  - Supports broadcast communication
  - Supports unexpected updates
  - Enables reusability of subjects and observers independently of each other
- Cons
  - Exposes the Observer to the Subject
  - Exposes the Subject to the Observer

Patterns & Definitions - Group 1

- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

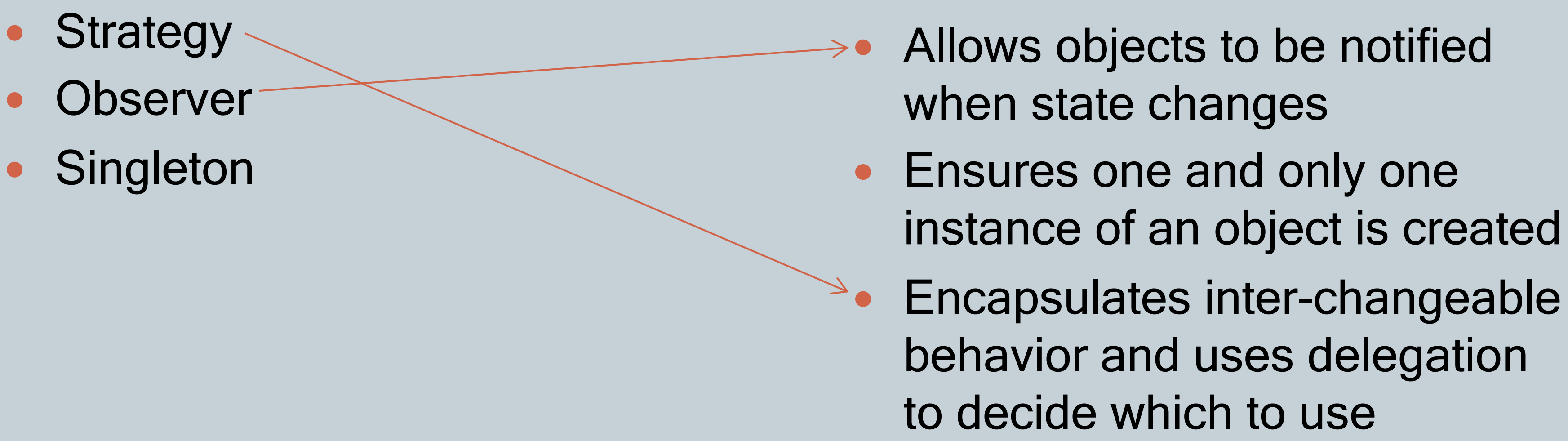
Patterns & Definitions - Group 1



- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

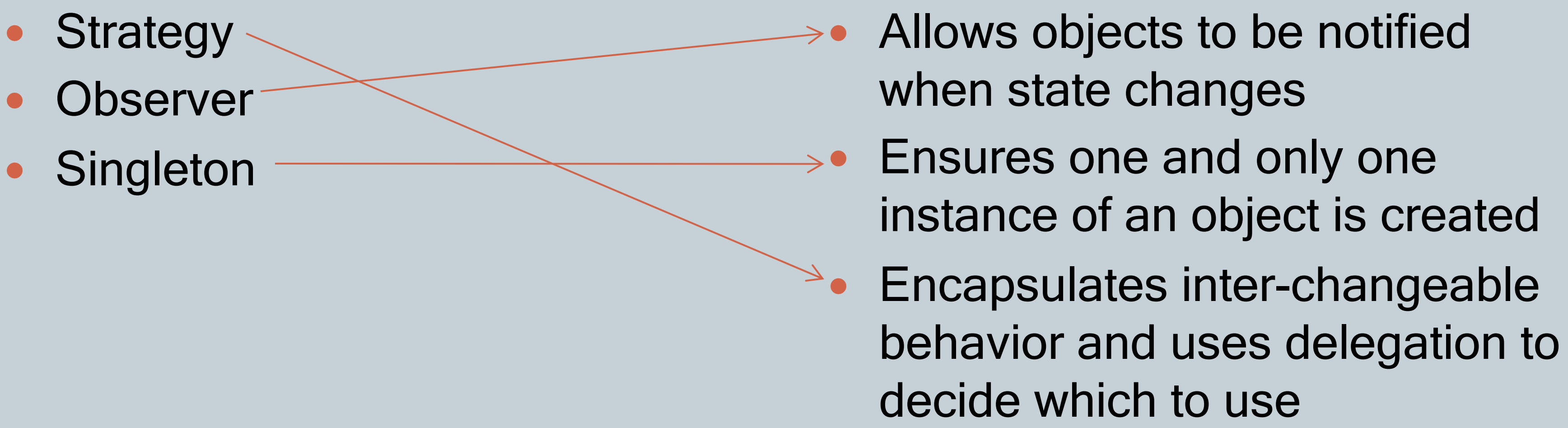
Patterns & Definitions - Group 1



- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

Patterns & Definitions - Group 1



- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

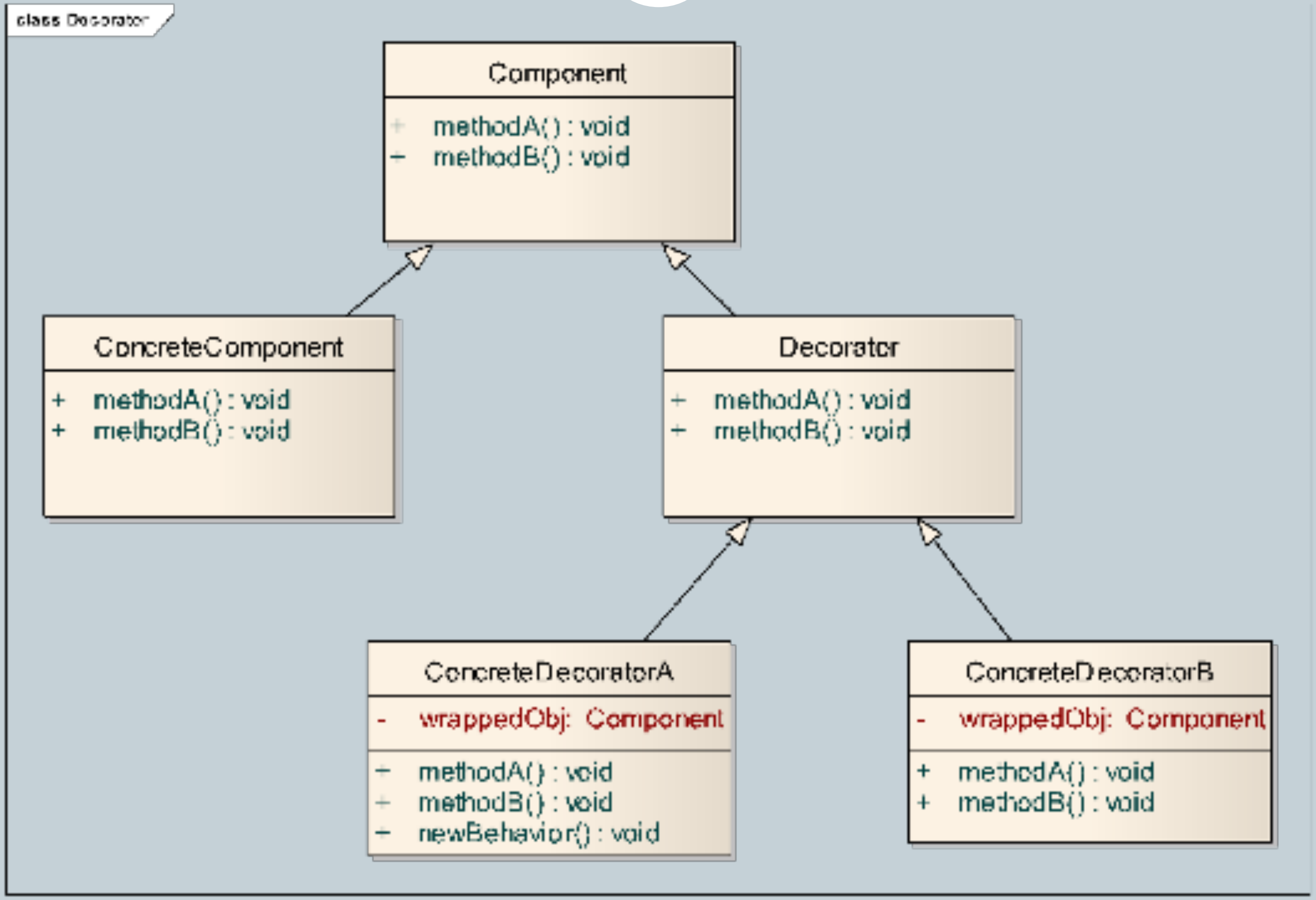
Decorator Definition

Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

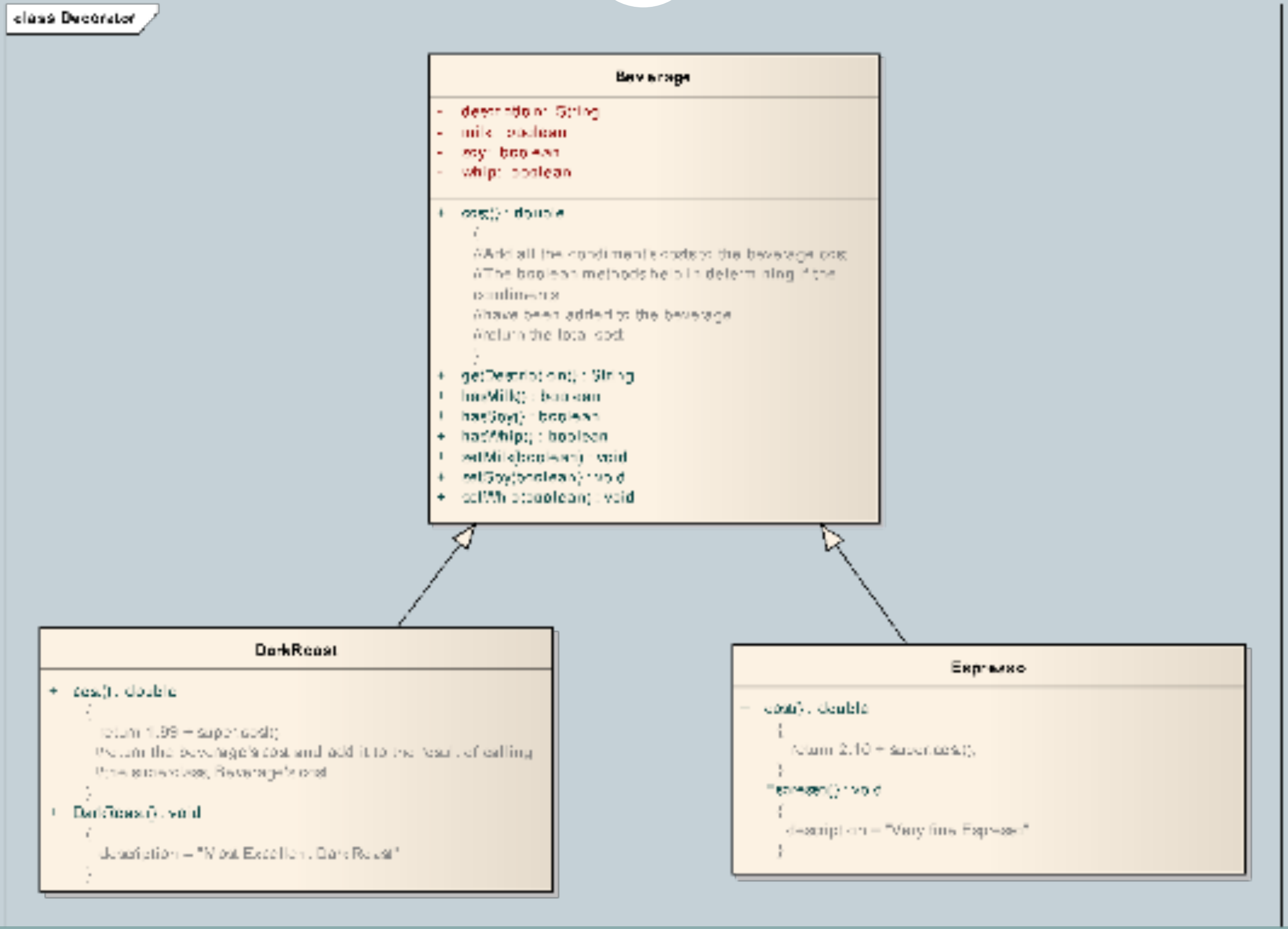
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification

Decorator - Class diagram



Decorator - Problem



```
Beverage

- description: String
- milk: boolean
- soy: boolean
- whip: boolean

+ cost() : double
    {
    //Add all the condiments costs to the beverage cost
    //The boolean methods help in determining if the
    condiments
    //have been added to the beverage
    //return the total cost
    }
+ getDescription() : String
+ hasMilk() : boolean
+ hasSoy() : boolean
+ hasWhip() : boolean
+ setMilk(boolean) : void
+ setSoy(boolean) : void
+ setWhip(boolean) : void
```

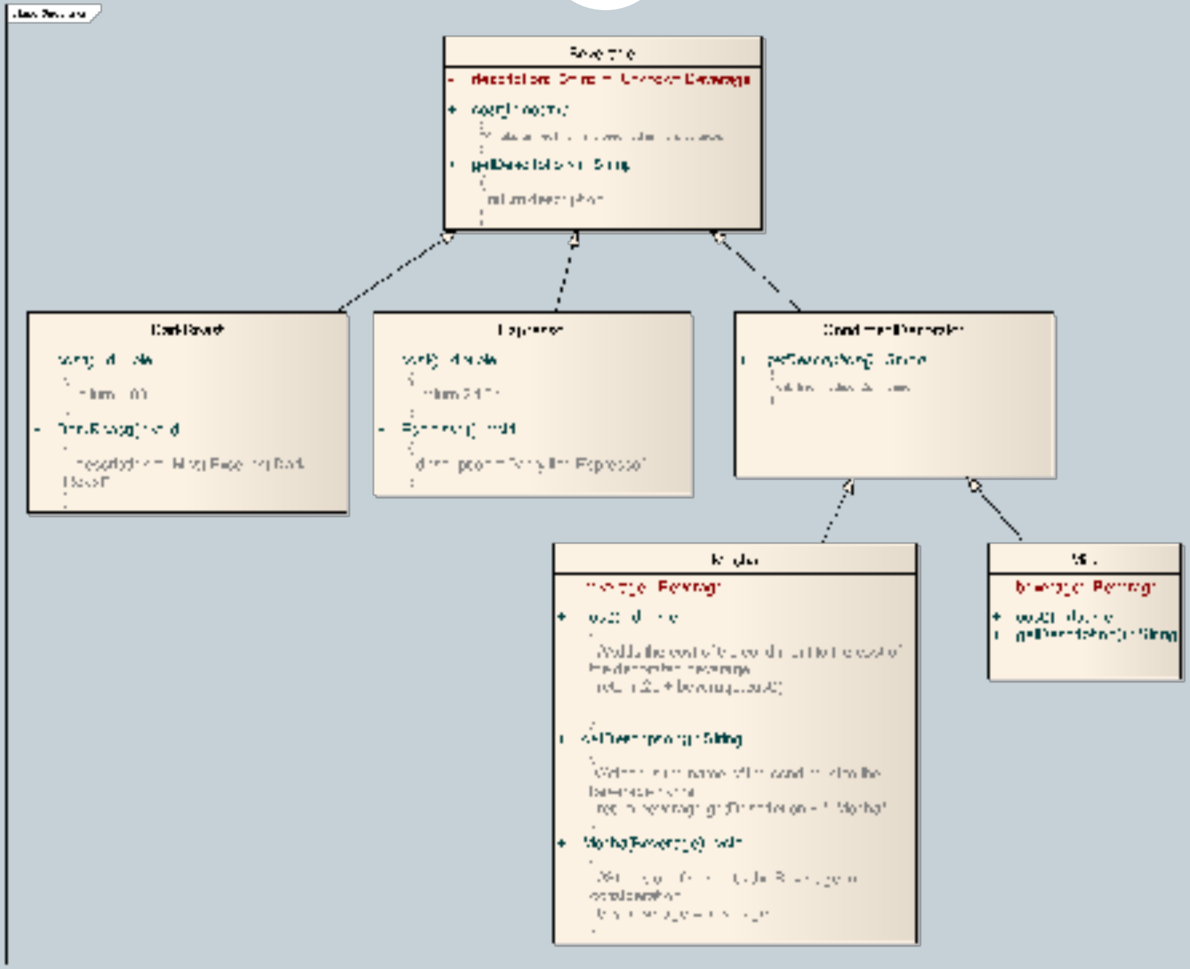
```
DarkRoast

+ cost() : double
    {
     return 1.99 + super.cost();
    //Return the beverage's cost and add it to the result of calling
    //the superclass, Beverage's cost
    }
+ DarkRoast() : void
    {
     description = "Most Excellent Dark Roast"
    }
```

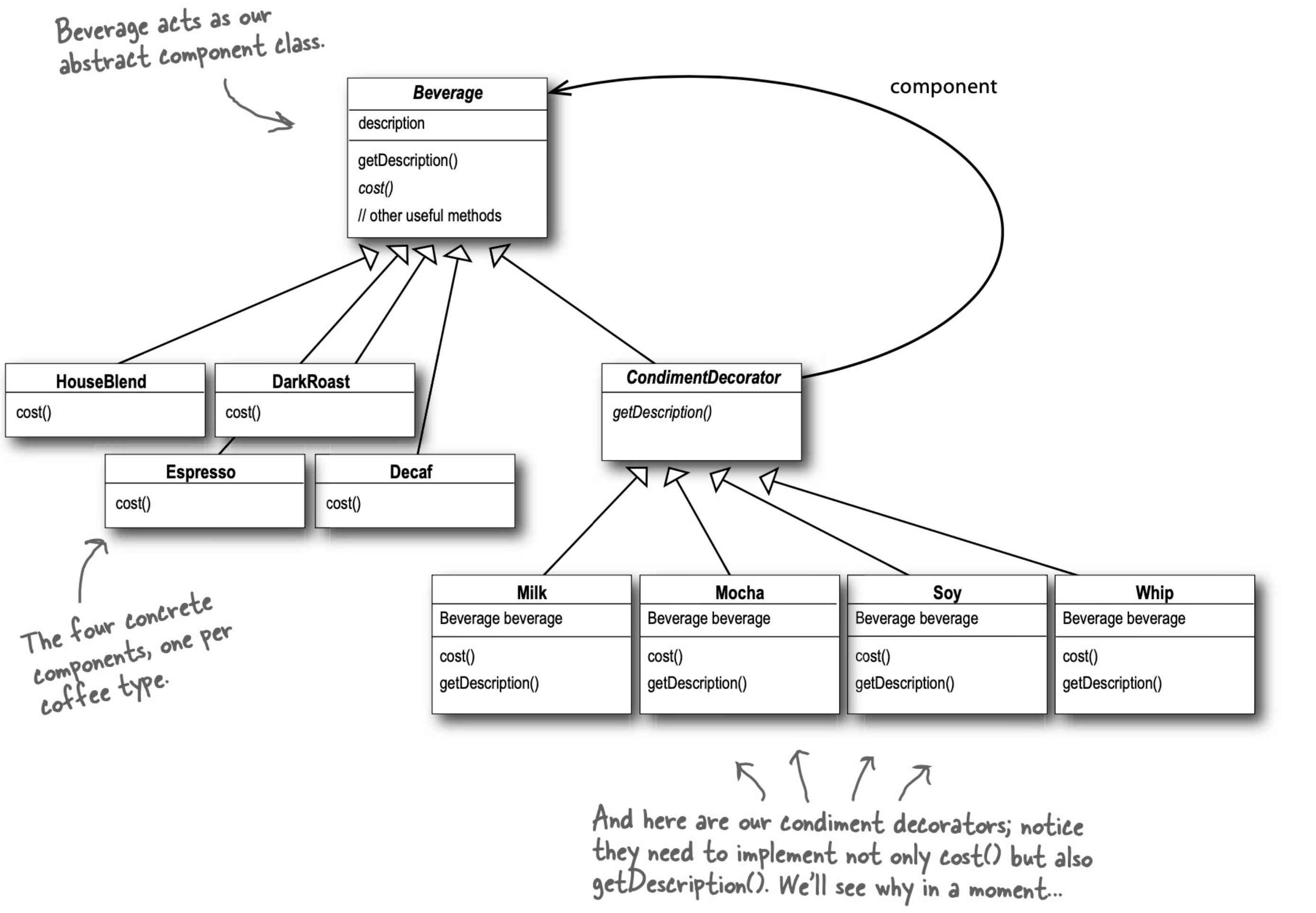
```
Espresso

+ cost() : double
    {
     return 2.10 + super.cost();
    }
 Espresso() : void
    {
     description = "Very fine Espresso"
    }
```

Decorator - Solution



Decorator - Example

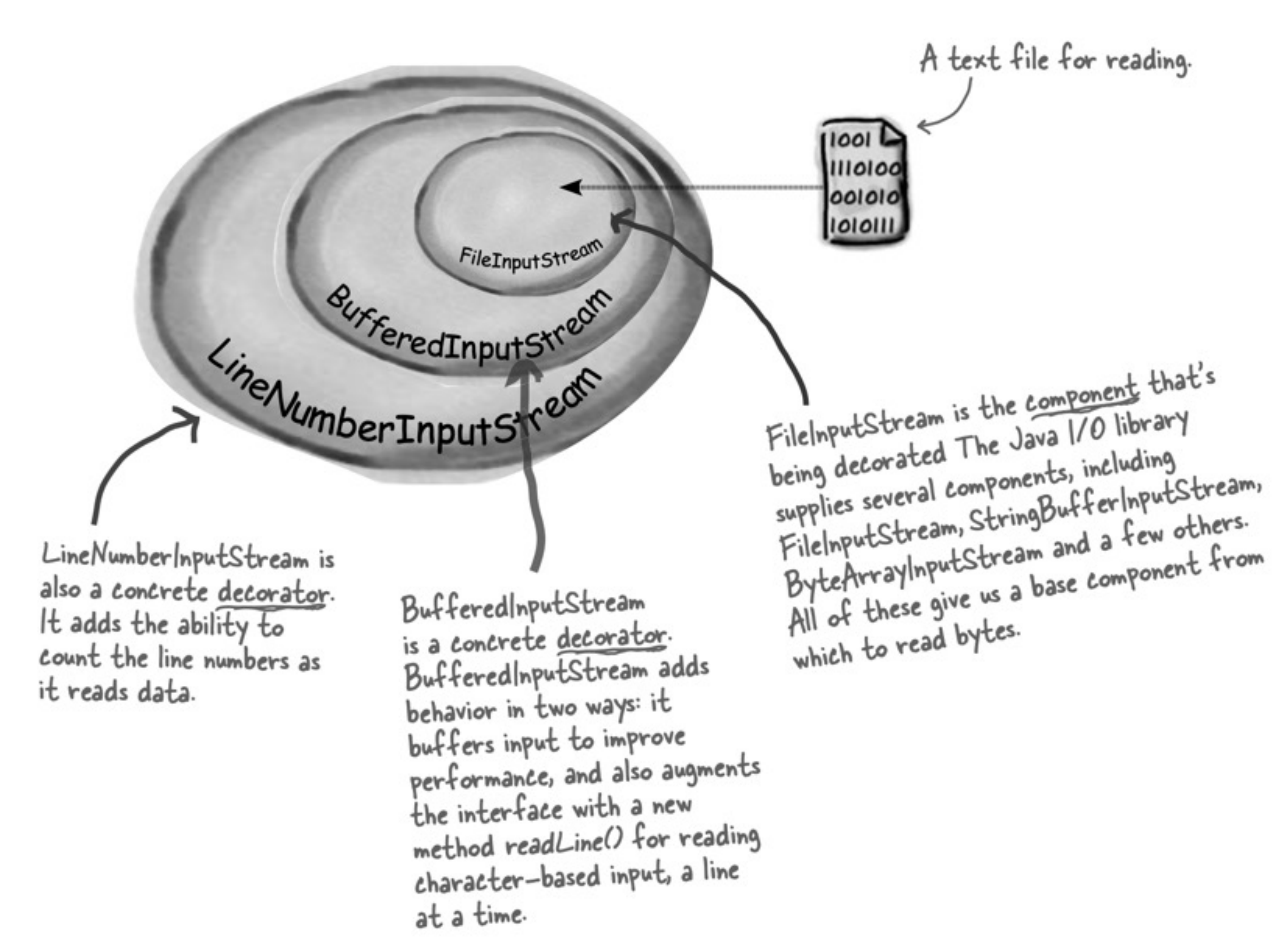


Beverage acts as our abstract component class.

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Decorator - Real World Example

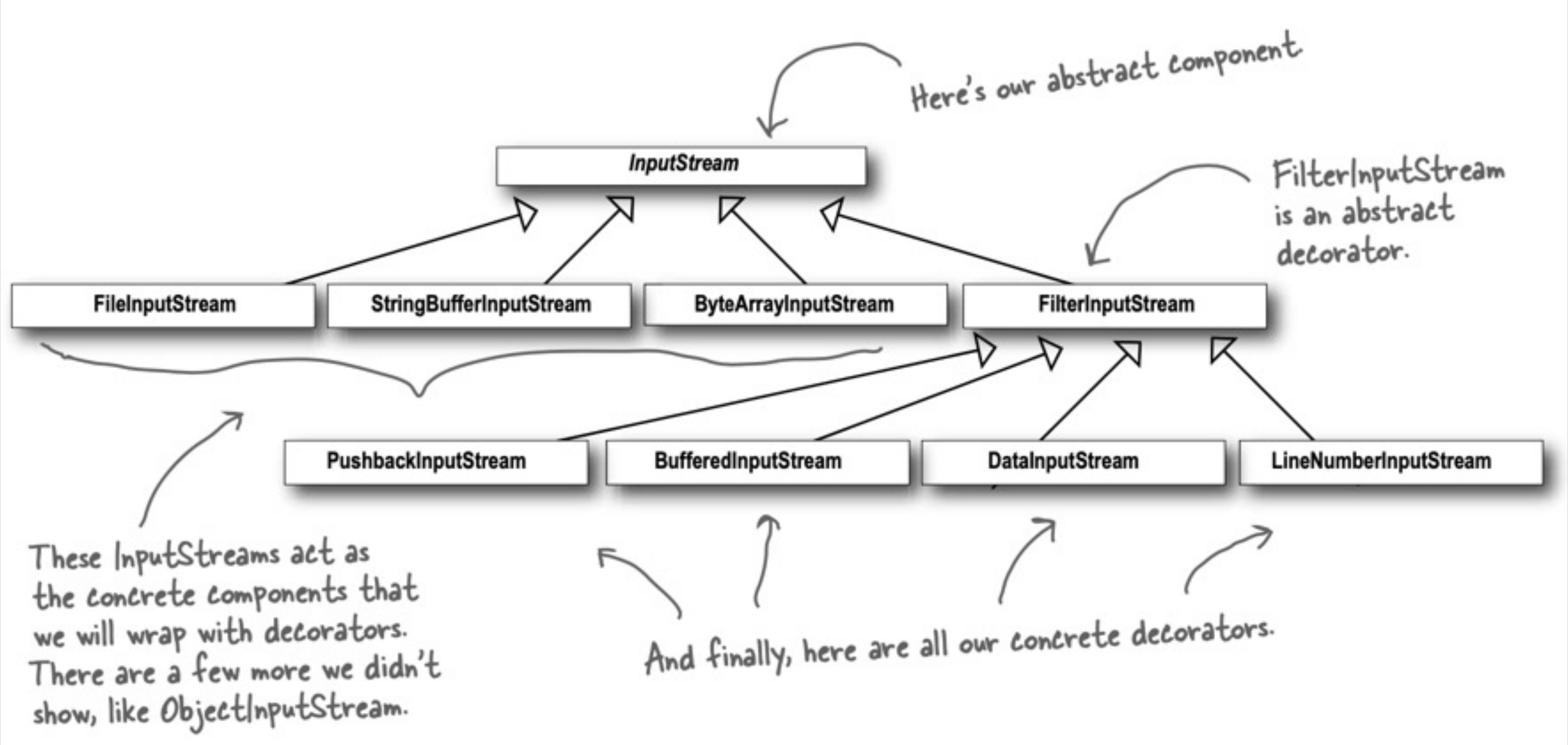


LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

FileInputStream is the component that's being decorated The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

Decorator - Real World Example


Here's our abstract component
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

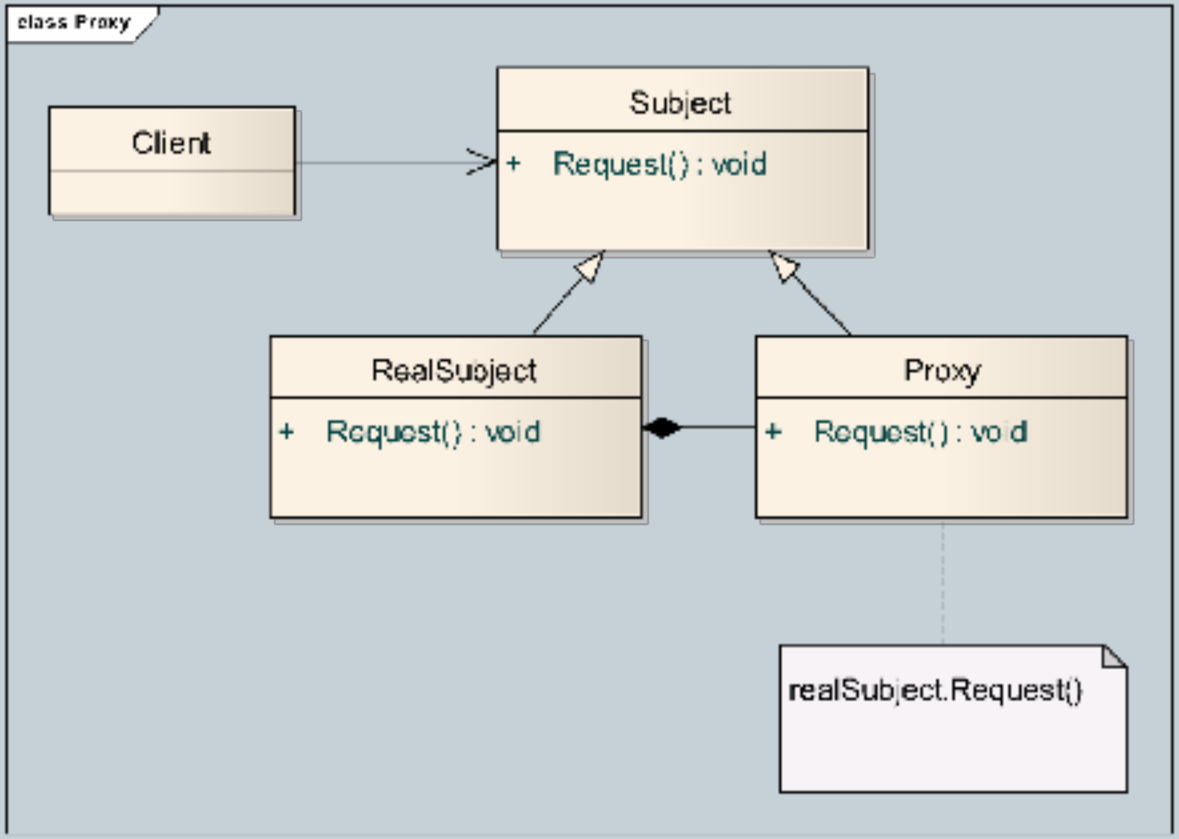
Decorator

- Pros
  - Extends class functionality at runtime
  - Helps in building flexible systems
  - Works great if coded against the abstract component type
- Cons
  - Results in problems if there is code that relies on the concrete component's type

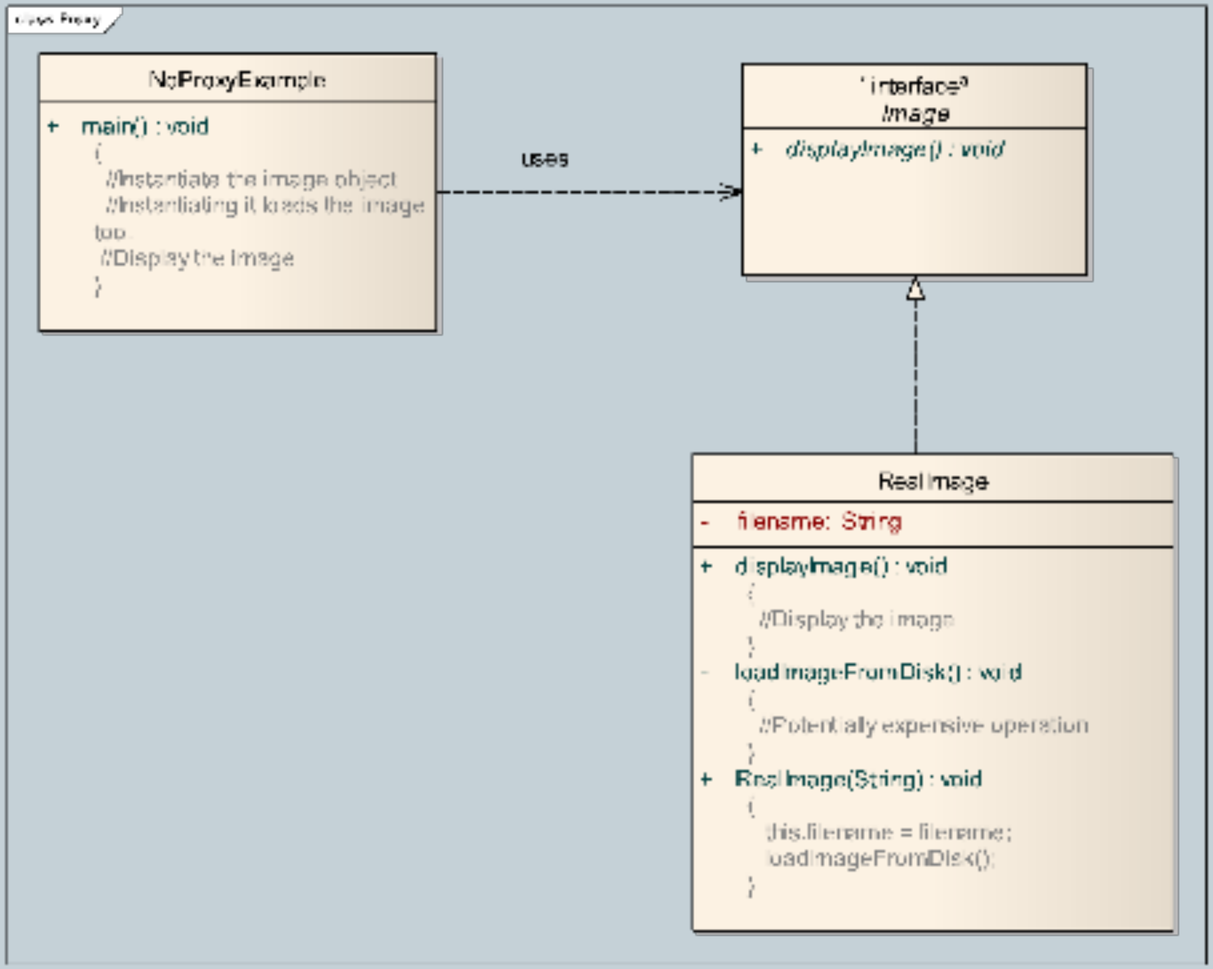
Proxy Definition

Provides a surrogate or placeholder for another object to control access to it

Proxy - Class diagram



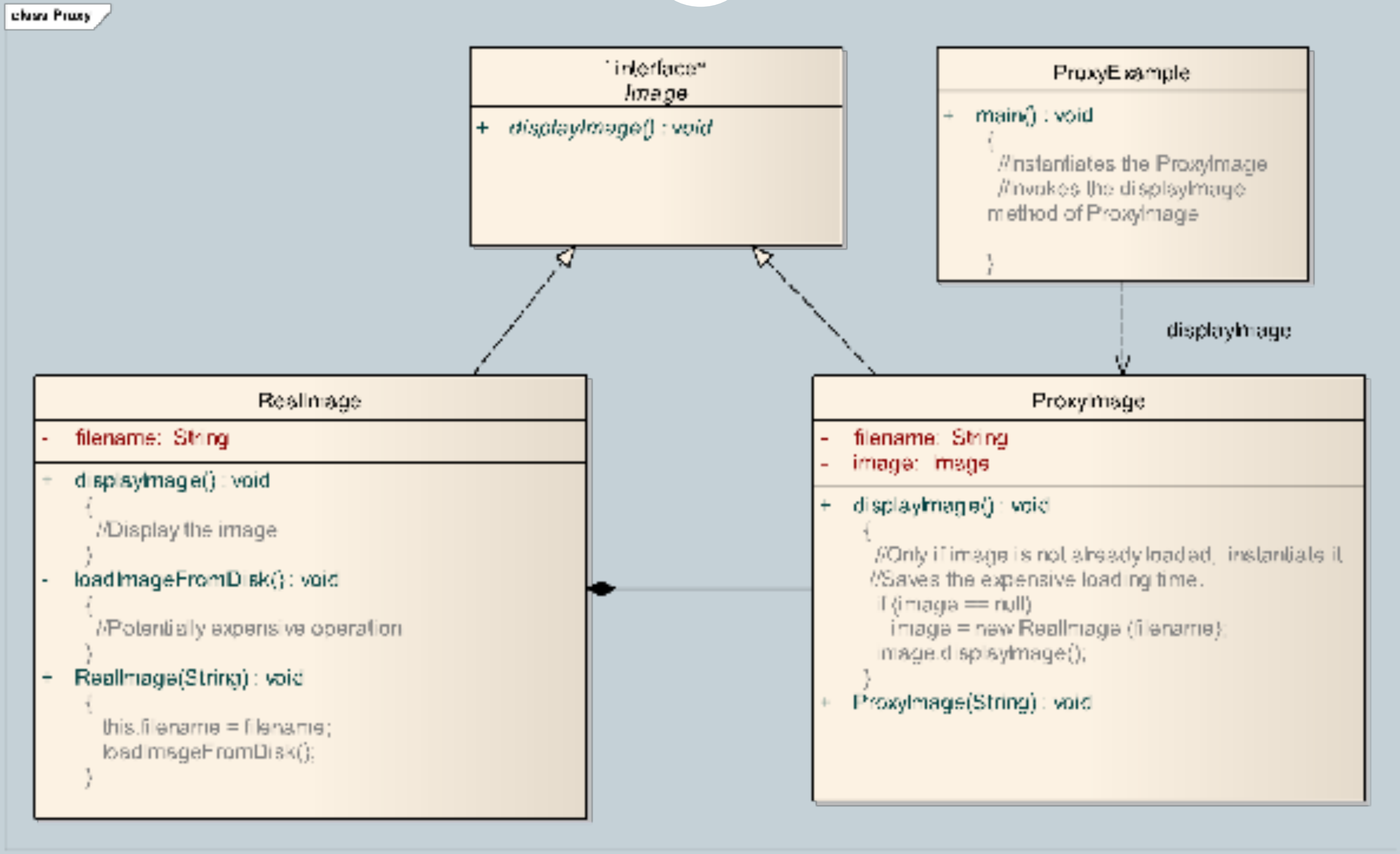
Proxy - Problem



```
+ main() : void
  {
    //Instantiate the image object
    //Instantiating it loads the image
  too
   //Display the image
  }
```

```
+ displayImage() : void
  {
    //Display the image
  }
- loadImageFromDisk() : void
  {
    //Potentially expensive operation
  }
+ RealImage(String) : void
  {
    this.filename = filename;
    loadImageFromDisk();
  }
```

Proxy - Solution



class Proxy

«interface»
Image

+ *displayImage() : void*

ProxyExample

```
+ main() : void
  {
   //Instantiates the ProxyImage
   //Invokes the displayImage
  method of ProxyImage

  }
```

displayImage

RealImage

- filename: String

```
+ displayImage() : void
  {
   //Display the image
  }
- loadImageFromDisk() : void
  {
   //Potentially expensive operation
  }
- RealImage(String) : void
  {
    this.filename = filename;
    loadImageFromDisk();
  }
```

ProxyImage

- filename: String
- image: Image

```
+ displayImage() : void
  {
   //Only if image is not already loaded, instantiate it.
  //Saves the expensive loading time.
   if (image == null)
     image = new RealImage (filename);
   image.displayImage();
  }
+ ProxyImage(String) : void
```

Proxy

- Pros
  - Prevents memory wastage
  - Creates expensive objects on demand

- Cons
  - Adds complexity when trying to ensure freshness

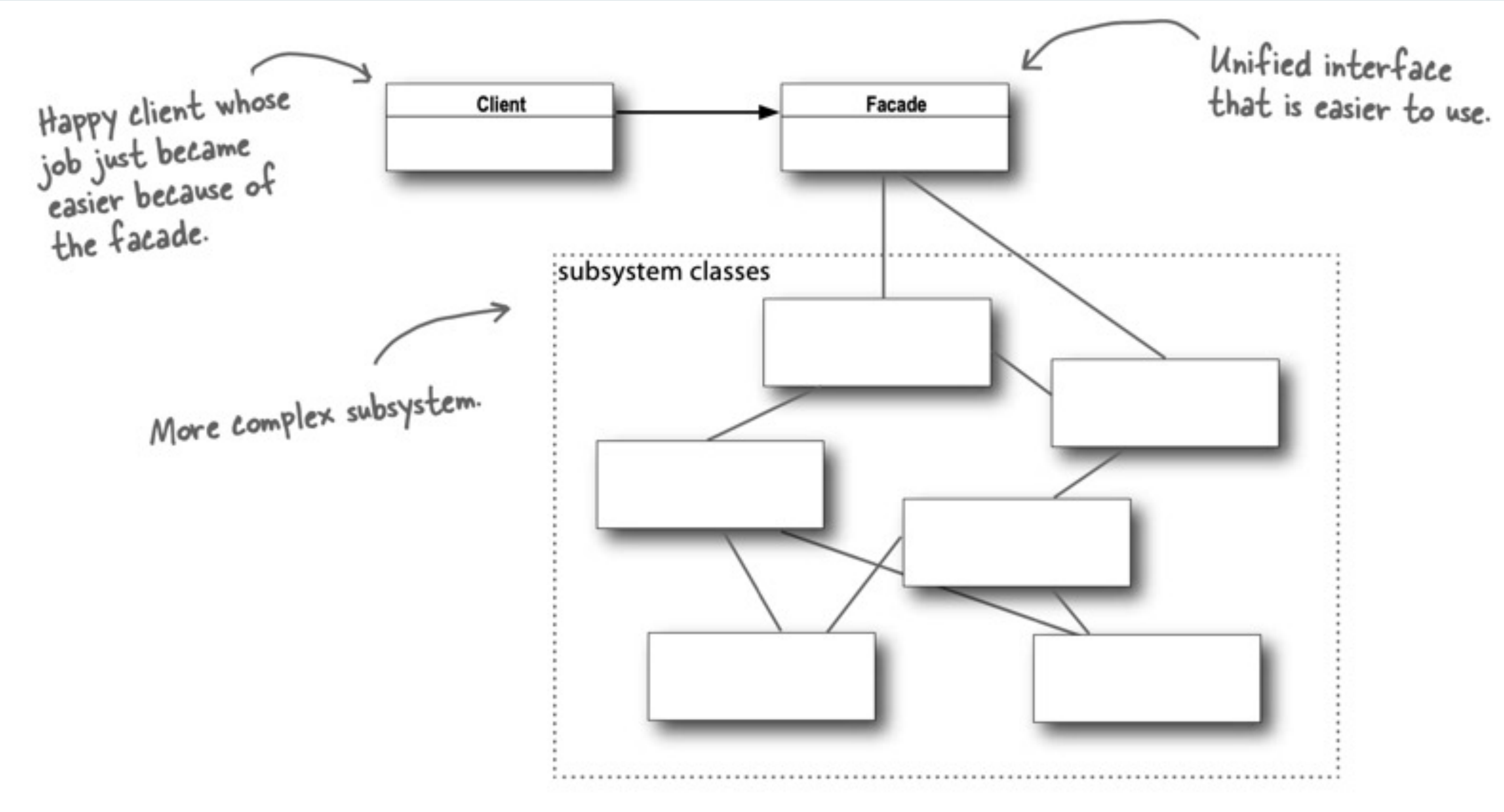
Facade Definition

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.

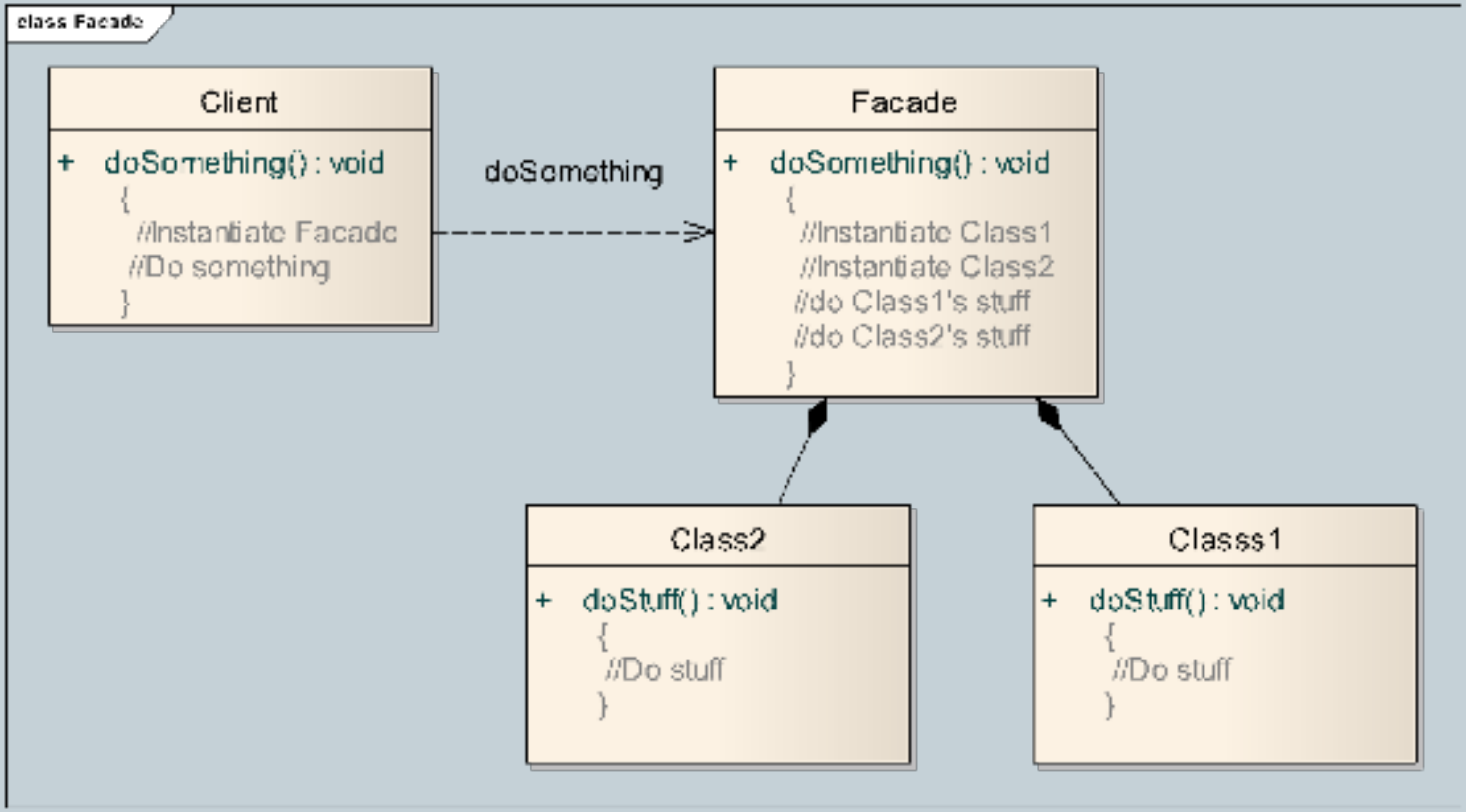
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

Façade - Class diagram


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

Façade - Class diagram



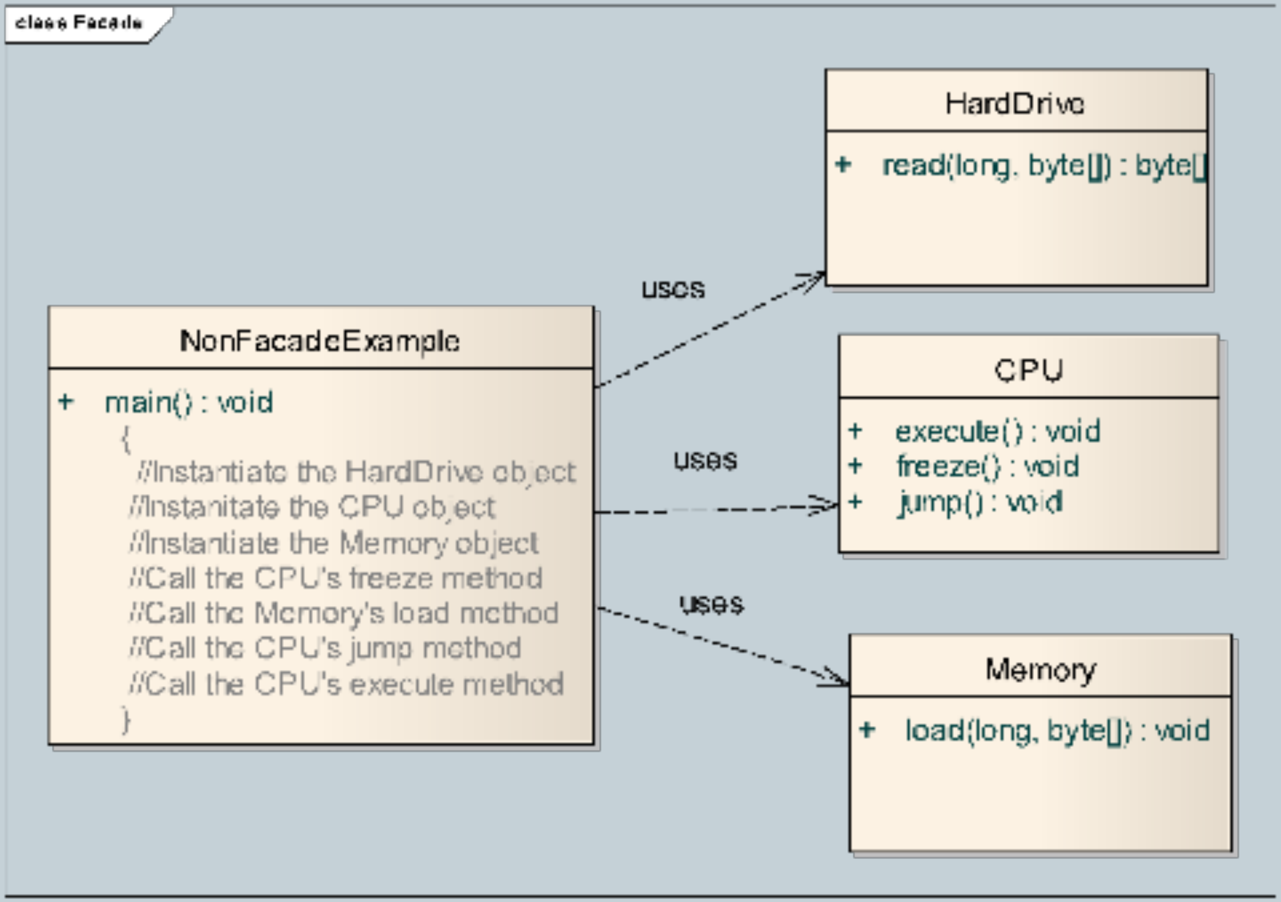
```
+   doSomething() : void
     {
       //Instantiate Facade
      //Do something
     }
```

```
+   doSomething() : void
     {
      //Instantiate Class1
      //Instantiate Class2
     //do Class1's stuff
     //do Class2's stuff
     }
```

```
+   doStuff() : void
     {
      //Do stuff
     }
```

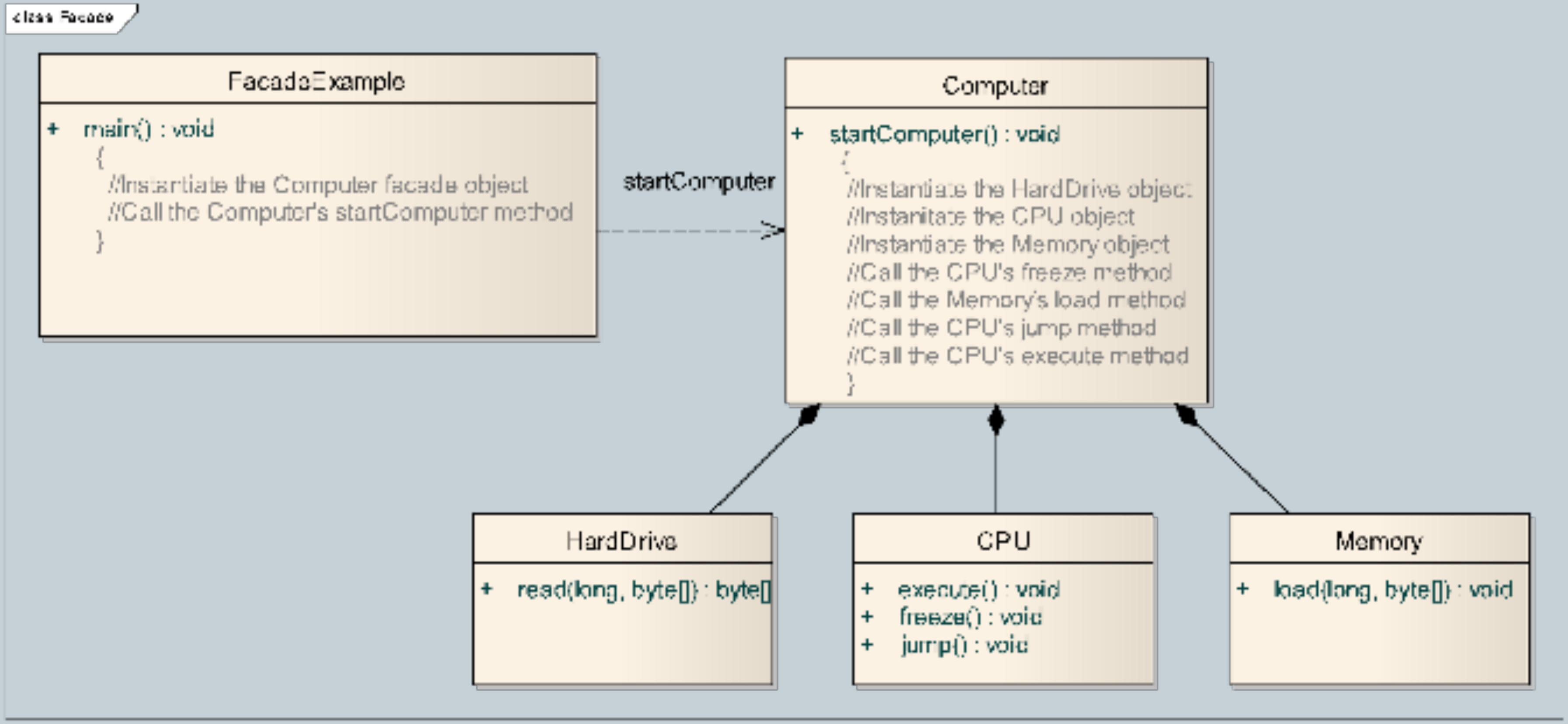
```
+   doStuff() : void
     {
      //Do stuff
     }
```

Façade - Problem



class Facade
HardDrive
+ read(long, byte[]) : byte[]
uses
NonFacadeExample
+ main() : void
{
//Instantiate the HardDrive object
//Instanitate the CPU object
//Instantiate the Memory object
//Call the CPU's freeze method
//Call the Memory's load method
//Call the CPU's jump method
//Call the CPU's execute method
}
uses
CPU
+ execute() : void
+ freeze() : void
+ jump() : void
uses
Memory
+ load(long, byte[]) : void

Façade - Solution



class Facade

FacadeExample

```
+ main() : void
  {
   //Instantiate the Computer facade object
   //Call the Computer's startComputer method
  }
```

startComputer

Computer

```
+ startComputer() : void
  {
   //Instantiate the HardDrive object
   //Instanitate the CPU object
   //Instantiate the Memory object
   //Call the CPU's freeze method
   //Call the Memory's load method
   //Call the CPU's jump method
   //Call the CPU's execute method
  }
```

HardDrive

```
+ read(long, byte[]) : byte[]
```

CPU

```
+ execute() : void
+ freeze() : void
+ jump() : void
```

Memory

```
+ load(long, byte[]) : void
```

Facade

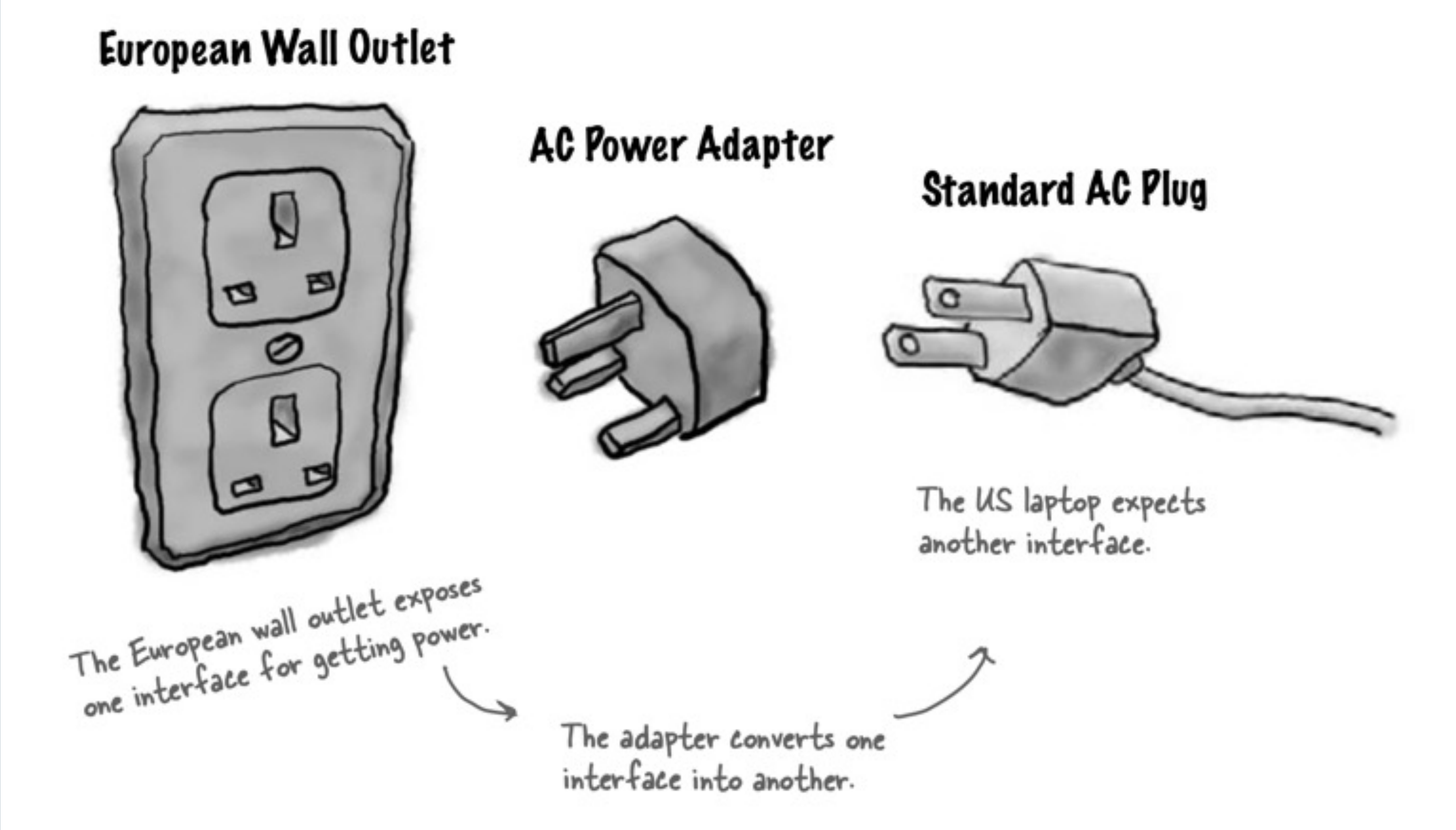
- Pros
  - Makes code easier to use and understand
  - Reduces dependencies on classes
  - Decouples a client from a complex system

- Cons
  - Results in more rework for improperly designed Façade class
  - Increases complexity and decreases runtime performance for large number of Façade classes

Adapter Definition

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Adapter Example



European Wall Outlet

AC Power Adapter

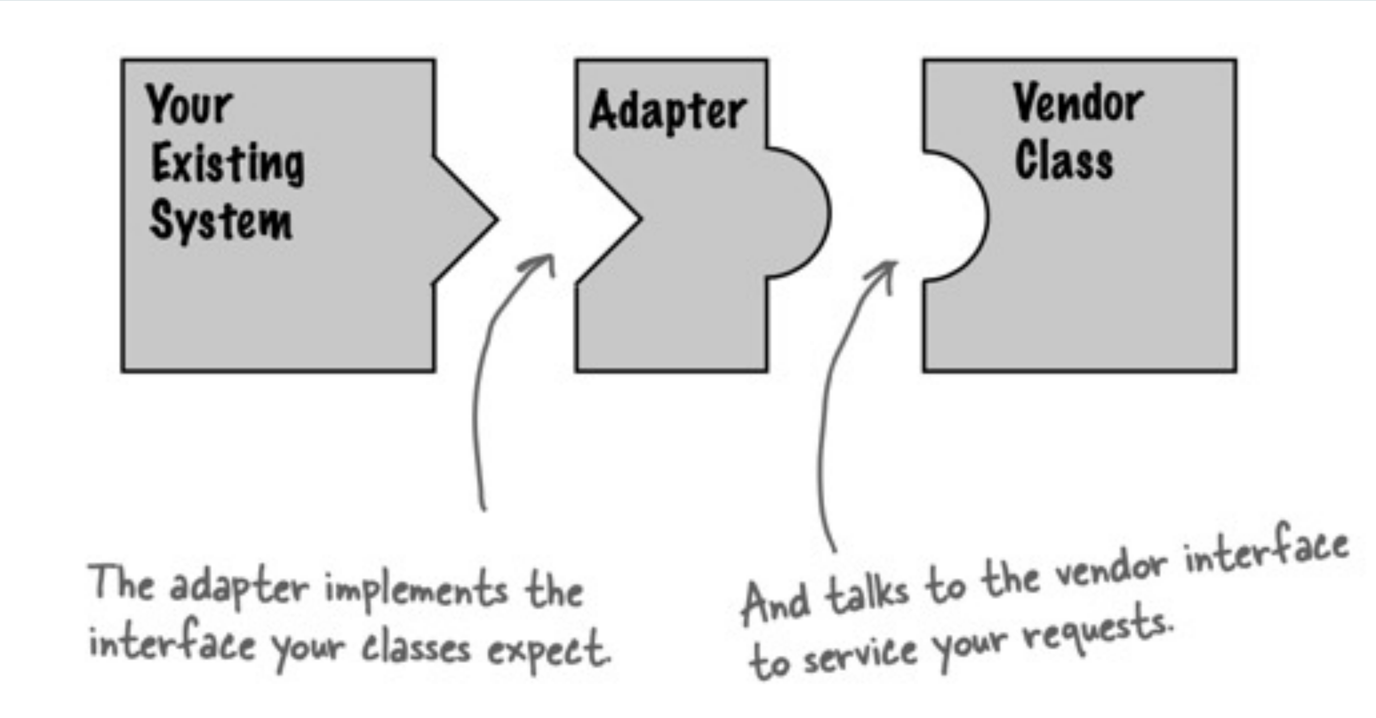
Standard AC Plug

The European wall outlet exposes one interface for getting power.

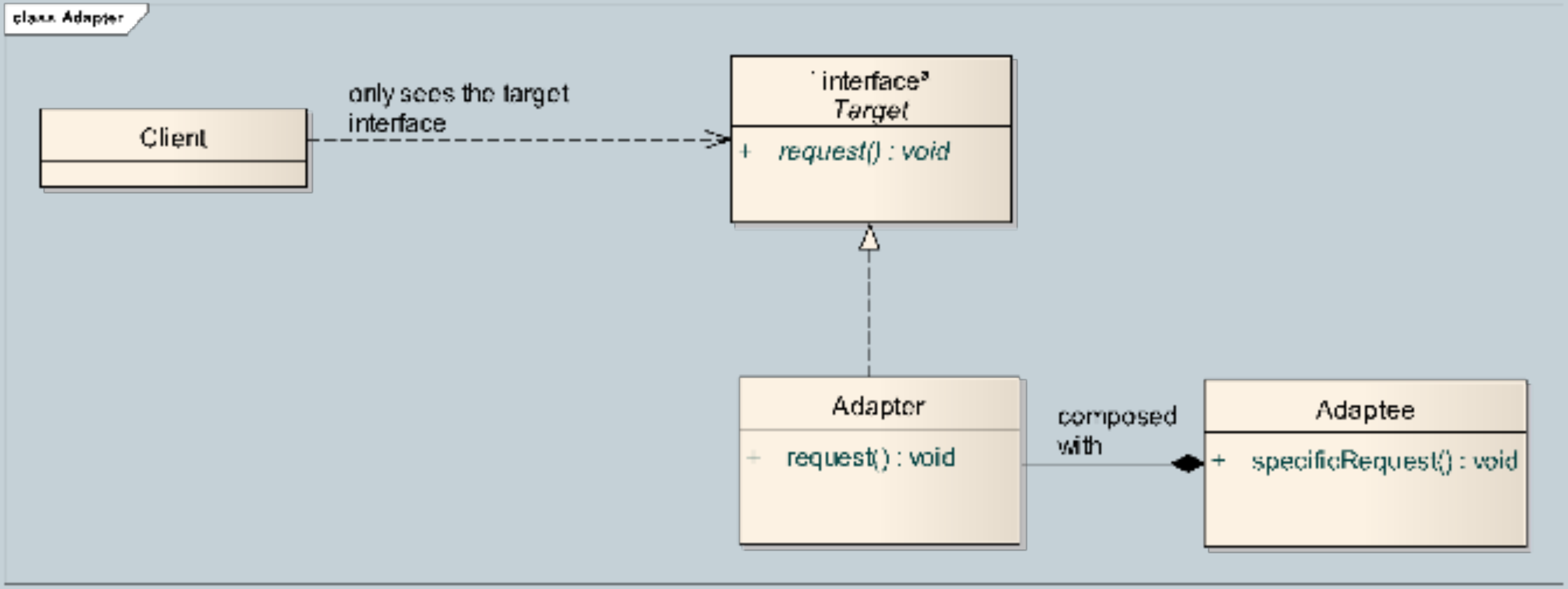
The adapter converts one interface into another.

The US laptop expects another interface.

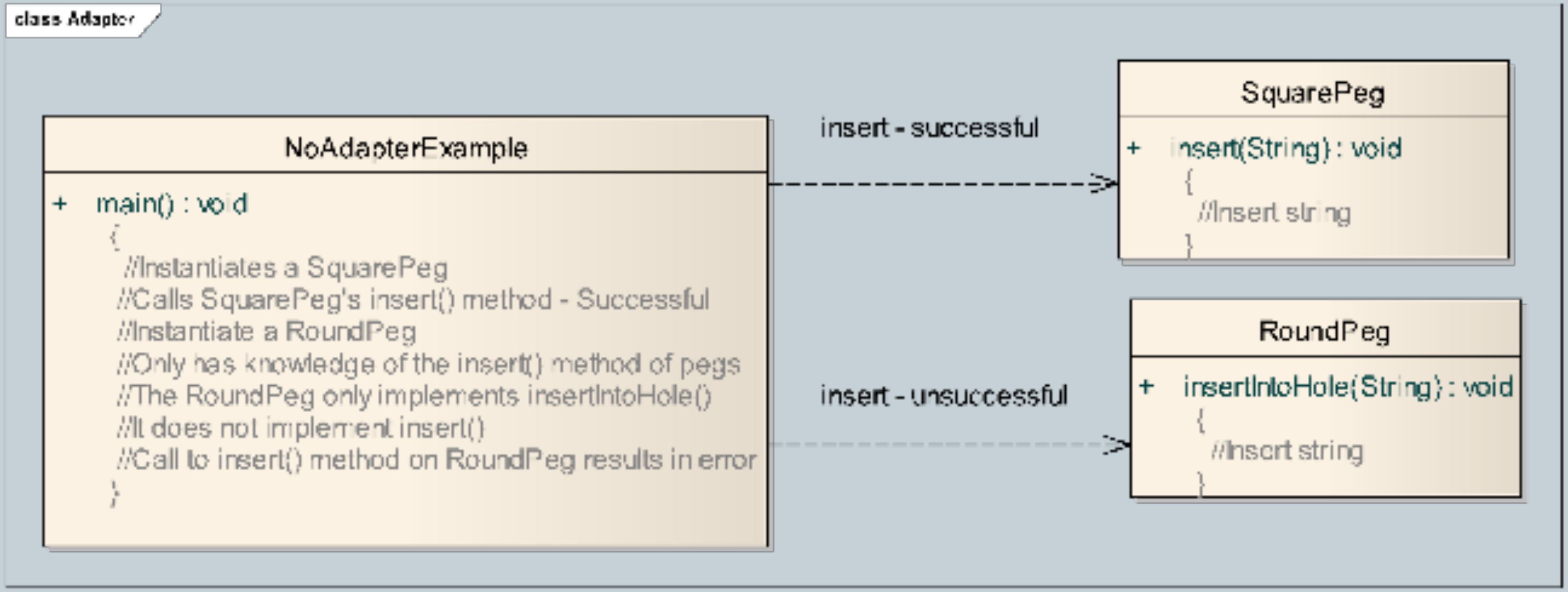
Software Adapter


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.

Adapter - Class diagram



Adapter - Problem



class Adapter

NoAdapterExample

```
+   main() : void
    {
     //Instantiates a SquarePeg
     //Calls SquarePeg's insert() method - Successful
     //Instantiate a RoundPeg
     //Only has knowledge of the insert() method of pegs
     //The RoundPeg only implements insertIntoHole()
     //It does not implement insert()
     //Call to insert() method on RoundPeg results in error
    }
```

insert - successful

SquarePeg

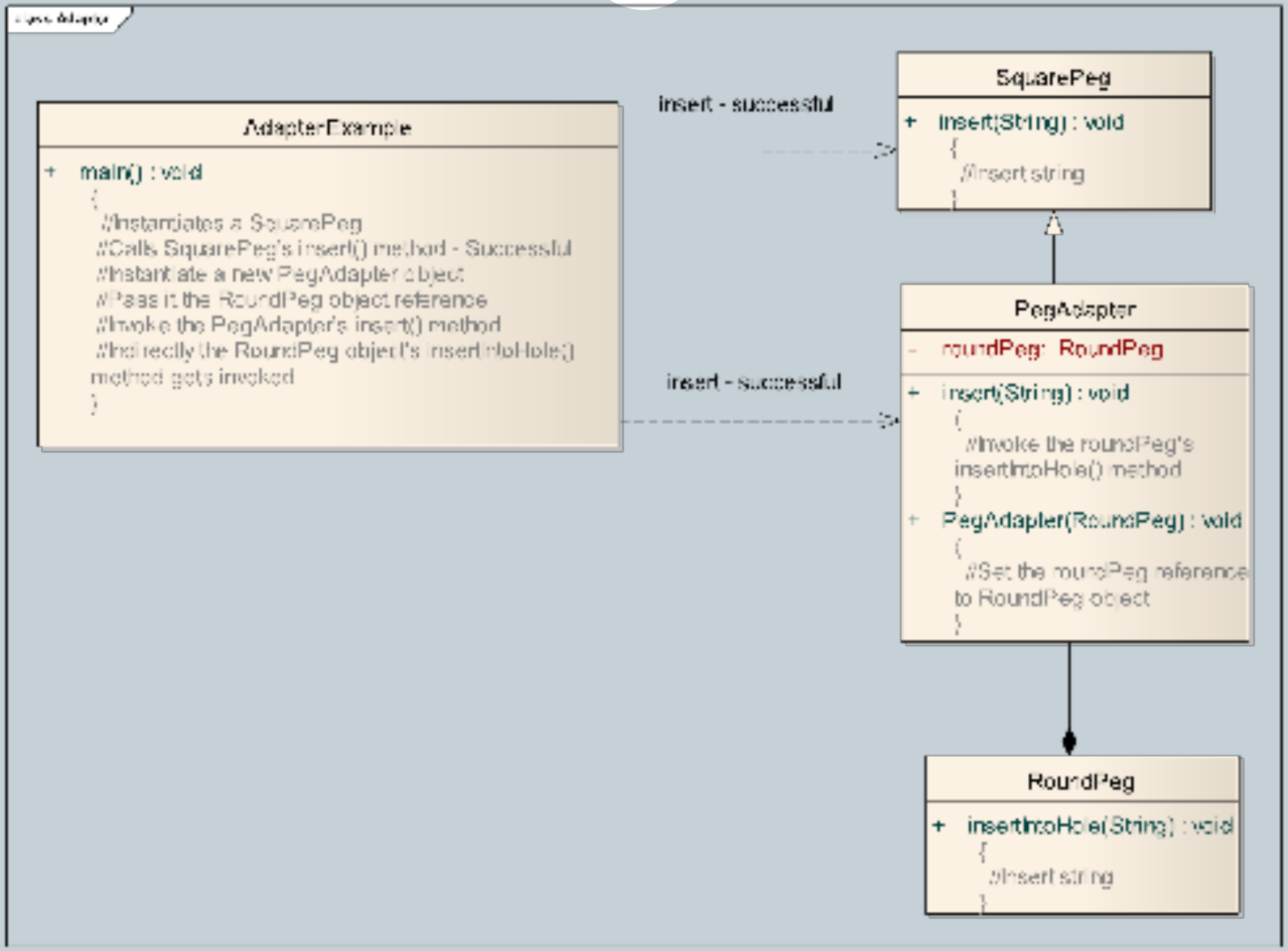
```
+   insert(String) : void
    {
     //Insert string
    }
```

insert - unsuccessful

RoundPeg

```
+   insertIntoHole(String) : void
    {
     //Insert string
    }
```

Adapter - Solution



```
AdapterExample
+   main() : void
     {
      //Instantiates a SquarePeg
     //Calls SquarePeg's insert() method - Successful
     //Instantiate a new PegAdapter object
     //Pass it the RoundPeg object reference
     //Invoke the PegAdapter's insert() method
     //Indirectly the RoundPeg object's insertIntoHole()
    method gets invoked
     }
```

```
SquarePeg
+   insert(String) : void
     {
      //Insert string
     }
```

```
PegAdapter
-   roundPeg:  RoundPeg
+   insert(String) : void
     {
      //Invoke the roundPeg's
     insertIntoHole() method
     }
+   PegAdapter(RoundPeg) : void
     {
      //Set the roundPeg reference
     to RoundPeg object
     }
```

```
RoundPeg
+   insertIntoHole(String) : void
     {
      //Insert string
     }
```

Adapter

- Pros
  - Increases code reuse
  - Encapsulates the interface change
  - Handles legacy code
- Cons
  - Increases complexity for large number of changes

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

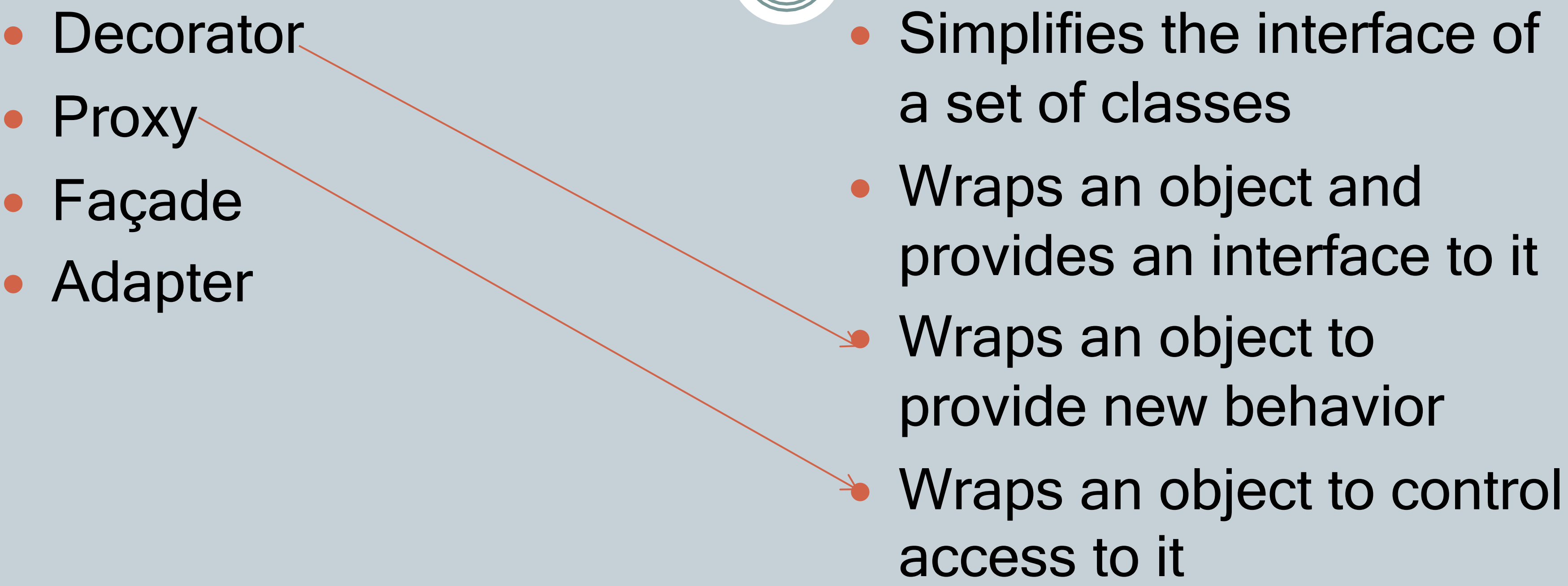
- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

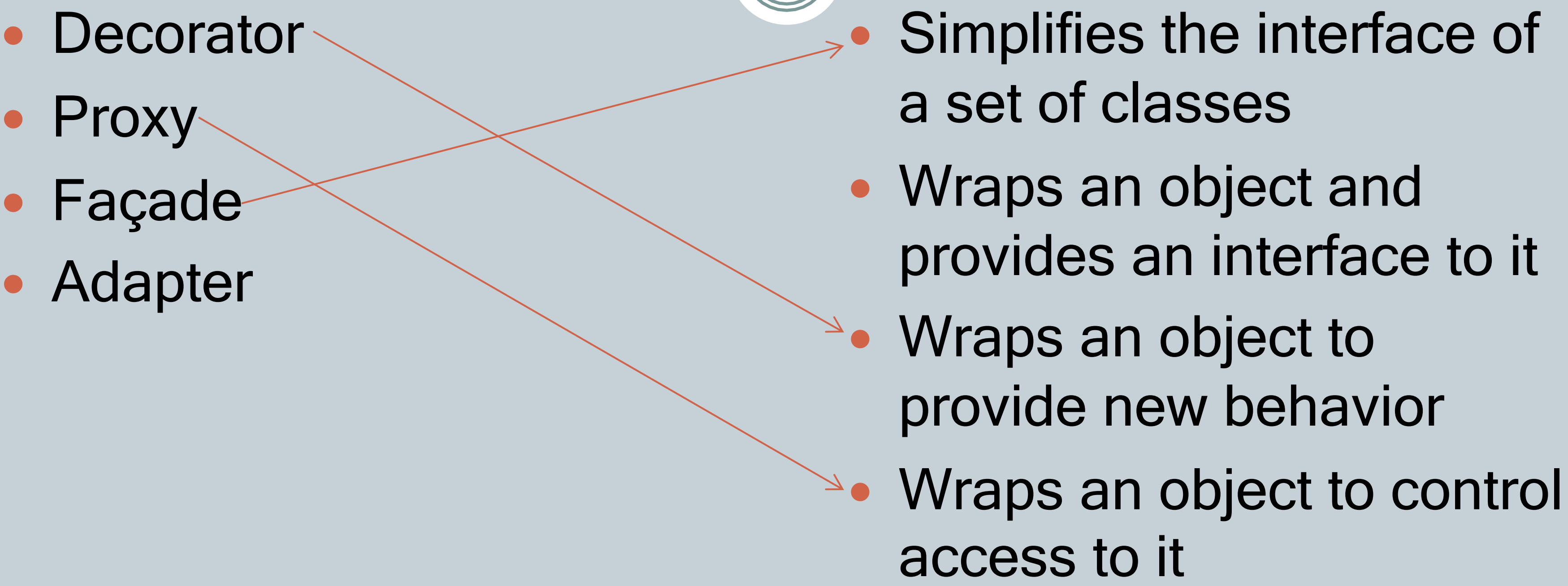
Patterns & Definitions - Group 2



- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

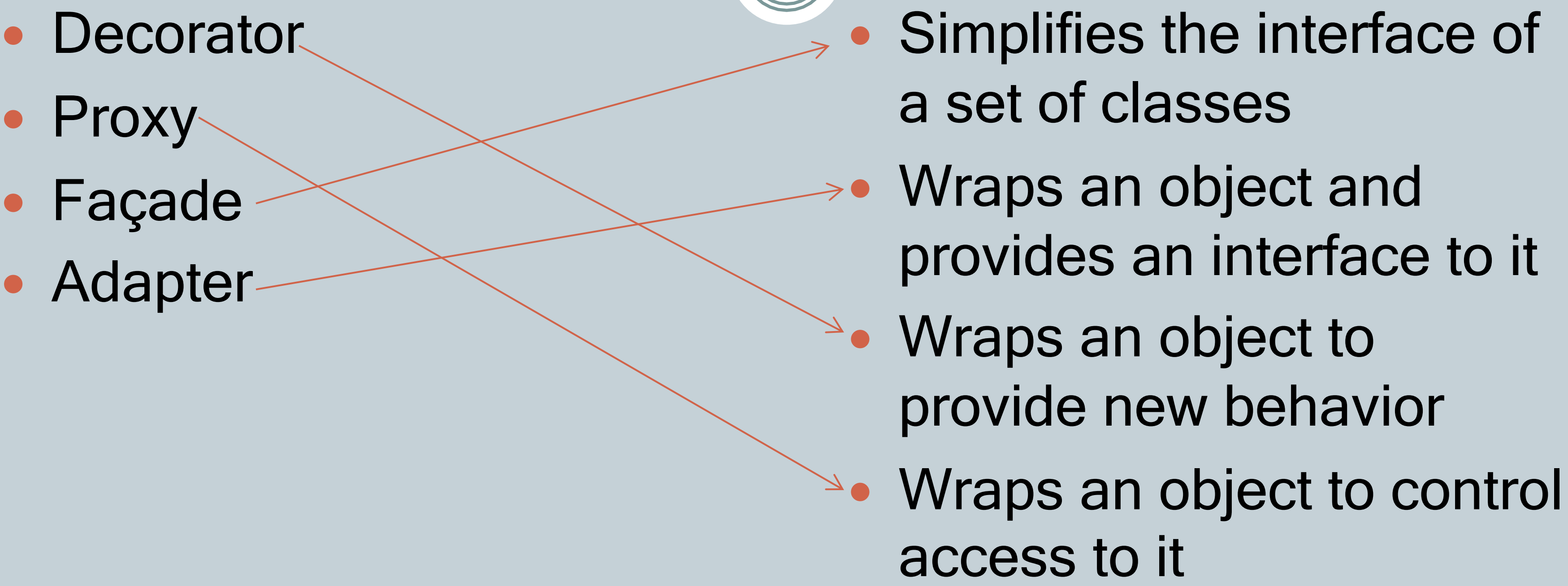
Patterns & Definitions - Group 2



- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2



- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral

Pattern Classification

Strategy	Behavioral
Observer	Behavioral
Singleton	Creational
Decorator	
Proxy	
Façade	
Adapter	

Pattern Classification

Strategy	Behavioral
Observer	Behavioral
Singleton	Creational
Decorator	Structural
Proxy	
Façade	
Adapter	

Pattern Classification

• Strategy	• Behavioral
• Observer	• Behavioral
• Singleton	• Creational
• Decorator	• Structural
• Proxy	• Structural
• Façade	
• Adapter	

Pattern Classification

Strategy	Behavioral
Observer	Behavioral
Singleton	Creational
Decorator	Structural
Proxy	Structural
Façade	Structural
Adapter	

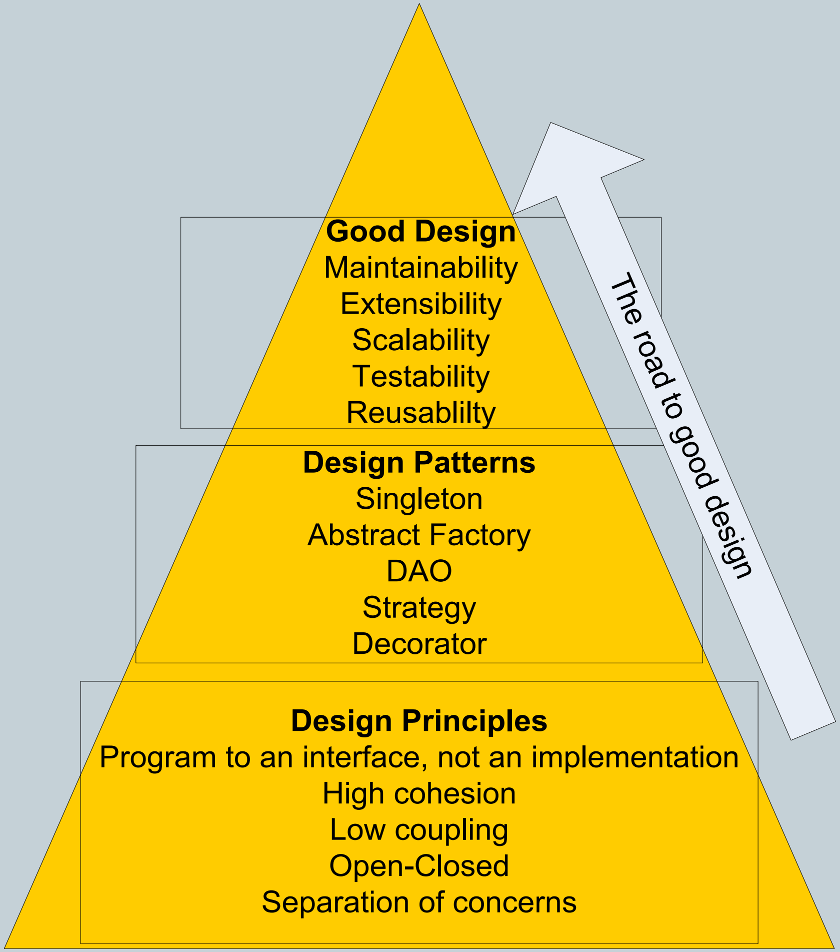
Pattern Classification

Strategy	Behavioral
Observer	Behavioral
Singleton	Creational
Decorator	Structural
Proxy	Structural
Façade	Structural
Adapter	Structural

Conclusion - Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

Conclusion



Good Design
Maintainability
Extensibility
Scalability
Testability
Reusablilty

Design Patterns
Singleton
Abstract Factory
DAO
Strategy
Decorator

Design Principles
Program to an interface, not an implementation
High cohesion
Low coupling
Open-Closed
Separation of concerns

The road to good design

Model View Controller (MVC)

BIGGER THAN A PATTERN: IT'S AN ARCHITECTURE

MVC

- The intent of MVC is to keep neatly separate objects into one of tree categories
  - Model
    - The data, the business logic, rules, strategies, and so on
  - View
    - Displays the model and usually has components that allows user to edit change the model
  - Controller
    - Allows data to flow between the view and the model
    - The controller mediates between the view and model

Model

- The Model's responsibilities
  - Provide access to the state of the system
  - Provide access to the system's functionality
  - Can notify the view(s) that its state has changed

View

- The view's responsibilities
  - Display the state of the model to the user
- At some point, the model (a.k.a. the observable) must registers the views (a.k.a. observers) so the model can notify the observers that its state has changed

Controller

- The controller's responsibilities
  - Accept user input
    - Button clicks, key presses, mouse movements, slider bar changes
  - Send messages to the model, which may in turn notify it observers
  - Send appropriate messages to the view
- In Java, listeners are controllers

MVC Misunderstood

- MVC is understood by different people in different ways
- It is often misunderstood, but most software developers will say it is important; powerful
- Let’s start it right, a little history, first Smalltalk

Smalltalk-80™

- In the MVC paradigm, the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of objects, each specialized for its task.
  - The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application.
  - The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
  - The **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

Smalltalk-80™ *continued*

- The formal separation of these three tasks is an important notion that is particularly suited to MVC Smalltalk-80 where the basic behavior can be embodied in abstract objects: *View*, *Controller, Model,* and *Object*
- MVC was discovered by Trygve Reenskaug in 1979

Sun says

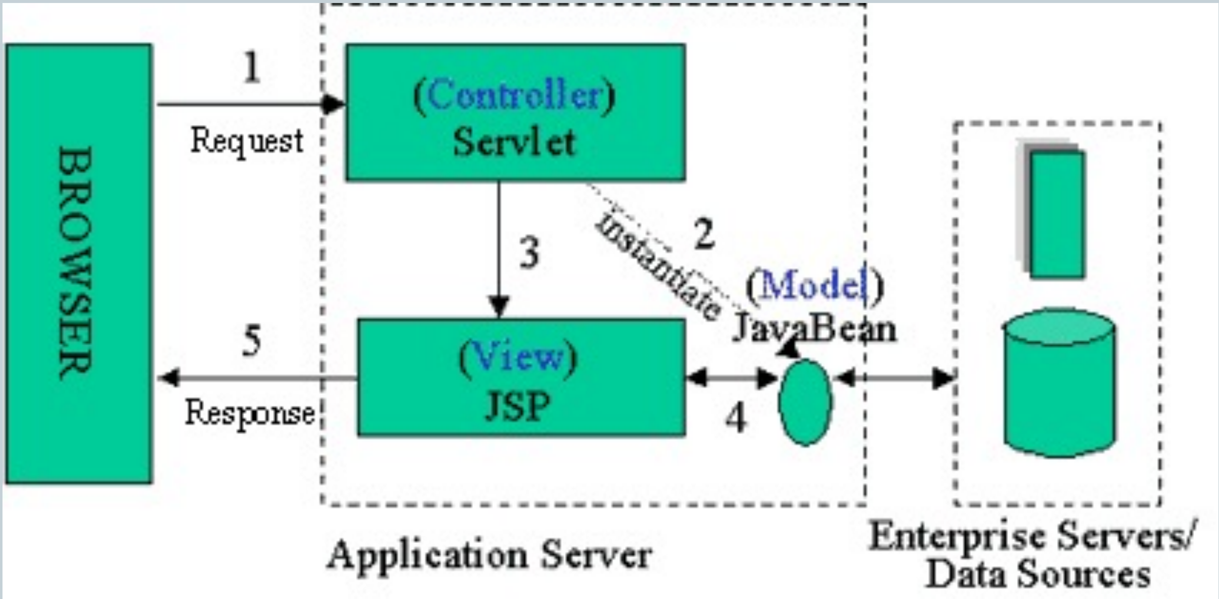
- Model-View-Controller ("MVC") is the recommended architectural design pattern for interactive applications
- MVC organizes an interactive application into three separate modules:
  - one for the application model with its data representation and business logic,
  - the second for views that provide data presentation and user input, and
  - the third for a controller to dispatch requests and control flow.

Sun continued

- Most Web application frameworks use some variation of the MVC design pattern
- The MVC (architectual) design pattern provides a host of design benefits

Java Server Pages

- Model: Enterprise Beans with data in the DBMS
  - JavaBean: a class that encapsulates objects and can be displayed graphically
- Controller: Servlets create beans, decide which JSP to return
  - Servlet object do the bulk of the processing
- View: The JSPs generated in the presentation layer (the browser)



OO-tips writes

- The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller.
- MVC was originally developed to map the traditional input, processing, output roles into GUIs:
  - Console Input → Processing → Output
  - GUI Input → Controller → Model → View

Wikipedia writes

- **Model-View-Controller (MVC)** is a software architecture that separates an application's data model, user interface, and control logic into three distinct components so that modifications to one component can be made with minimal impact to the others.
- MVC is often thought of as a software design pattern. However, MVC encompasses more of the architecture of an application than is typical for a design pattern. Hence the term architectural pattern may be useful (Buschmann, et al 1996), or perhaps an aggregate design pattern.

MVC Benefits

- Clarity of design
  - easier to implement and maintain
- Modularity
  - changes to one doesn't affect other modules
  - can develop in parallel once you have the interfaces
- Multiple views
  - spreadsheets, powerpoint, file browsers, games,….

Summary (MVC)

- The intent of MVC is to keep neatly separate objects into one of three categories
  - Model
    - The data, the business logic, rules, strategies, and so on
  - View
    - Displays the model and often has components to allow users to change the state of the model
  - Controller
    - Allows data to flow between the view and the model
    - The controller mediates between the view and model

Model

- The Model's responsibilities
  - Provide access to the state of the model
    - getters, toString, other methods that provide info
  - Provide access to the system's functionality
    - changeRoom(int), shootArrow(int)
  - Notify the view(s) that its state has changed

```
// If extending Java's Obervable class, do NOT forget
// to tell yourself your state has changed
super.setChanged();
// Otherwise, the next notifyObservers message will not
// send update messages to the registered Observers
this.notifyObservers();
```

View

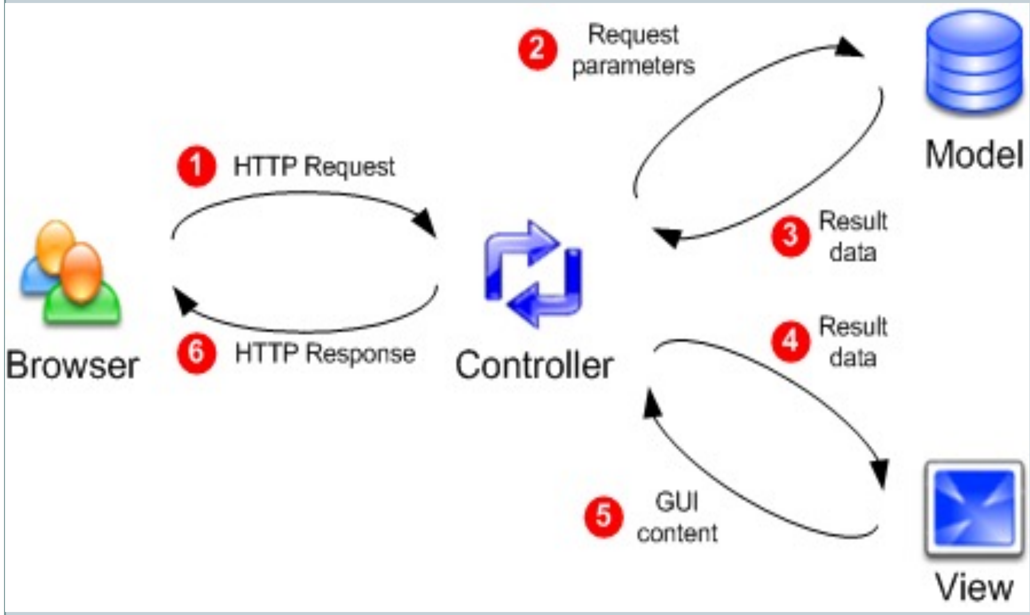
- The view's responsibilities
  - Display the state of the model to users, accept input
- The model (a.k.a. the Observable) must register the views (a.k.a. Observers) so the model can notify the observers that its state has changed
  - Java’s Observer/Observable support provides

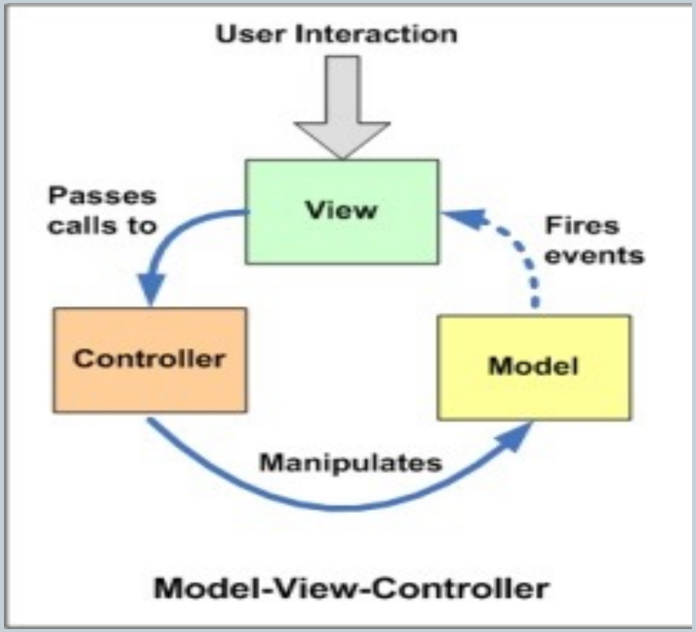
```
public void addObserver(Observer o)
// Adds an observer to the set of observers for this
// object, provided that it is not the same as some
// observer already in the set.
```

Controller

- The controller's responsibilities
  - Respond to user input (events)
    - Button click, key press, mouse click, slider bar change
  - Send messages to the model, which may in turn notify it observers
  - Send appropriate messages to the view
- In Java, controllers are implemented as listeners
  - An ActionListener object and its actionPerformed method is a Controller

Two Views of MVC

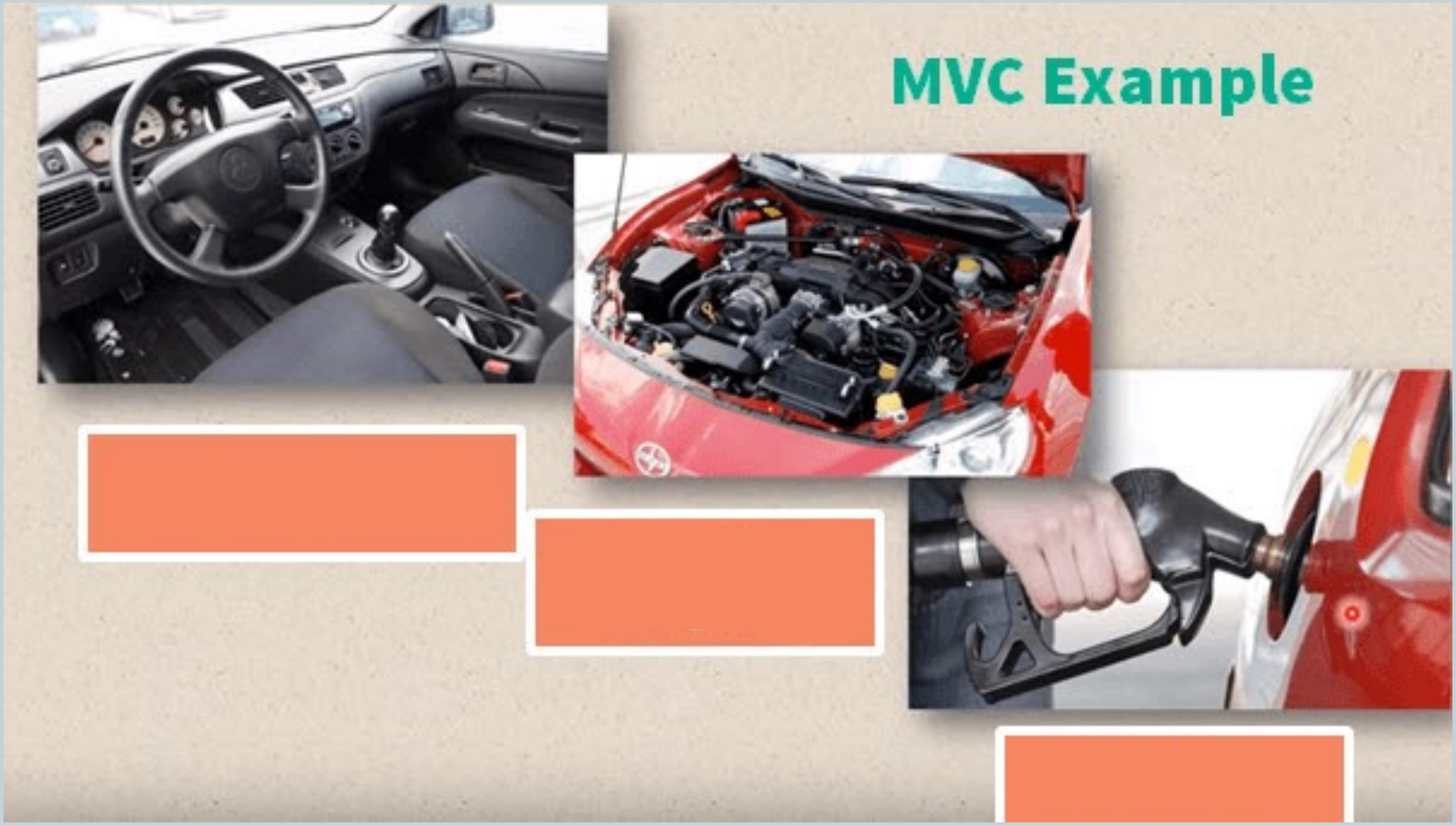




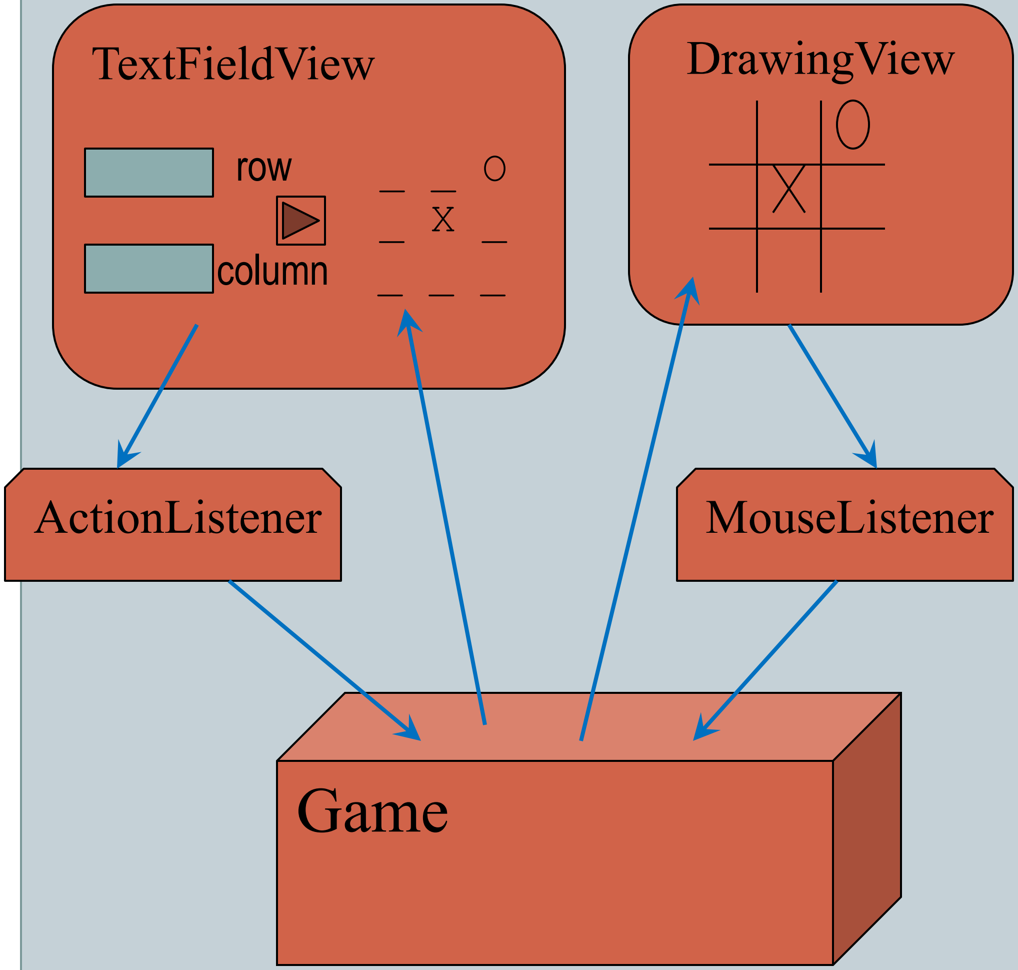
MVC Example

MVC Example

MVC Example



MVC in an Old TicTacToe Project

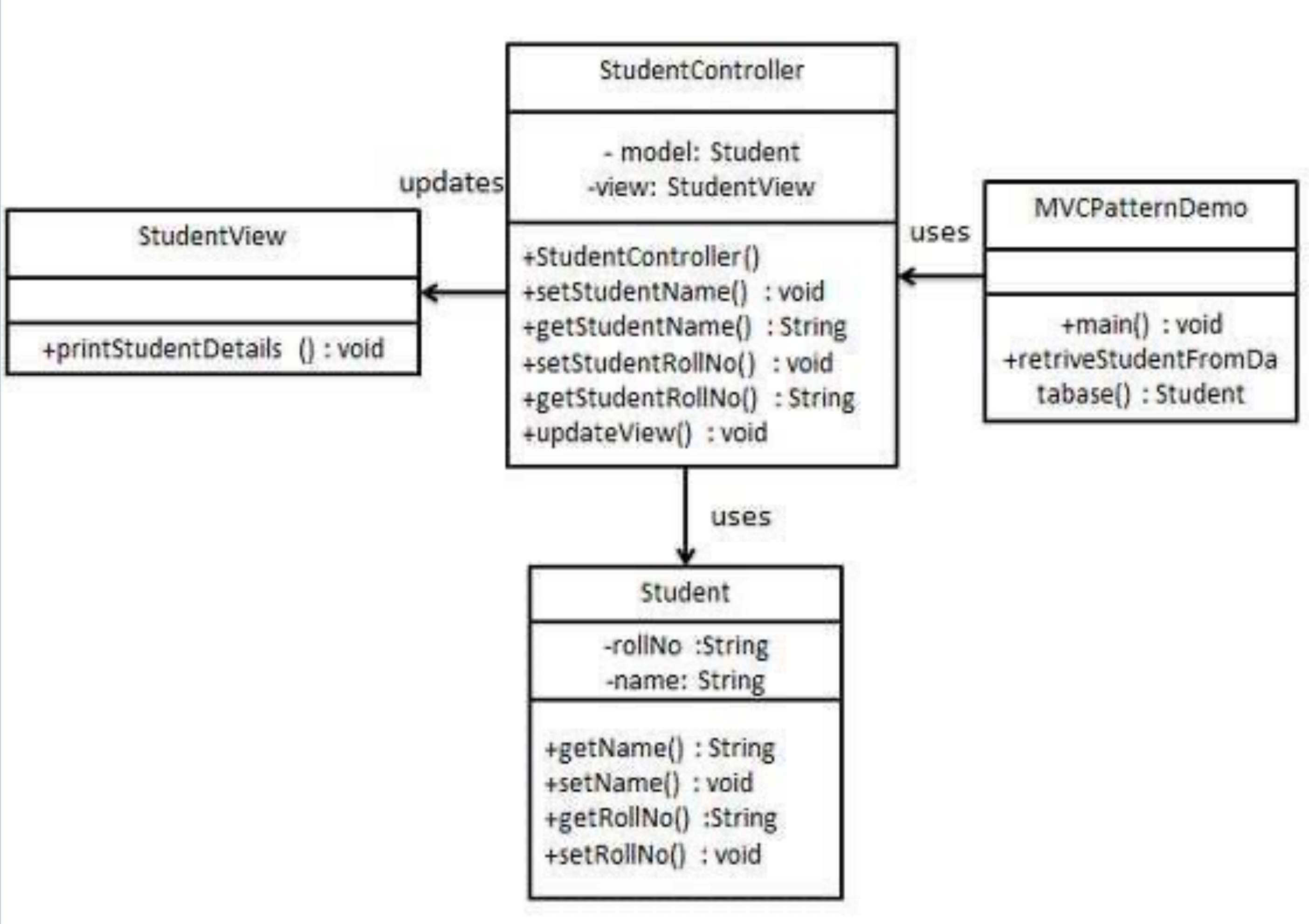


1. Name the controllers
2. Name the model
3. Do the controllers have a user interface?
4. Does a view allow user input?
5. Can a view send messages to the model?
6. Can a view send messages to the controller?
7. Can a system have more than one view?
8. Can a system have more than one controller?

Popular MVC Web Framework

- Django (Python)
- Ruby on Rails
- CakePHP
- Lavarel
- Symphony
- …

MVC - Example


StudentController
- model: Student
-view: StudentView
+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void
updates
StudentView
+printStudentDetails () : void
uses
MVCPatternDemo
+main() : void
+retriveStudentFromDatabase() : Student
uses
Student
-rollNo :String
-name: String
+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void