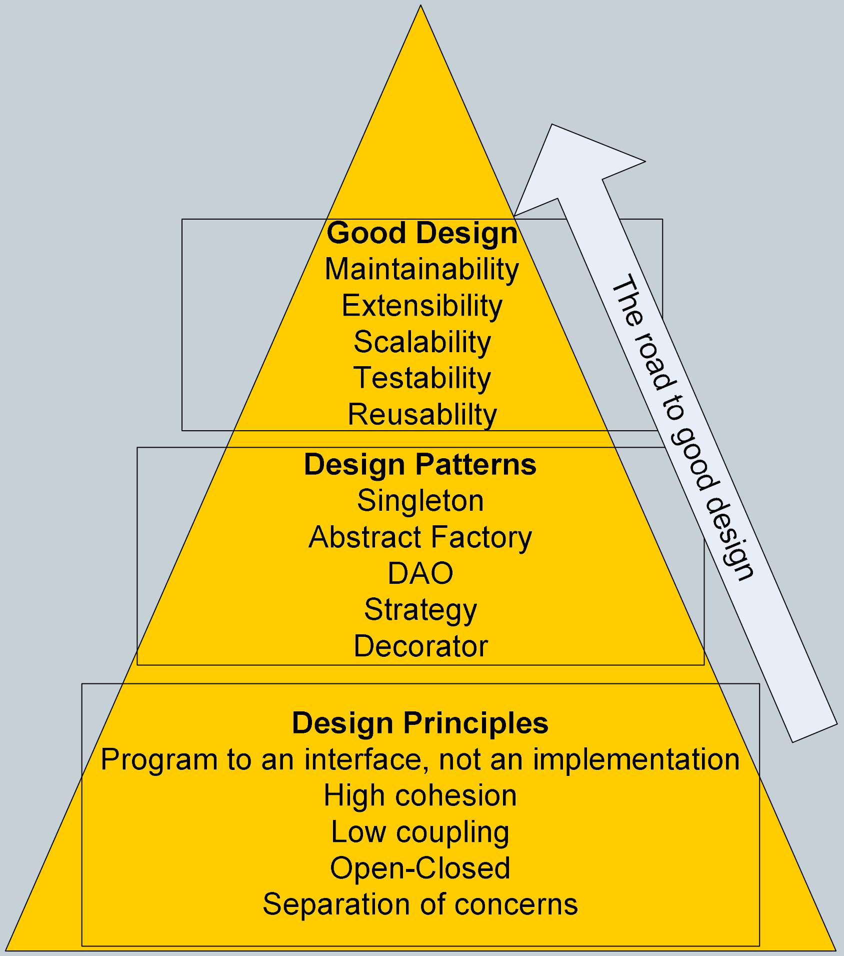
What are Design Patterns?

- 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)

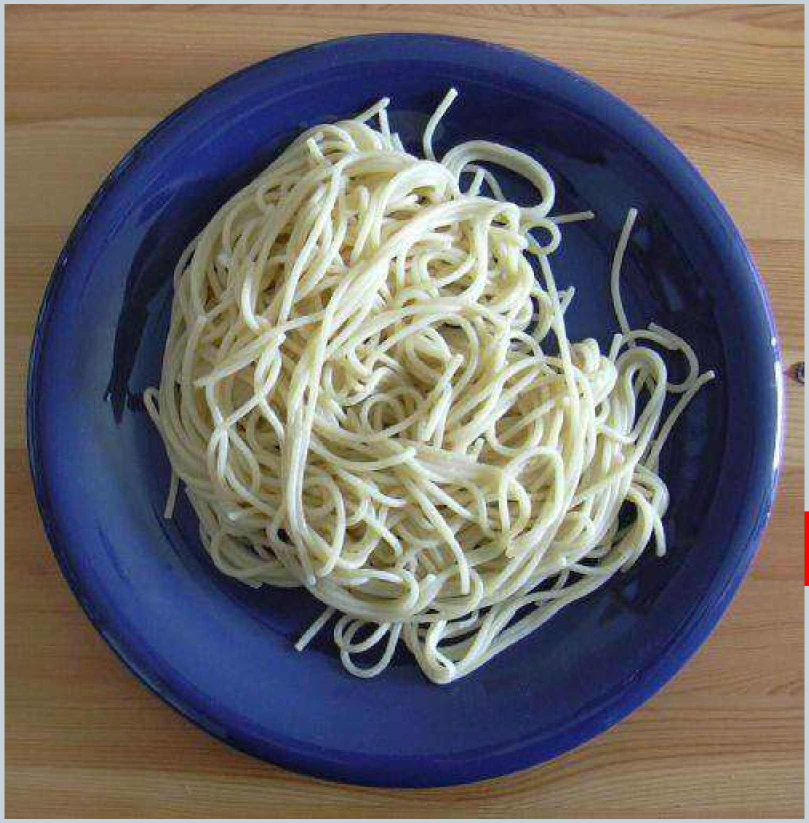
Why use Design Patterns?



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

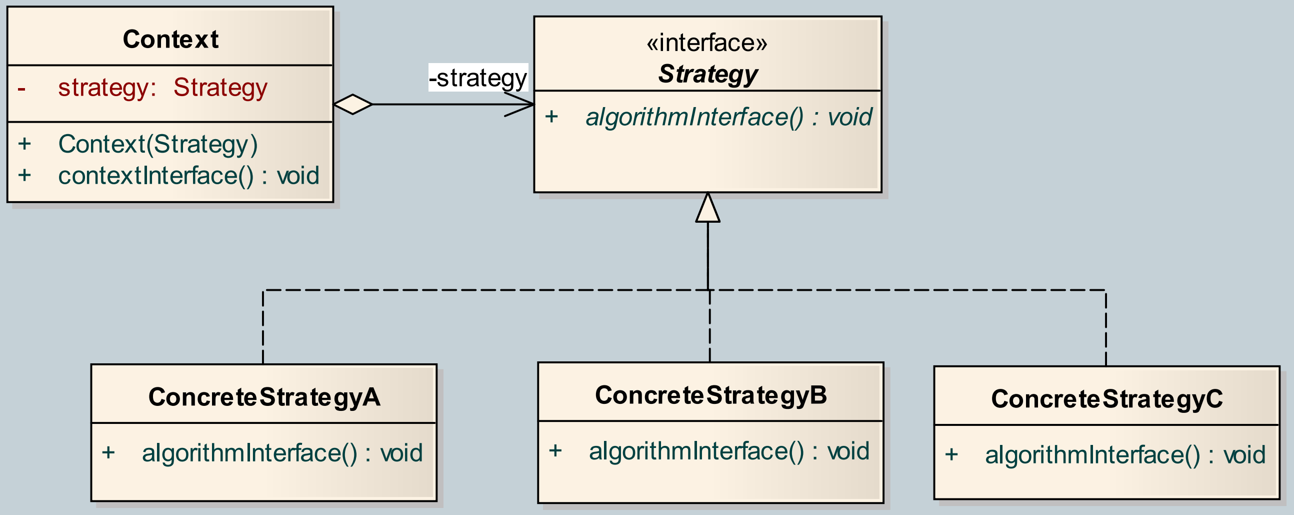
Strategy Definition

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

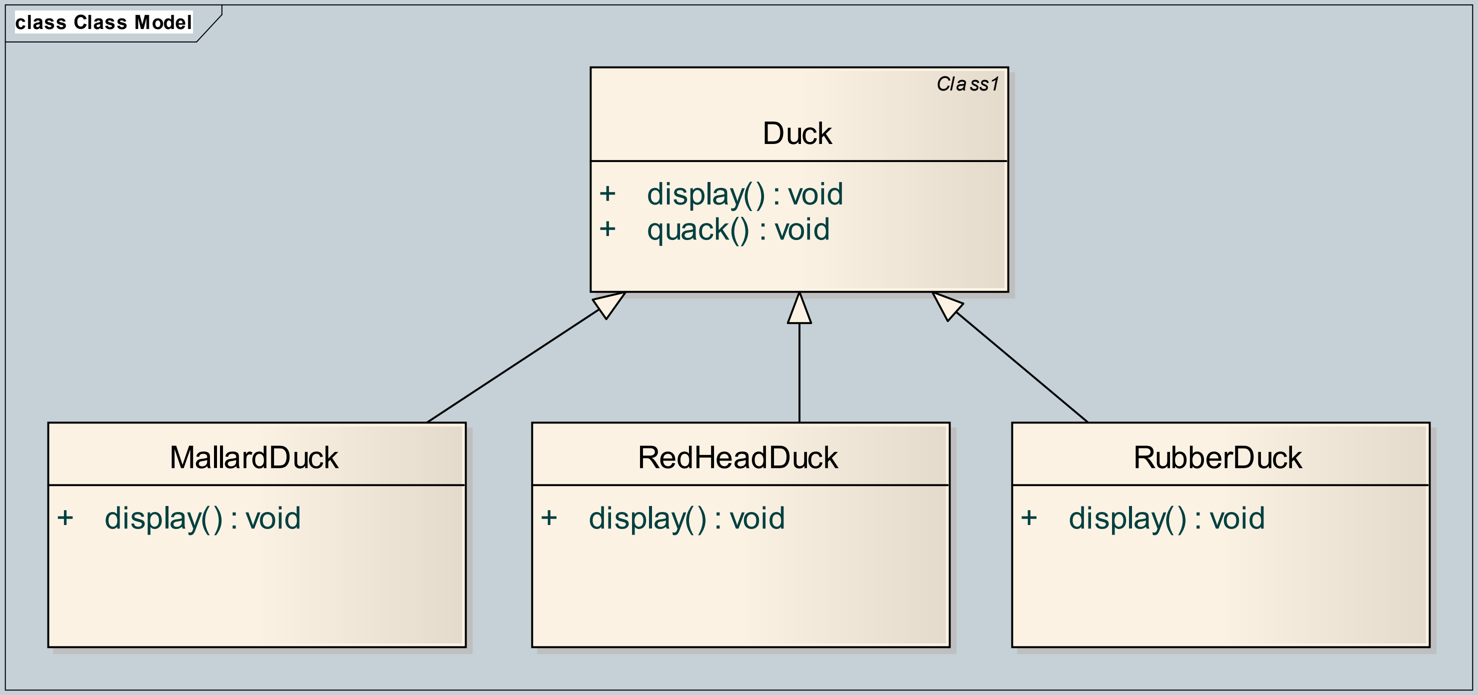
Design Principles

- 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

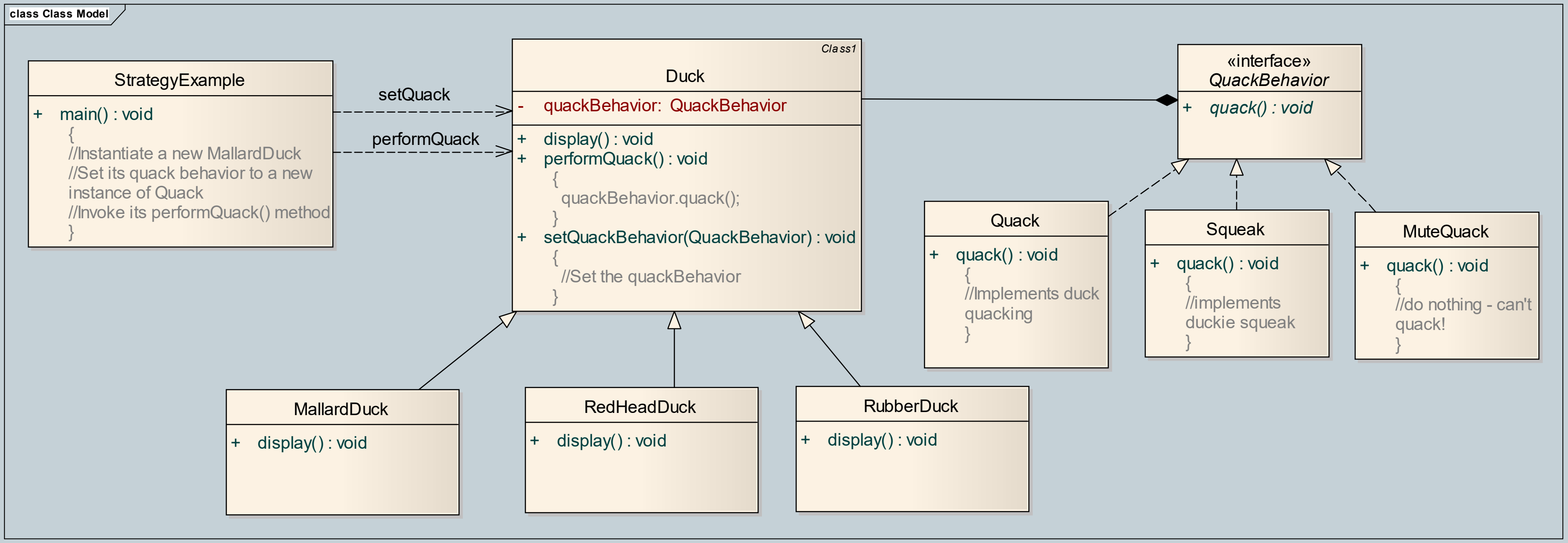
Strategy - Class diagram


Context
- strategy: Strategy
+ Context(Strategy)
+ contextInterface() : void
-strategy
«interface»
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

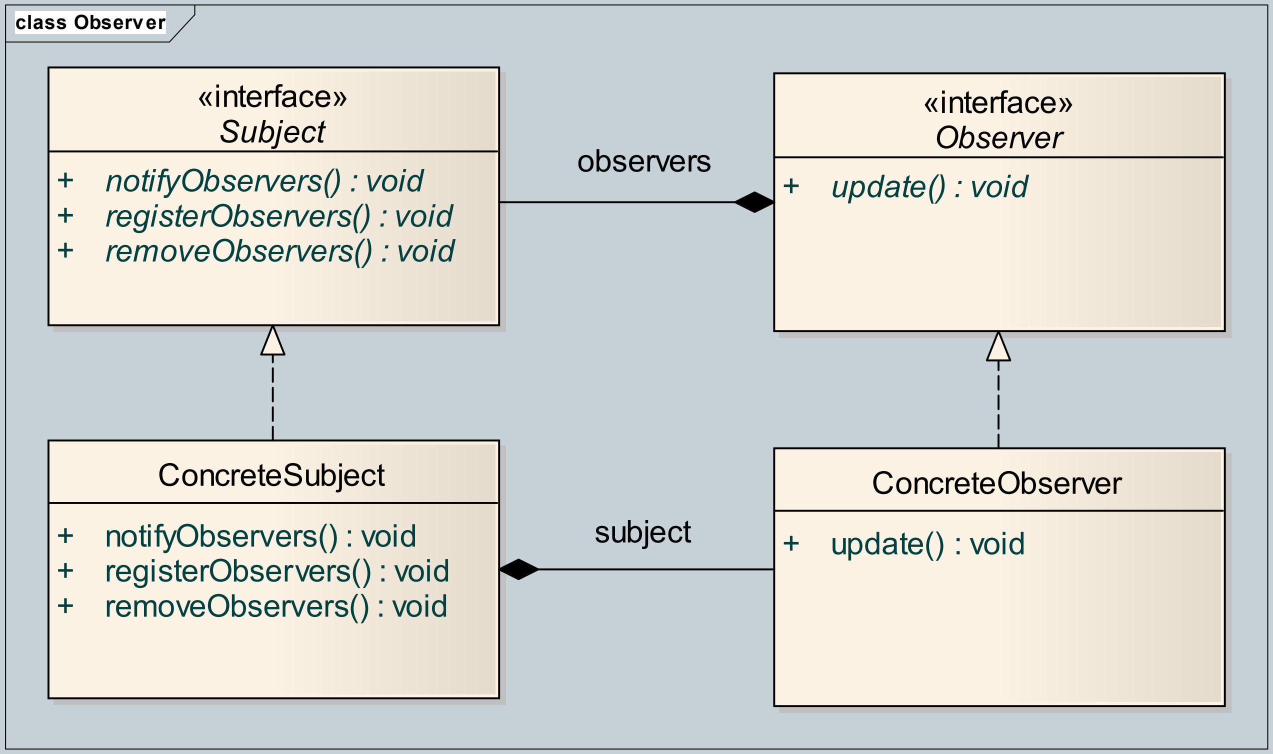
Observer Definition

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

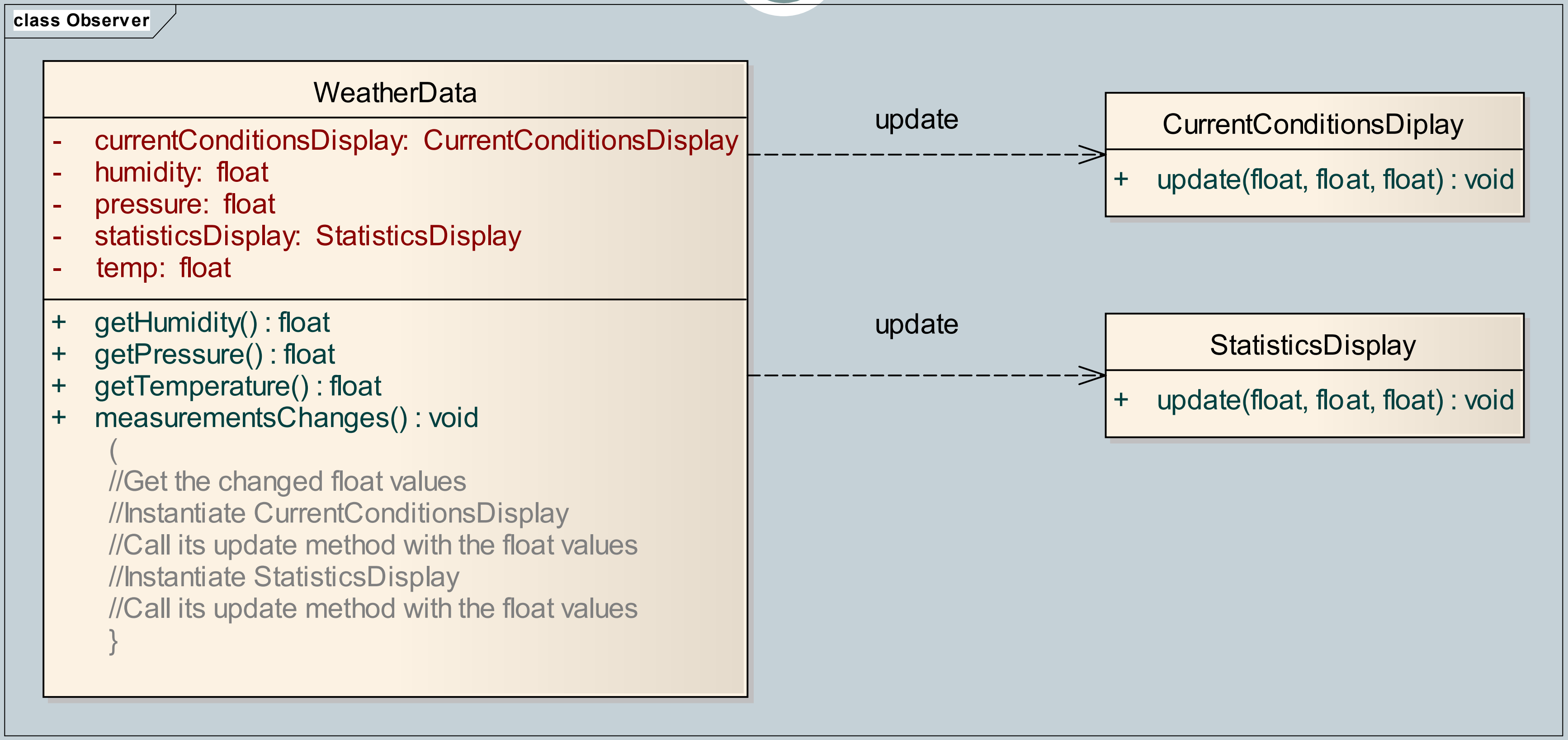
Design Principles

- 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

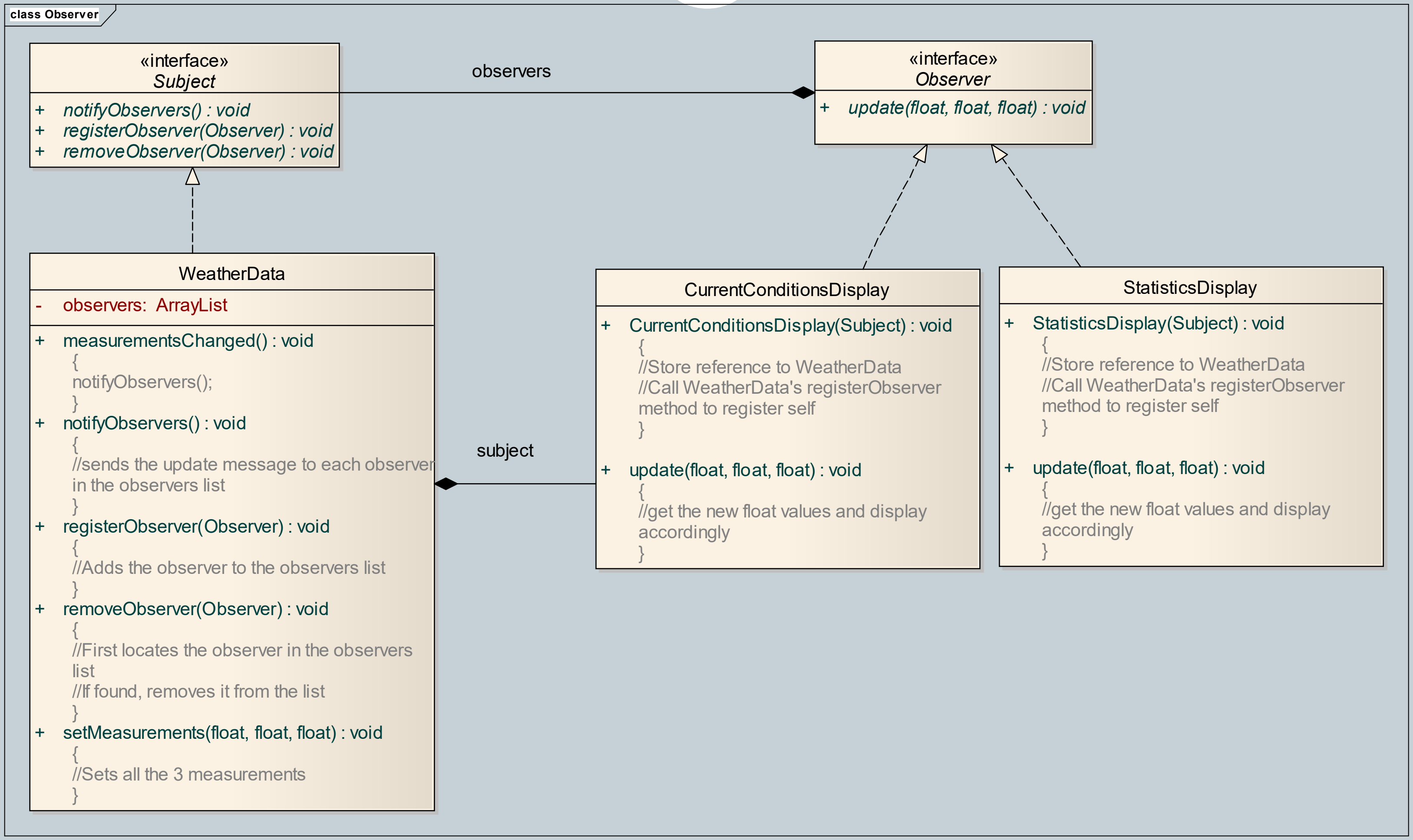
Observer - Class diagram


class Observer
«interface»
Subject
+ notifyObservers() : void
+ registerObservers() : void
+ removeObservers() : void
observers
«interface»
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
update
CurrentConditionsDiplay
+ update(float, float, float) : void
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
}

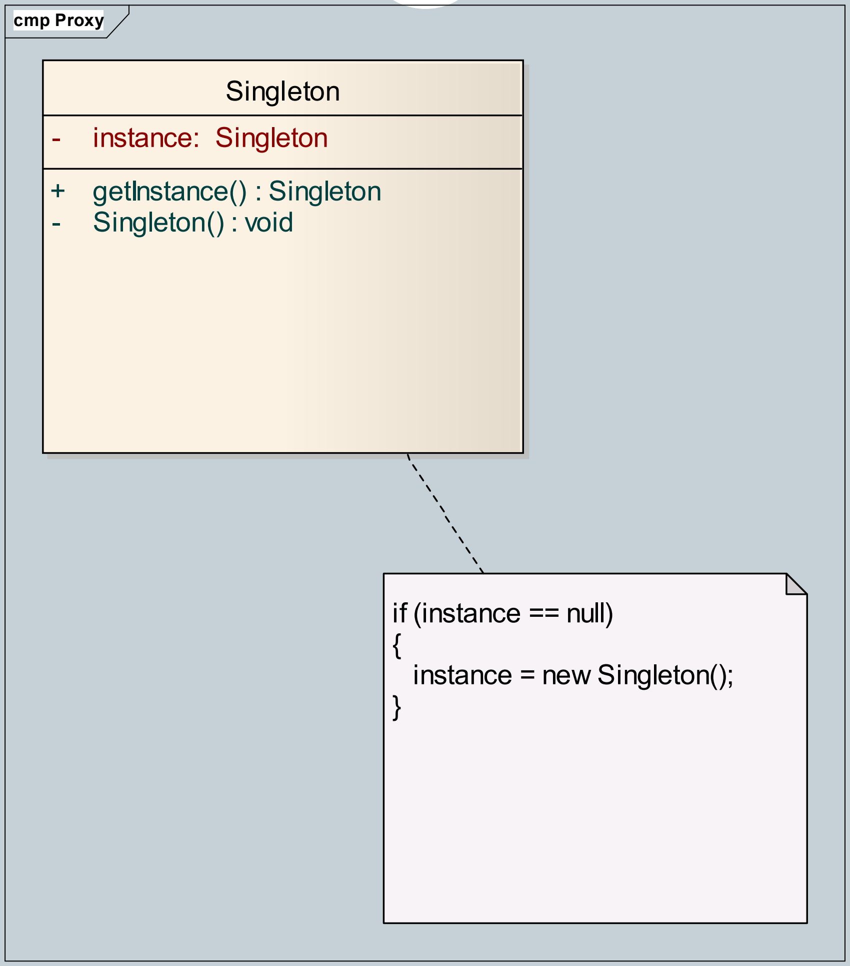
Observer

- 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 (with push)
  - Exposes the Subject to the Observer (with pull)

Singleton Definition

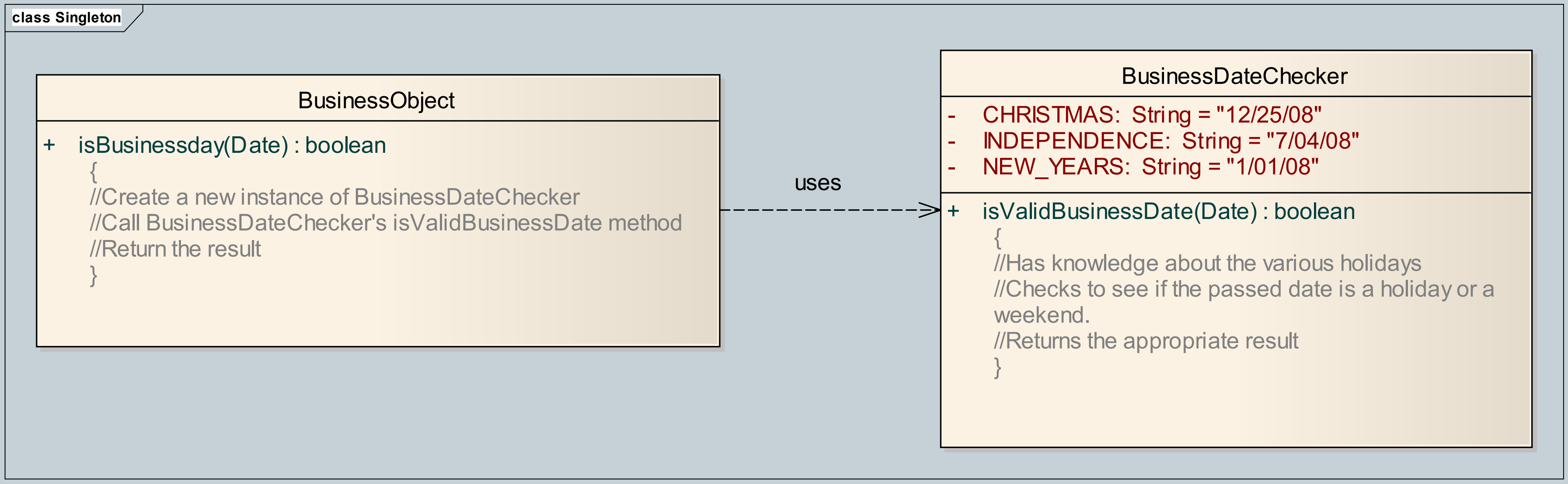
Ensure a class only has one instance and provide a global point of access to it.

Singleton - Class diagram

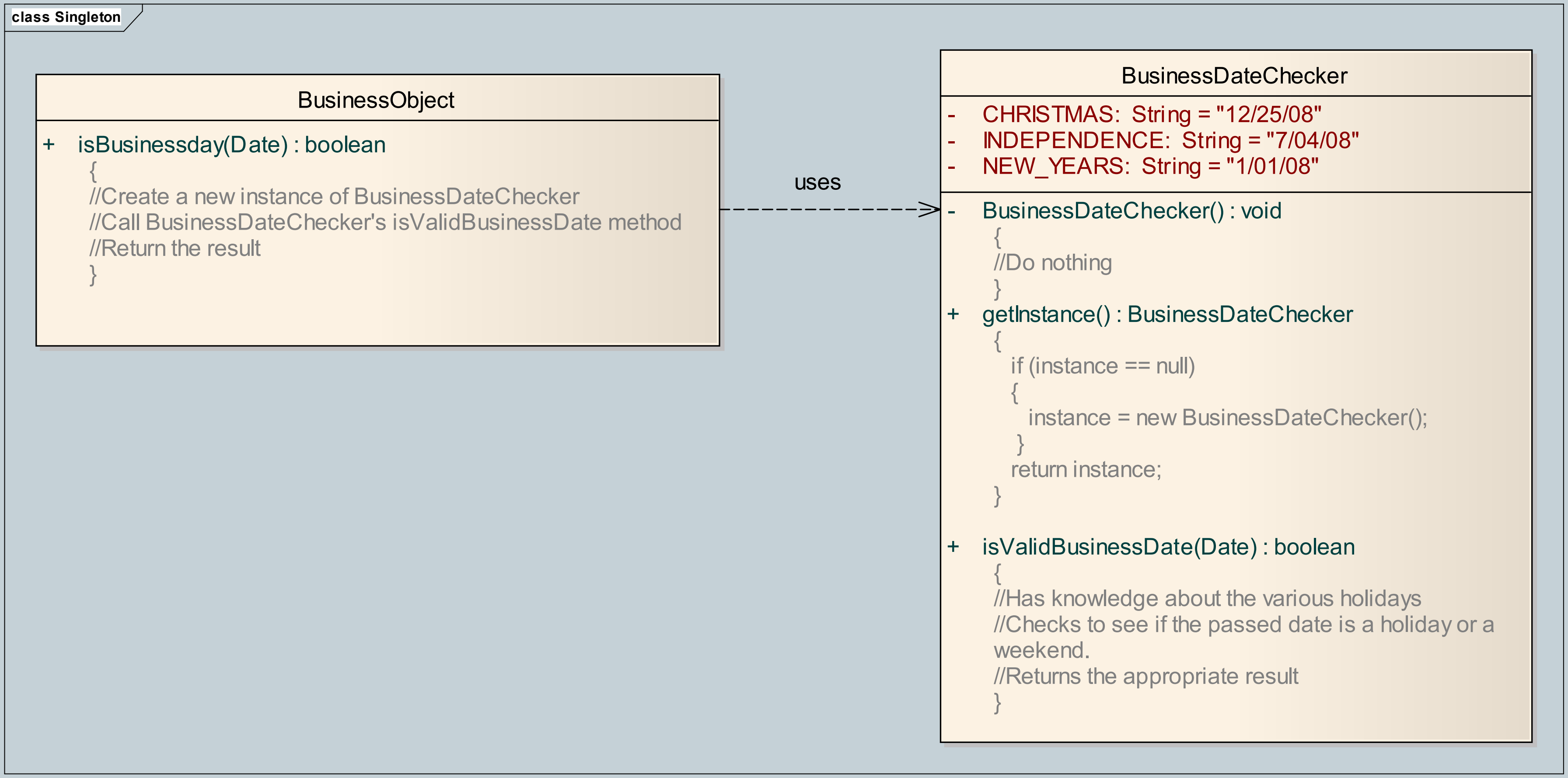


```
if (instance == null)
{
   instance = new Singleton();
}
```

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
      }
```

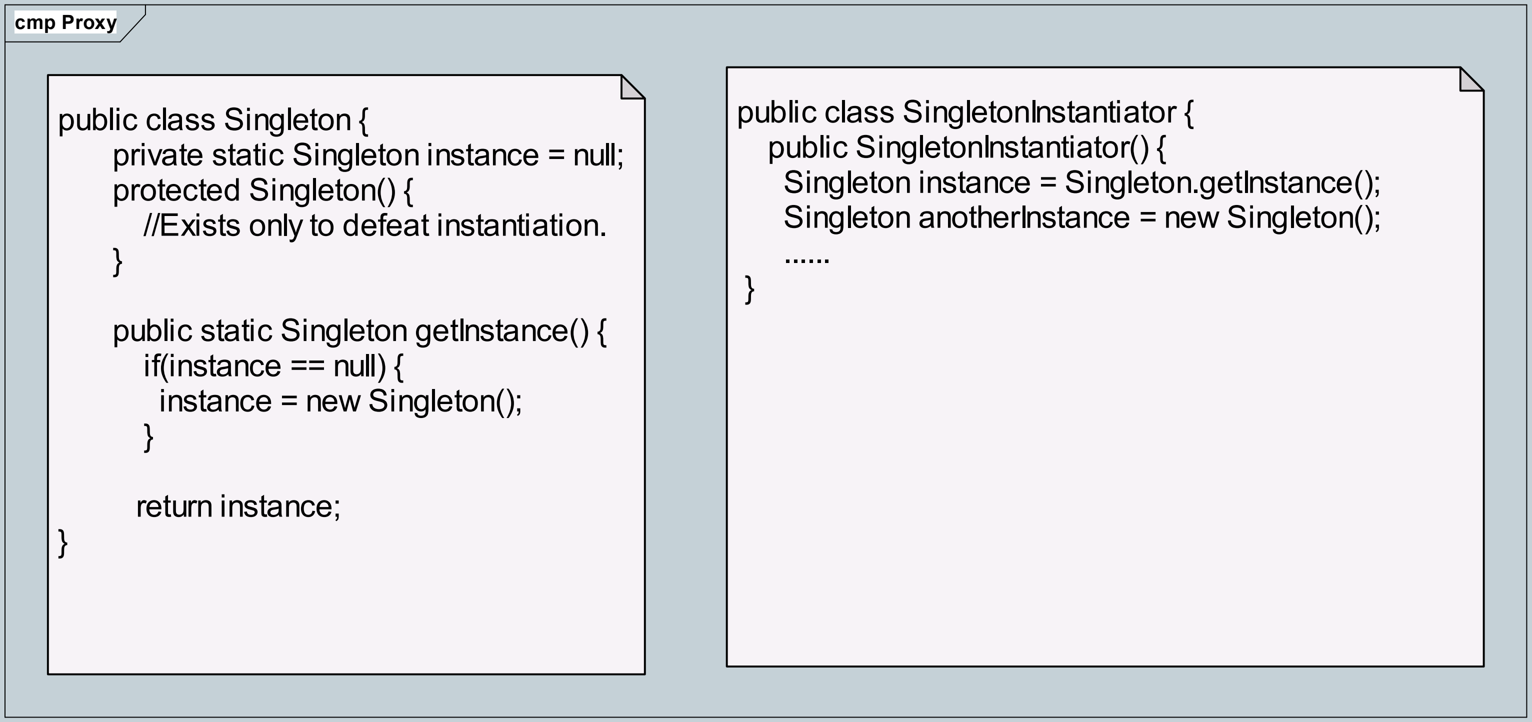
BusinessDateChecker

```
-   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



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

        public static Singleton getInstance() {
            if(instance == null) {
              instance = new Singleton();
            }

           return instance;
}
```

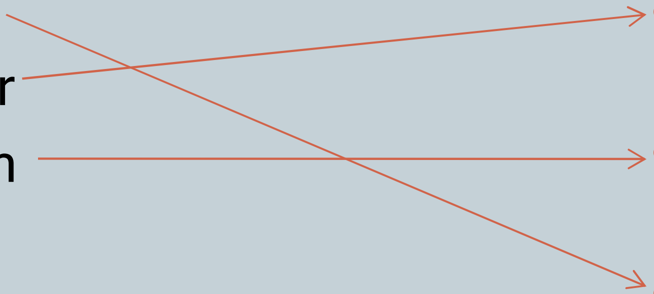
```
public class SingletonInstantiator {
    public SingletonInstantiator() {
      Singleton instance = Singleton.getInstance();
      Singleton anotherInstance = new Singleton();
      ......
 }
```

Singleton

- Pros
  - Increases performance
  - Prevents memory wastage
  - Increases global data sharing
- Cons
  - Results in multithreading issues

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

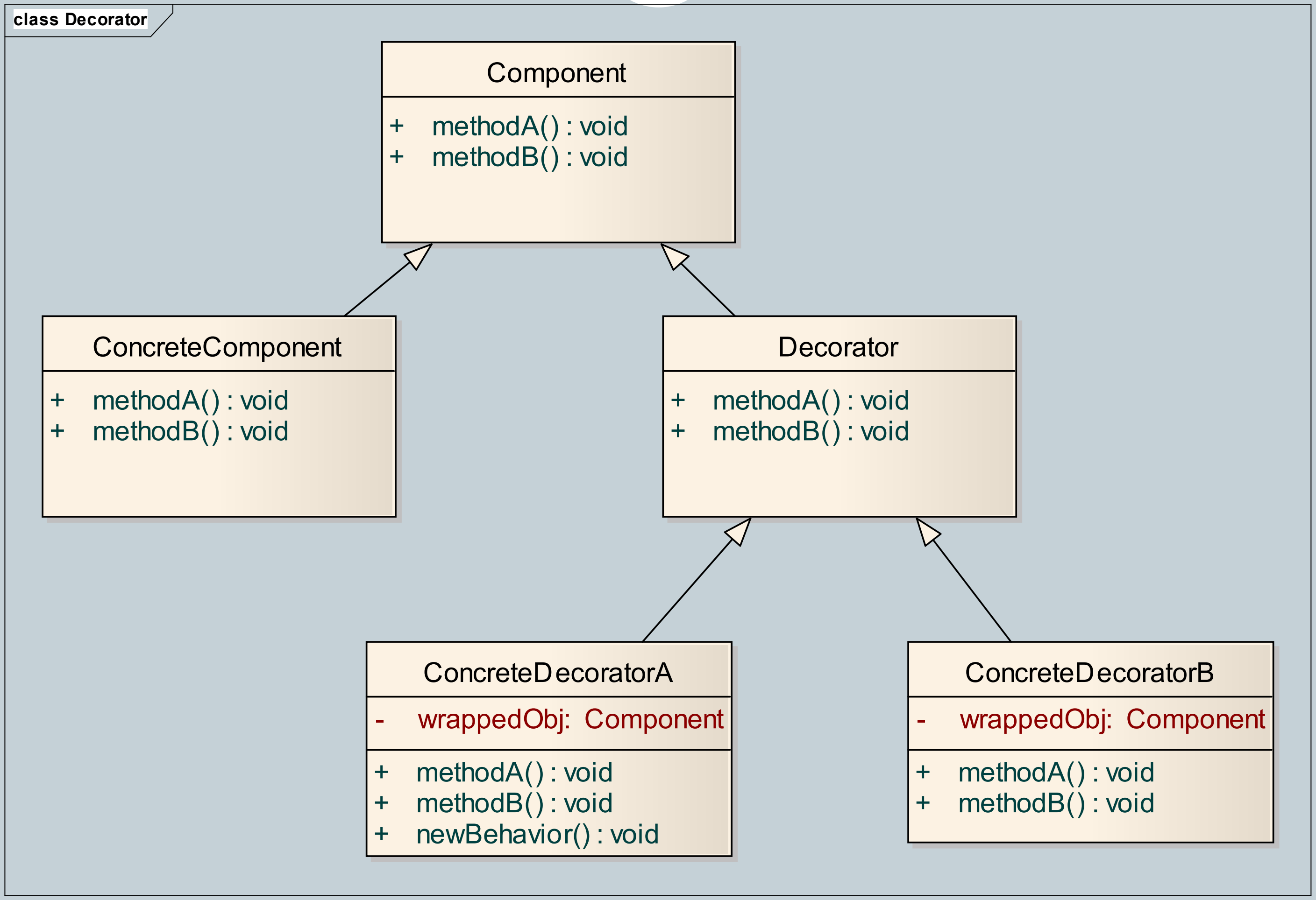
Decorator Definition

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

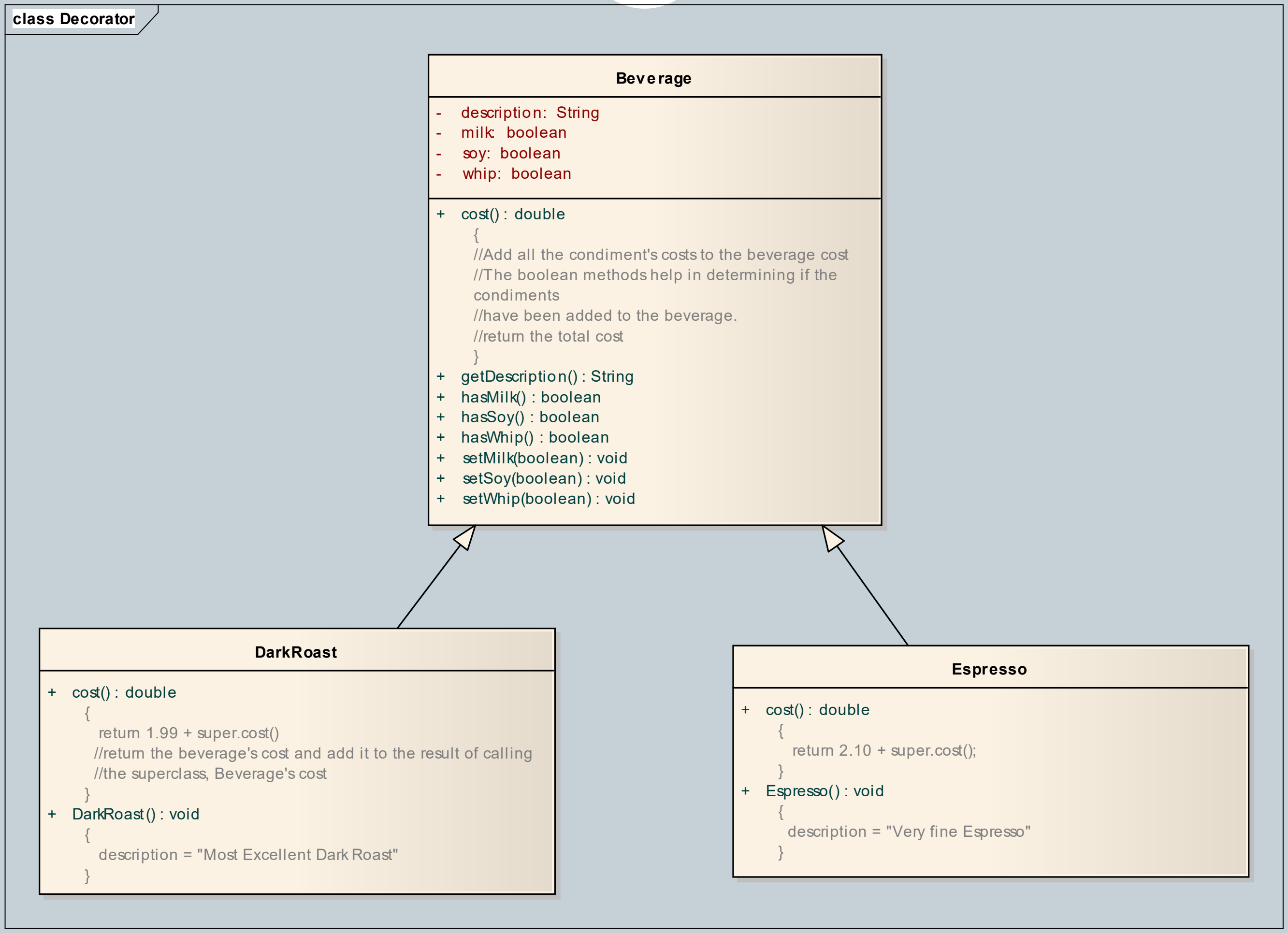
Design Principles

- 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



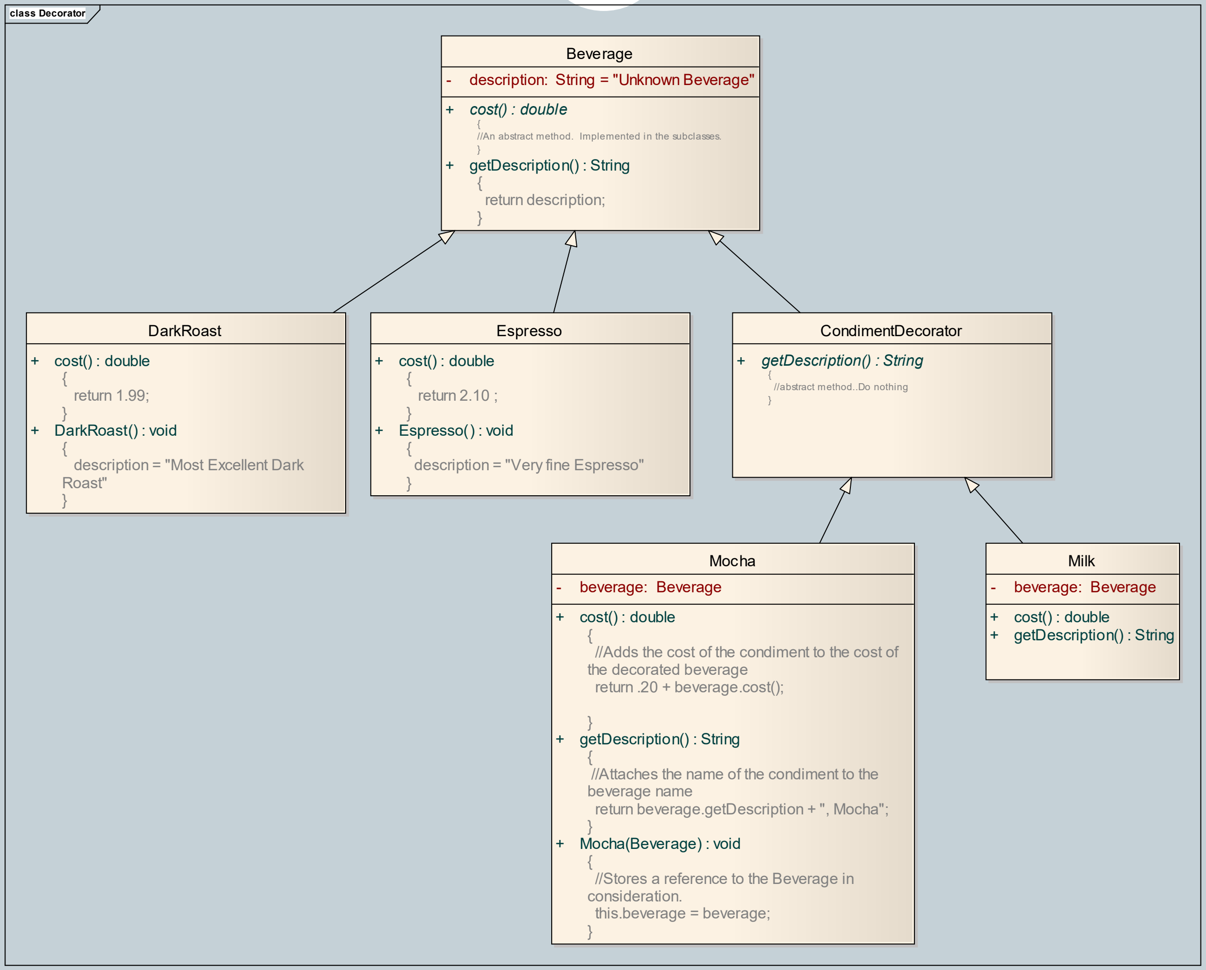
```
Beverage
-  description:  String
-  milk:  boolean
-  soy:  boolean
-  whip:  boolean

+  cost() : double
     {
     //Add all the condiment's 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


class Decorator
Beverage
- description: String = "Unknown Beverage"
+ cost() : double
{
//An abstract method. Implemented in the subclasses.
}
+ getDescription() : String
{
return description;
}
DarkRoast
+ cost() : double
{
return 1.99;
}
+ DarkRoast() : void
{
description = "Most Excellent Dark Roast"
}
Espresso
+ cost() : double
{
return 2.10 ;
}
+ Espresso() : void
{
description = "Very fine Espresso"
}
CondimentDecorator
+ getDescription() : String
{
//abstract method..Do nothing
}
Mocha
- beverage: Beverage
+ cost() : double
{
//Adds the cost of the condiment to the cost of the decorated beverage
return .20 + beverage.cost();
}
+ getDescription() : String
{
//Attaches the name of the condiment to the beverage name
return beverage.getDescription + ", Mocha";
}
+ Mocha(Beverage) : void
{
//Stores a reference to the Beverage in consideration.
this.beverage = beverage;
}
Milk
- beverage: Beverage
+ cost() : double
+ getDescription() : String

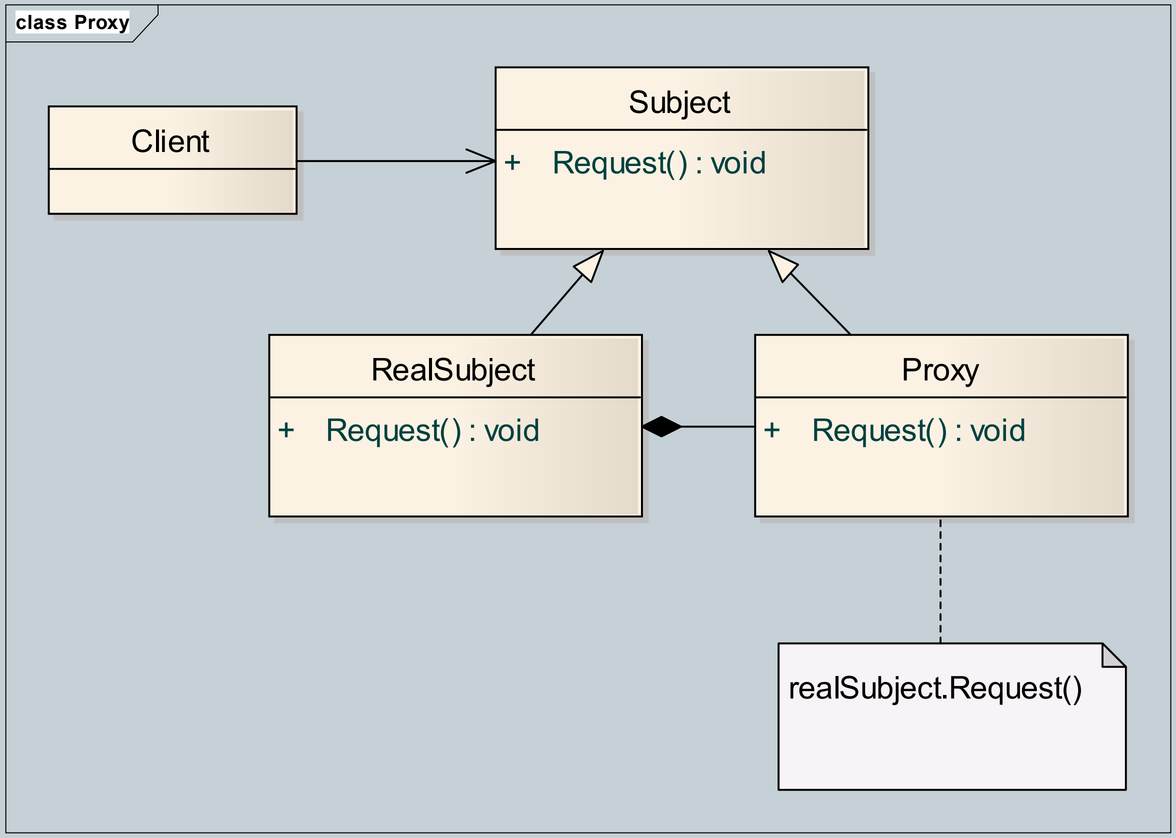
Decorator

- 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

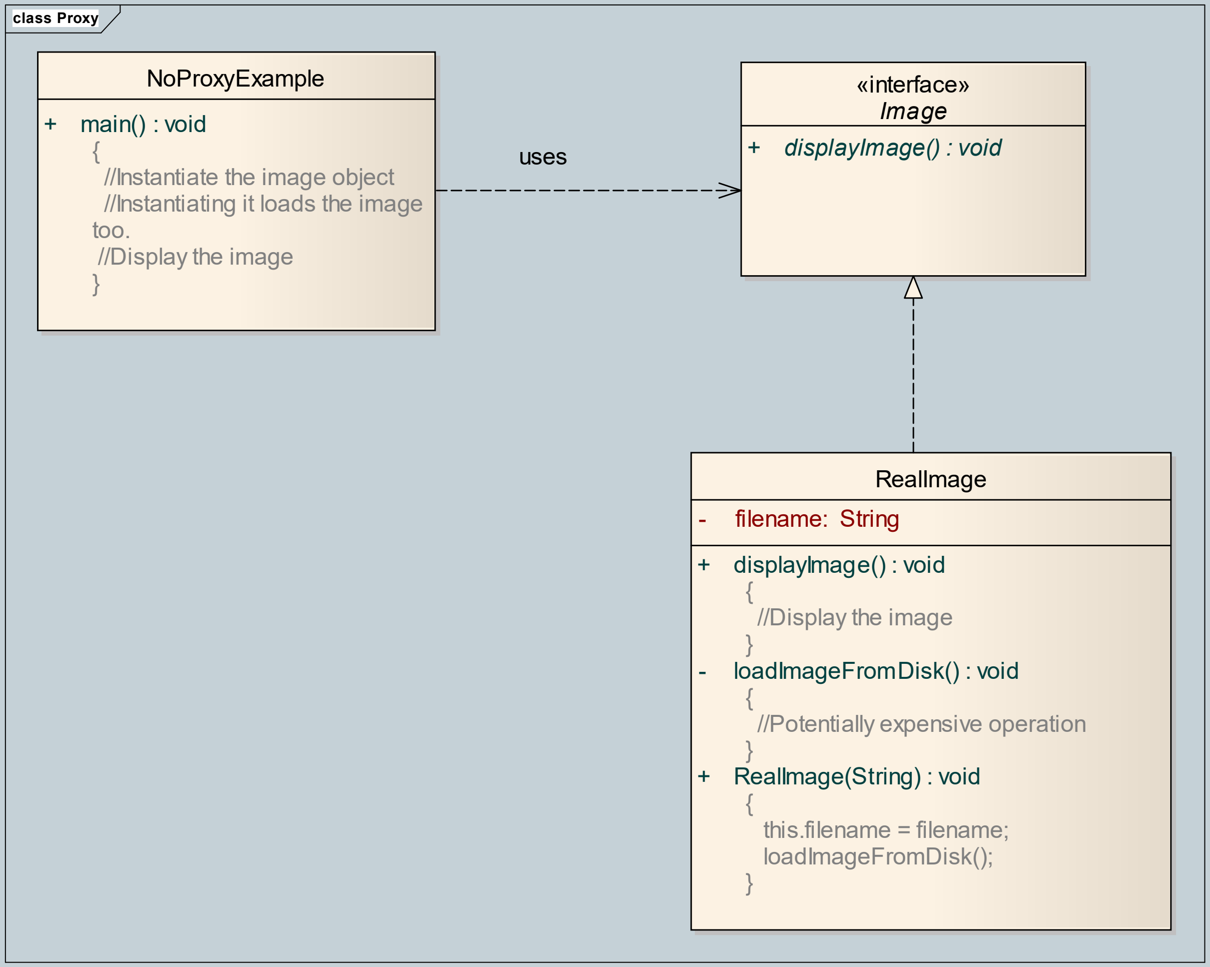
Proxy Definition

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

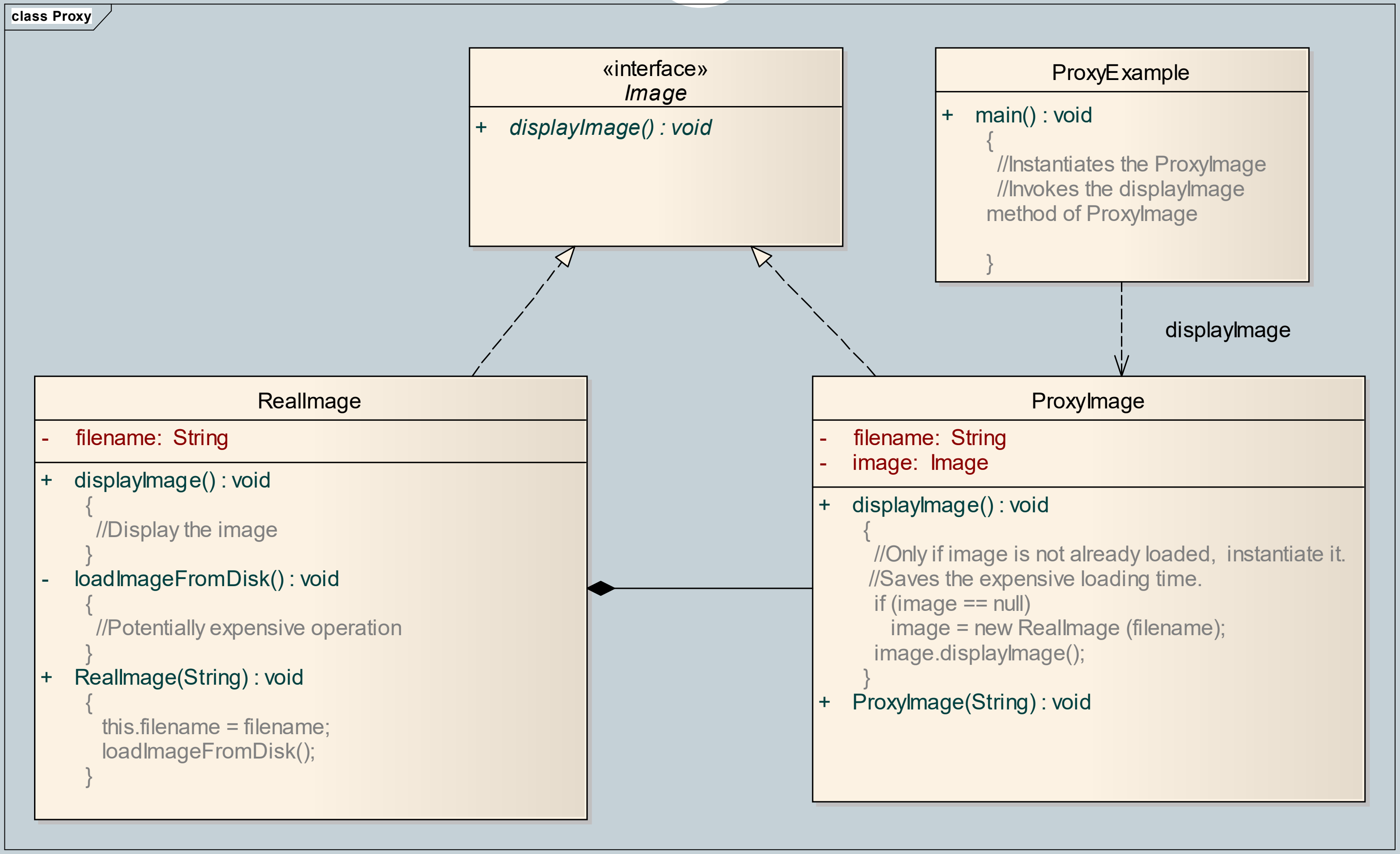
Proxy - Class diagram



Proxy - Problem



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

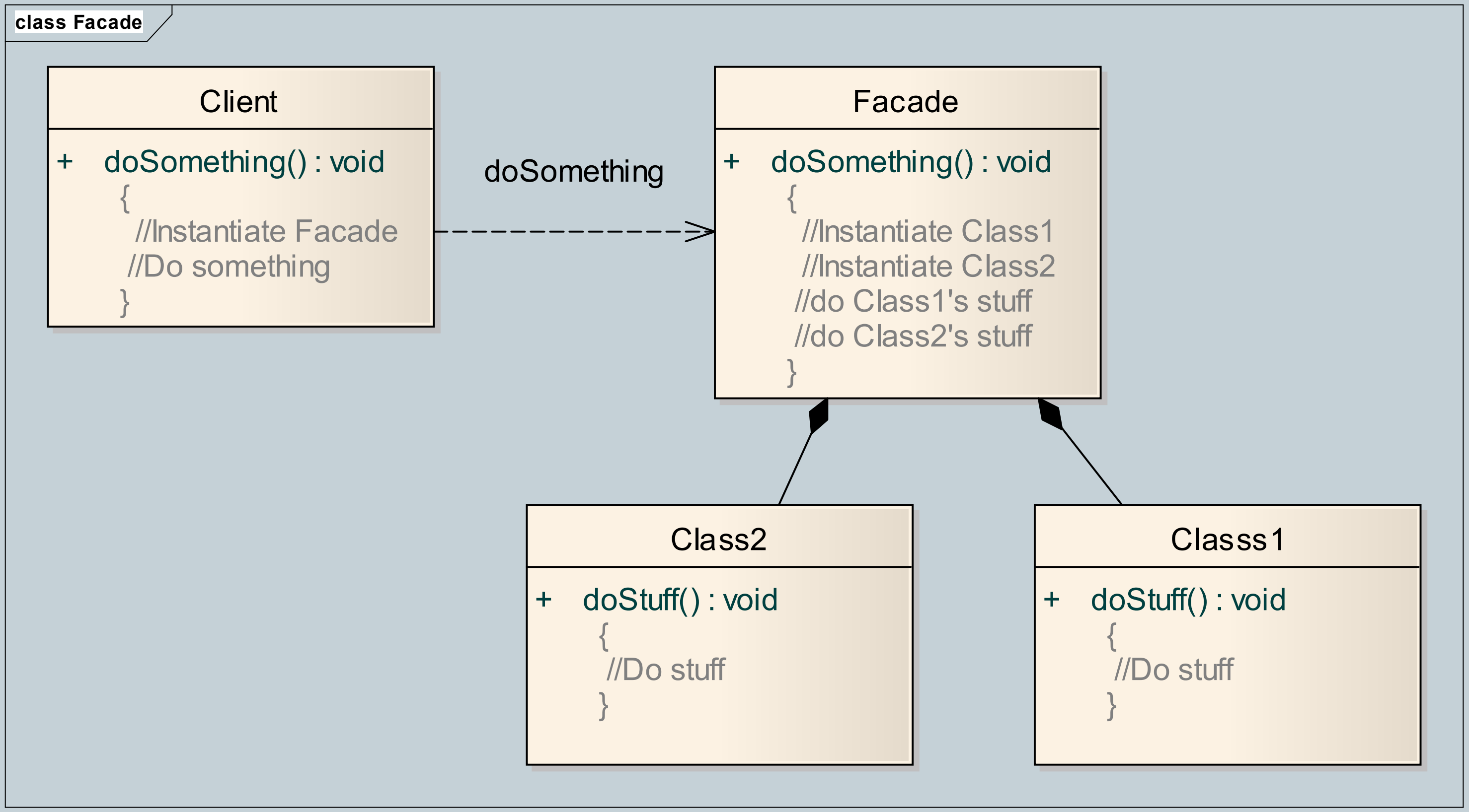
Facade Definition

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.

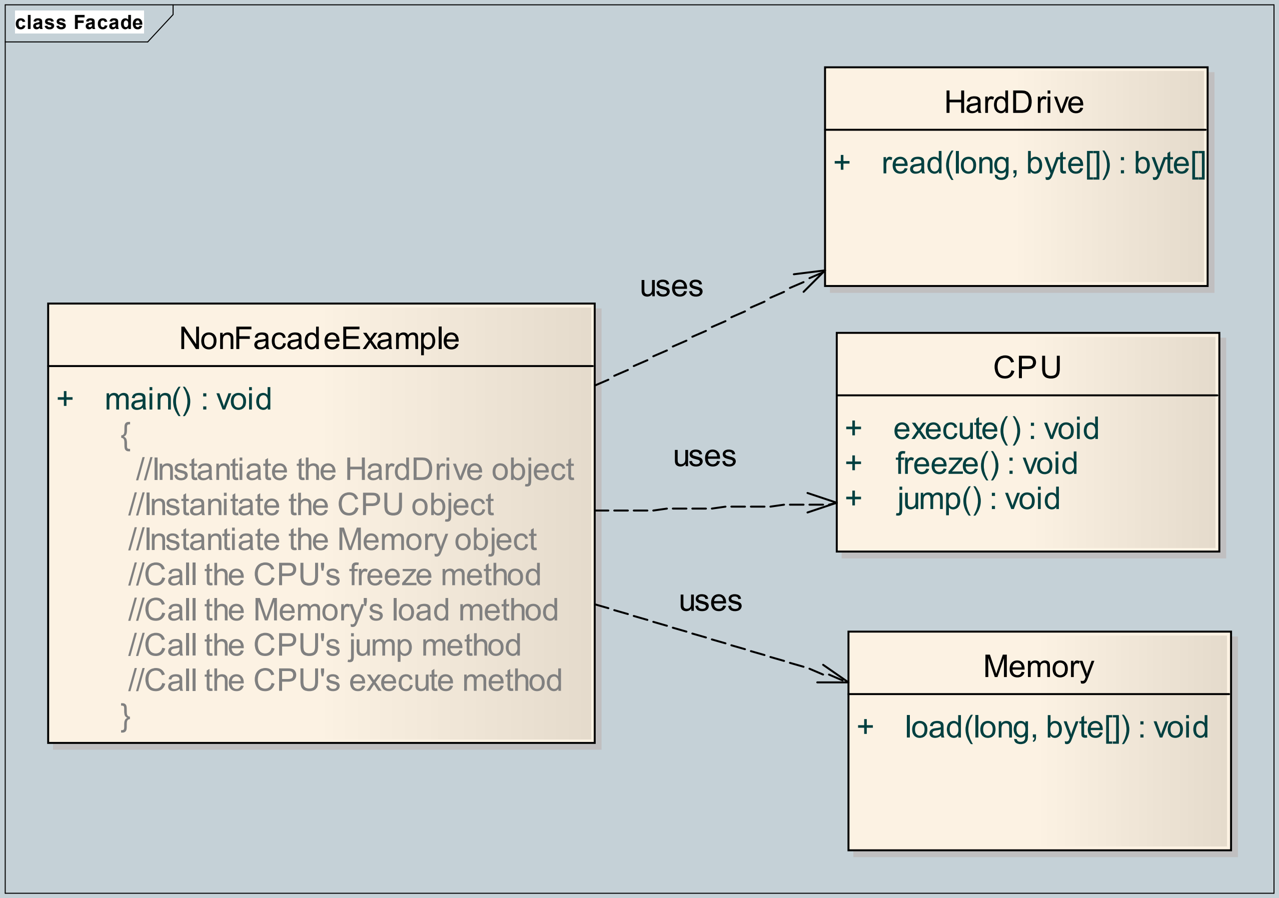
Design Principles

- 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

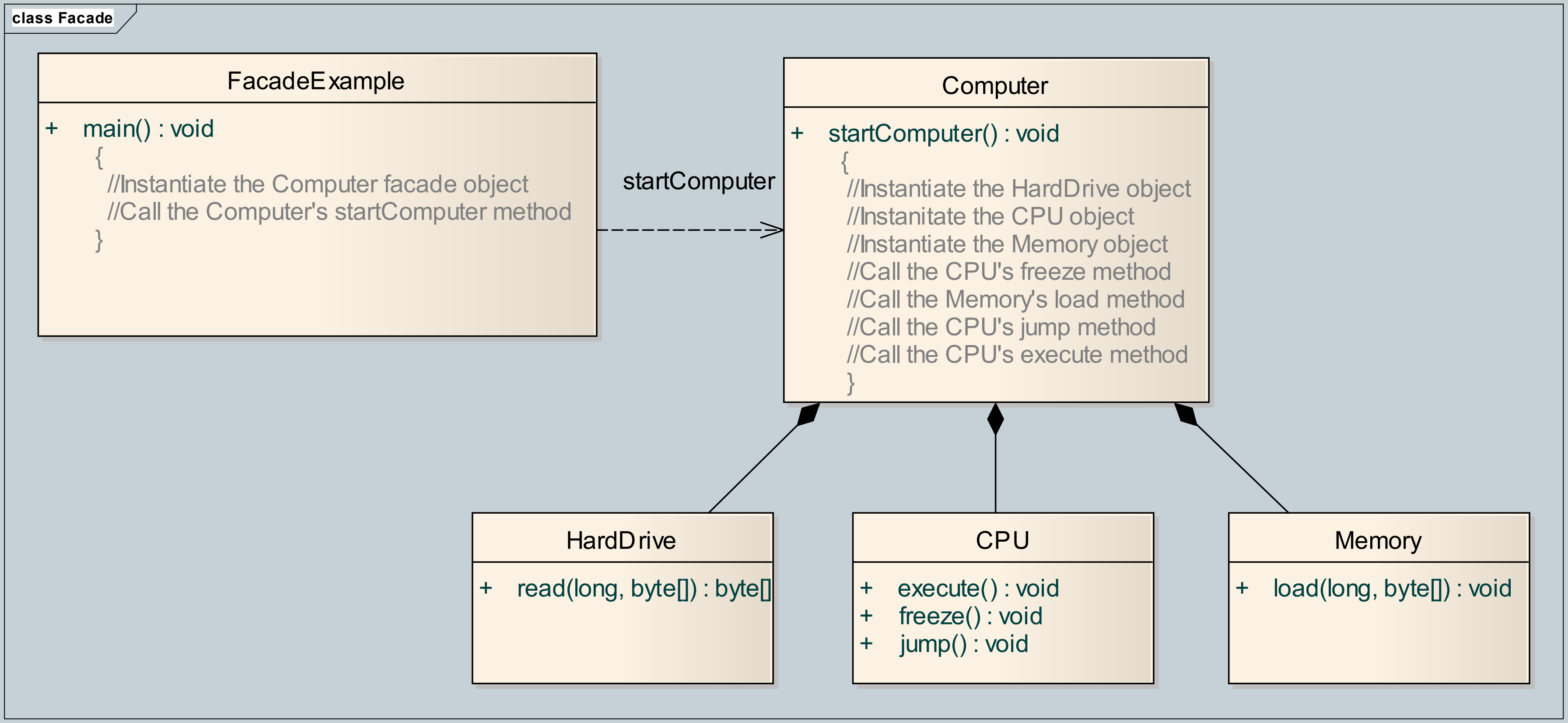
Façade - Class diagram



Façade - Problem


class Facade
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
uses
uses
HardDrive
+ read(long, byte[]) : byte[]
CPU
+ execute() : void
+ freeze() : void
+ jump() : void
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

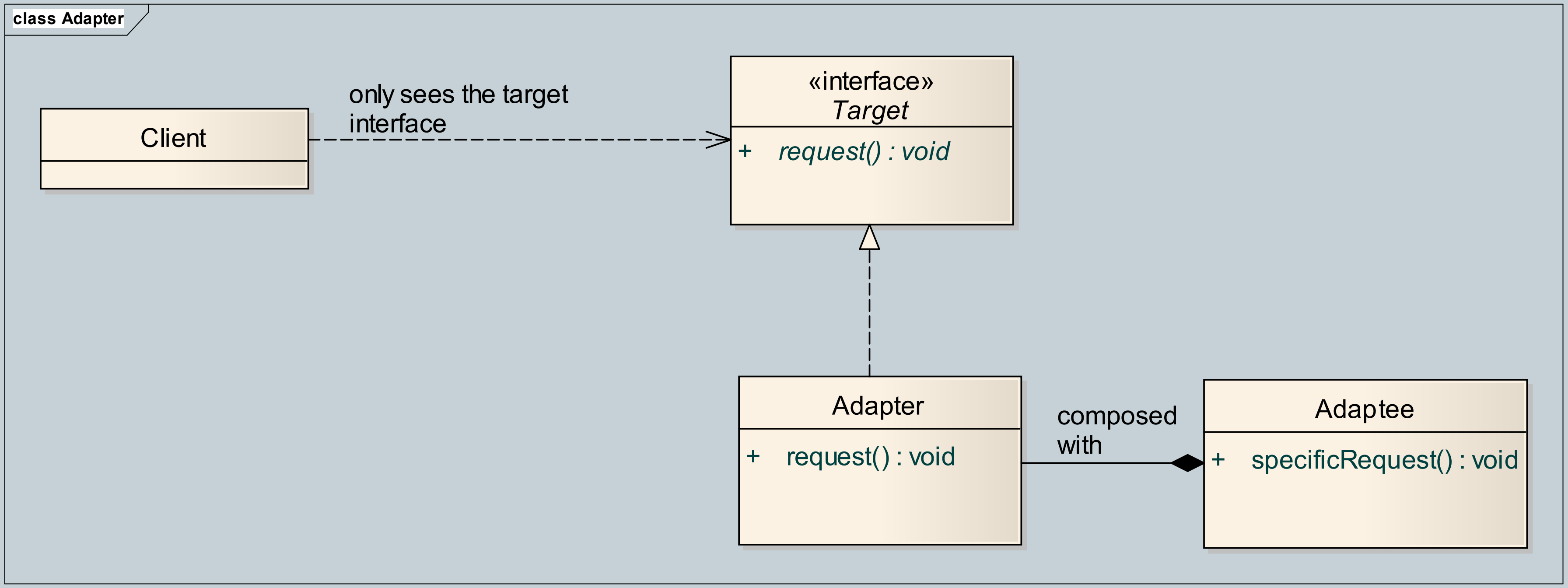
Facade

- 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

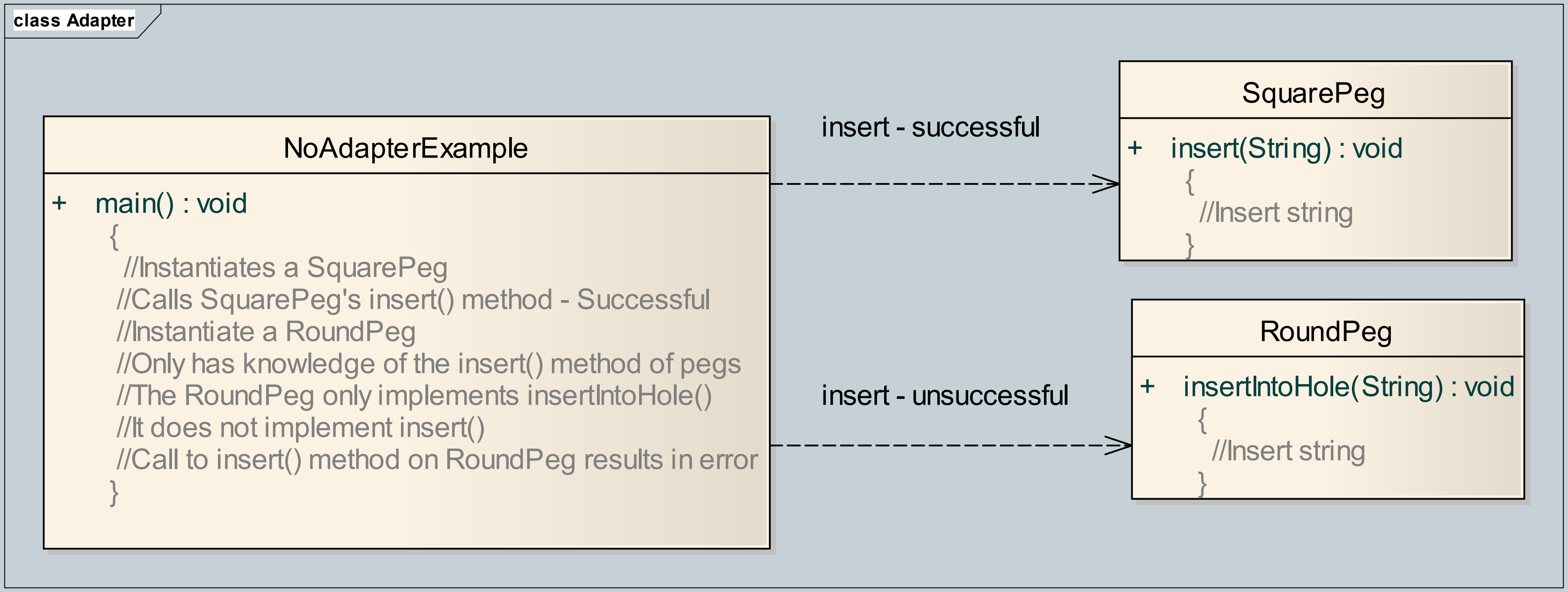
Adapter Definition

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.

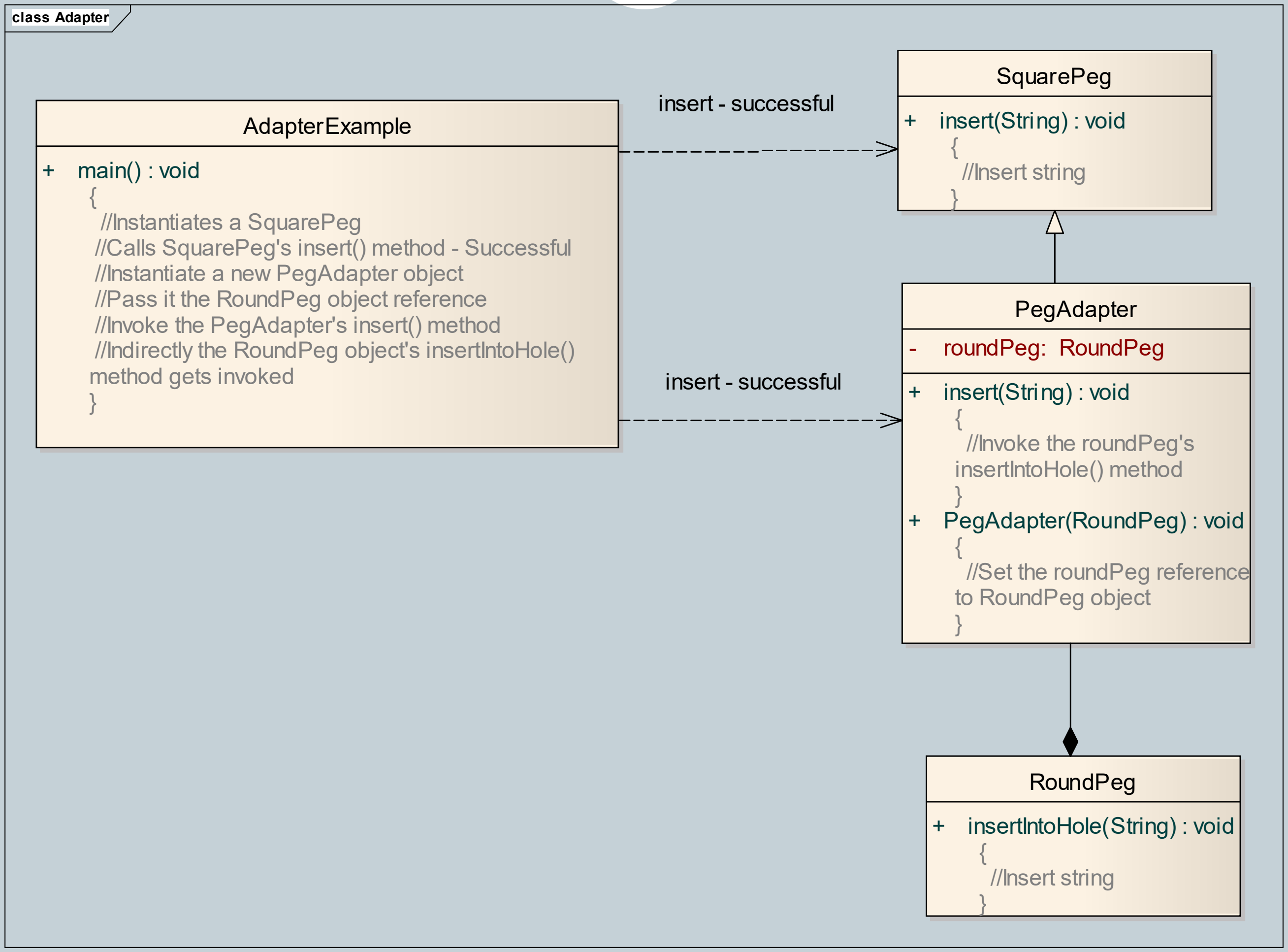
Adapter - Class diagram



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
insert - unsuccessful
SquarePeg
+ insert(String) : void
{
//Insert string
}
RoundPeg
+ insertIntoHole(String) : void
{
//Insert string
}

Adapter - Solution

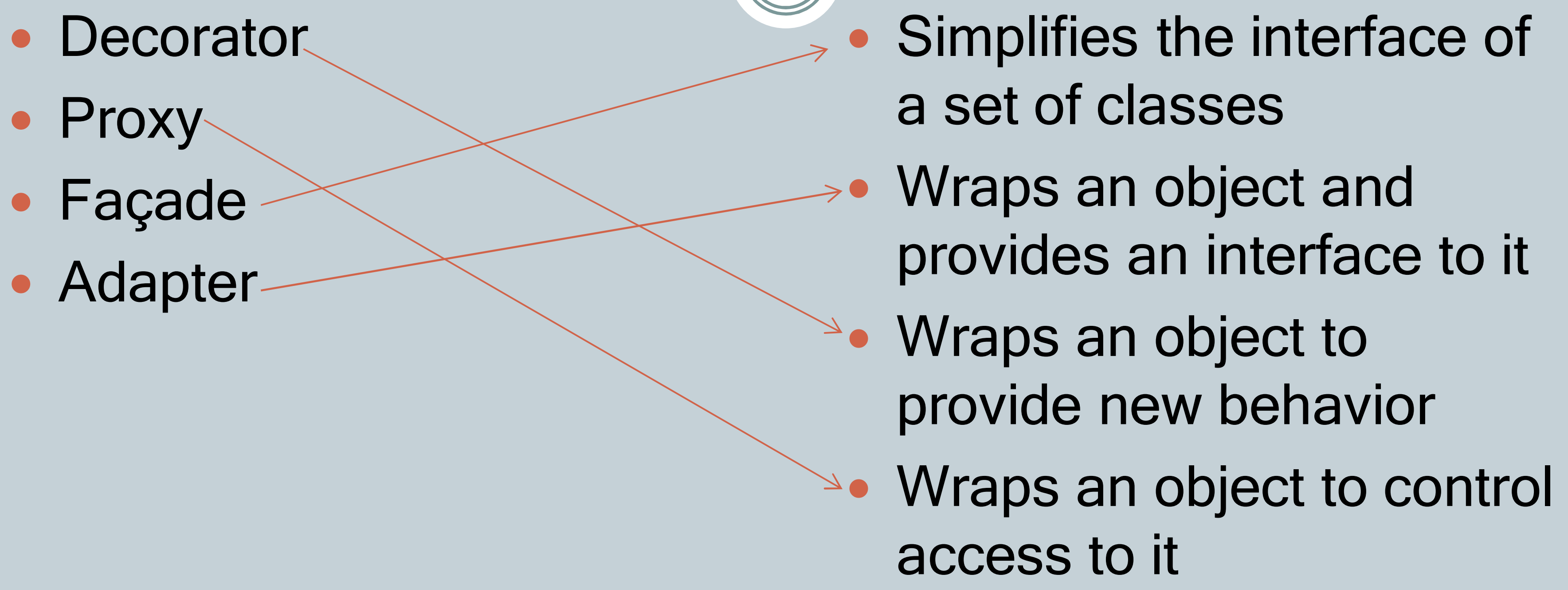


class Adapter
AdapterExample
+ main() : void
{
//Instantiates a SquarePeg
//Calls SquarePeg's insert() method - Successful
//Instantiate a new PegAdapter object
//Pass it the RoundPeg object reference
//Invoke the PegAdapter's insert() method
//Indirectly the RoundPeg object's insertIntoHole() method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

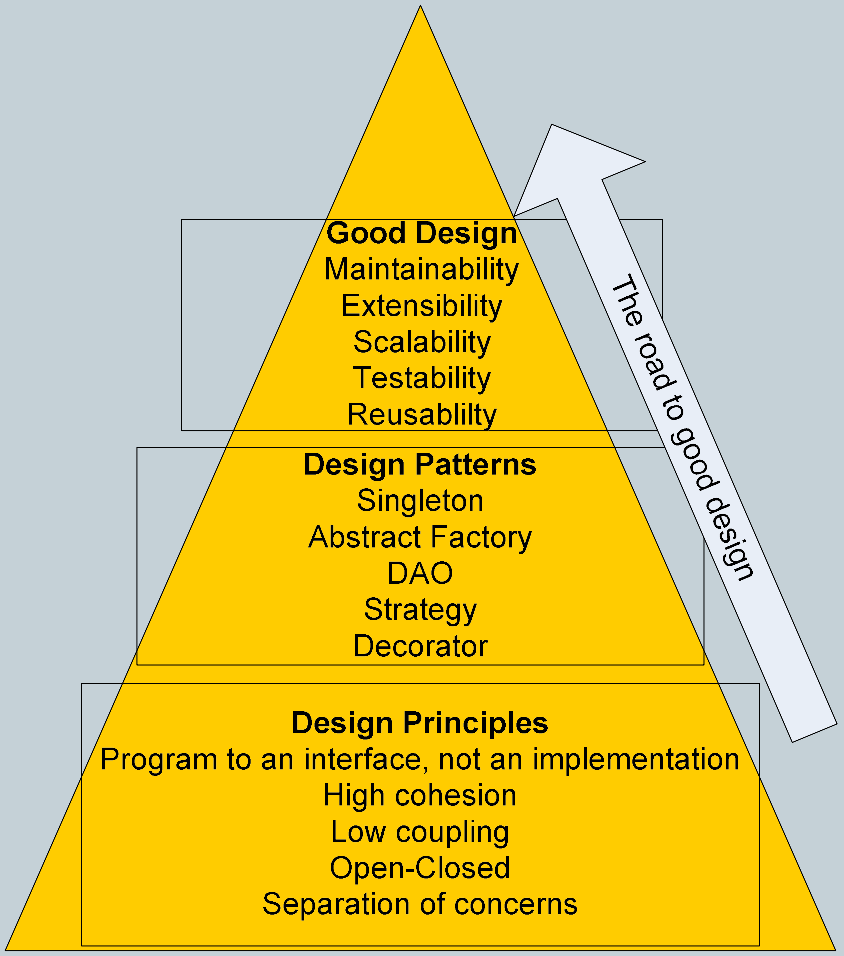
Pattern Classification

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