Software Design Patterns

YOUR UTILITY IT PLATFORM

What will we learn today?

- A quick refresher on OOP
  - Classes/Objects, inheritance, polymorphism, example
- What is a software design pattern?
  - Origins
  - Why would you use them?
  - How can you apply them?
- Types of patterns
  - Singleton Pattern
  - Null object pattern
  - Composite pattern
- Summary
- Three exercises
- Questions?

A quick refresher on OOP

Classes/Objects, inheritance, polymorphism, example

YOUR UTILITY IT PLATFORM

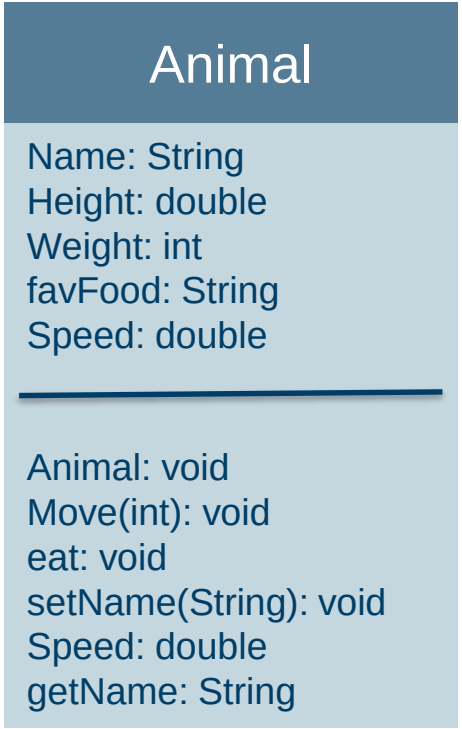
Object
Lorem Ipsum Dolor Sit Amet

Animal
Name: String Height: double Weight: int favFood: String Speed: double Animal: void Move(int): void eat: void setName(String): void Speed: double getName: String

Classes and objects

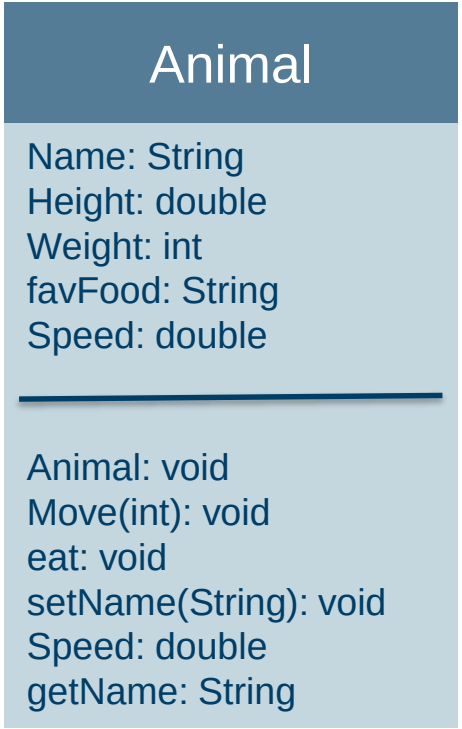
- Class is a blueprint for an object
- Class defines object properties and behavior
- You create an instance of an object, not a class

- Objects can have states, defined by properties
- Objects can have behavior, defined by methods

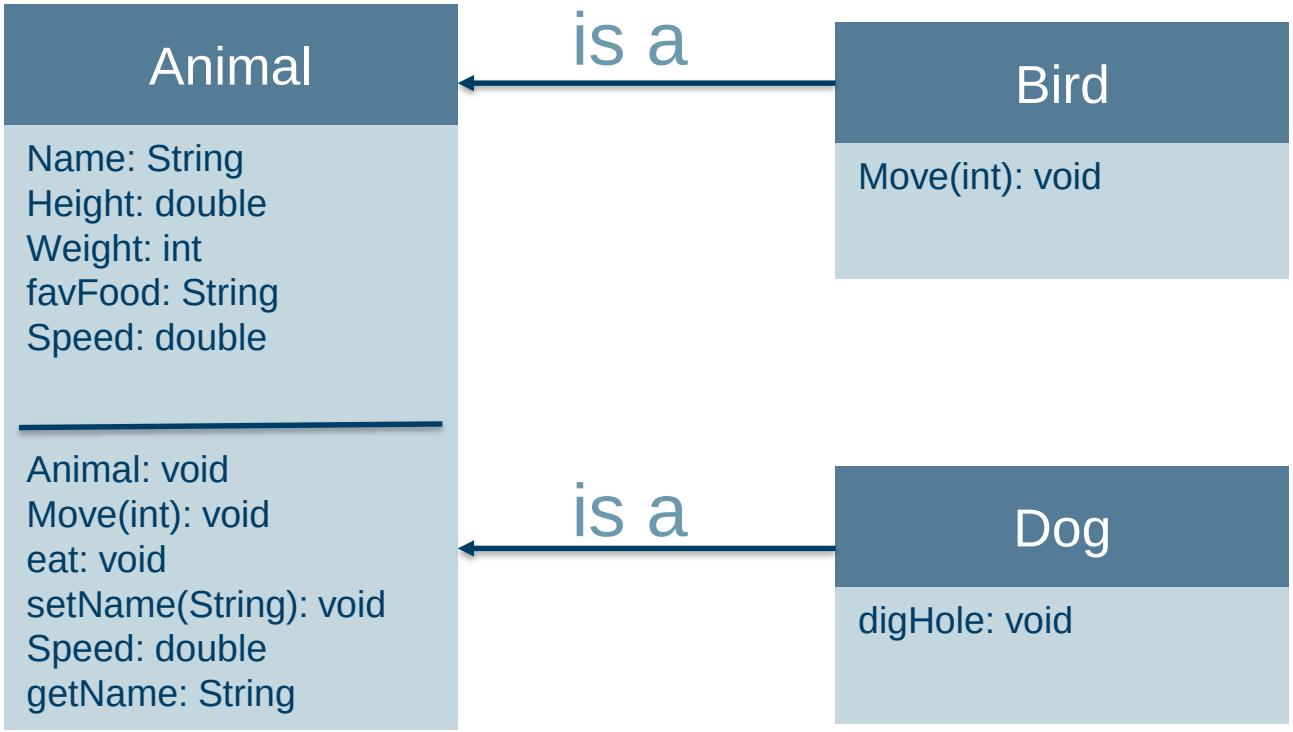


Inheritance

- A way to define common behavior and state in one place while defining specific behavior elsewhere
- Classes inherit properties/methods from a superclass
- Common behavior and state in abstract classes, the ‘final’ class that uses it is the concrete implementation



Inheritance



When to apply inheritance?

Ask the “is a“ question:

- Is a Dog an Animal? Yes
- Is a Bird an Animal? Yes
- Is a Laptop a Computer? Yes
- Is a Bus a Car? Yes
- Is an Apache Attack Helicopter a Politician? No

Wrong!

Microwave
bark: void urinate: void eatHomework: void

Use this to decide whether you can define common properties like a dogs height or a laptops dimensions or find situations where inheritance might not be so logical

Some situations are not as obvious as the given example

Why use inheritance?

- Avoid duplicate code (efficiency)
- Changes to superclass instantly reflected in subclasses (maintainability)
- Easier for yourself and other developers to understand code (understandability)

Polymorphism

Cast a subclass object in the form of a superclass

Allows you to work with objects in superclass form while retaining some of their subclass properties and methods

Animal muttley = new Dog();

Animal garfield = new Cat();

Animal bob = new Flamingo();

Animals[ ] array
Dog Object
Cat Object
Flamingo Object

Methods are executed from subclass, but only if they are also in superclass

garfield.getSound();

> *“Mew mew mew”*

Quick Demo

- Making a superclass and giving it properties/methods
- Making a subclass and giving it properties/methods

Design patterns

The what, why and how, history, examples and exercises

YOUR UTILITY IT PLATFORM

What is a software design pattern?

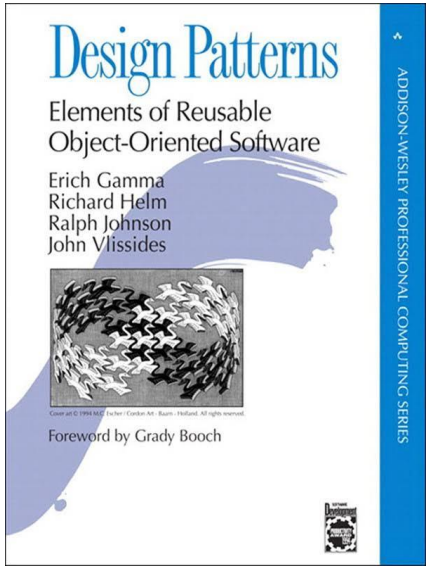
- A template for solving a common design challenge
  - A pattern has a name, intent/purpose, solution, etc.
  - It describes common high-level structures in code, beyond the level of the programming language
  - Patterns are generic, but the exact “wording” of the solution depends on the programming language

- Examples of software design challenges
  - Representing a complex object (e.g. a web page)
  - Handling different file formats (e.g. txt, doc, docx)
  - Notifying objects of a change (e.g. resize window)

Origins

- In architecture: “A Pattern Language” (1977)
  - Written by architect Christopher Alexander
  - Described patterns in building/construction
  - Inspired others to describe patterns in other fields

- In software engineering: “Design Patterns” (1994)
  - Written by four computer scientists (“Gang of four”)
  - Provides a list of 23 common patterns in software programming with examples in C++ and Smalltalk
  - Highly influential in programming practice and in subsequent programming language design



Why would you use them?

- How to write software that is reliable, maintainable, and easy to understand?
  - Use a modular design
  - Do not repeat yourself
  - Write clean and consistent code

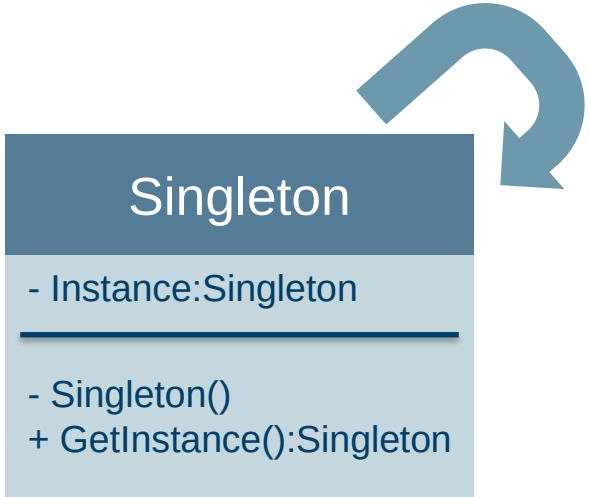
- How to achieve this in practice?
  - Design before you write (e.g. UML diagrams)
  - Apply default solutions to recurring problems
  - Leverage the power of the programming language (i.e.x. OOP)

Types of patterns

- Creational – Use a pattern to create objects
- Structural – Use composition / aggregation to obtain new functionality
- Behavioral – Govern how objects communicate with each other

Singleton Pattern (Creational)

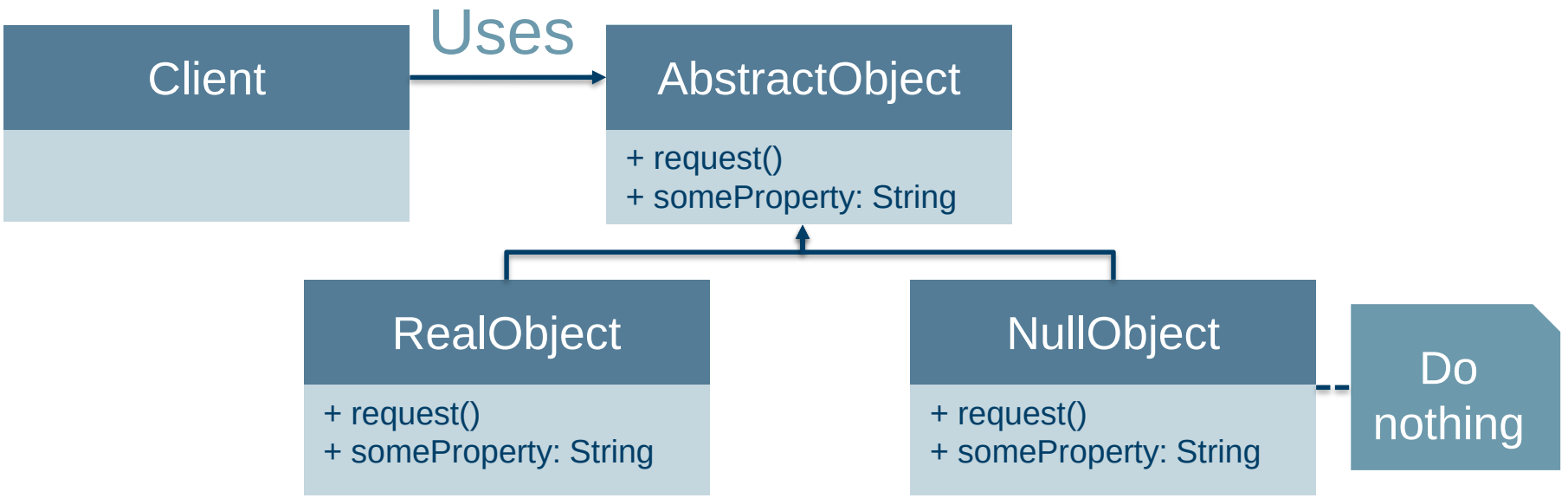
- The singleton pattern restricts the instantiation of a class to one object
- Ensure that one and the same object is used at all times
- Prevent creating an instance of the object the *normal way*



Null object pattern (Behavioral)

- Use a special object to represent empty/non-existence instead of ‘null’
- Create an abstract class that represents the object (no implementation)
  - Create a concrete class that represents a valid object
  - Create a concrete class that represents an invalid object (a “null object”)
- Return a null object whenever a result is undefined or non-existent

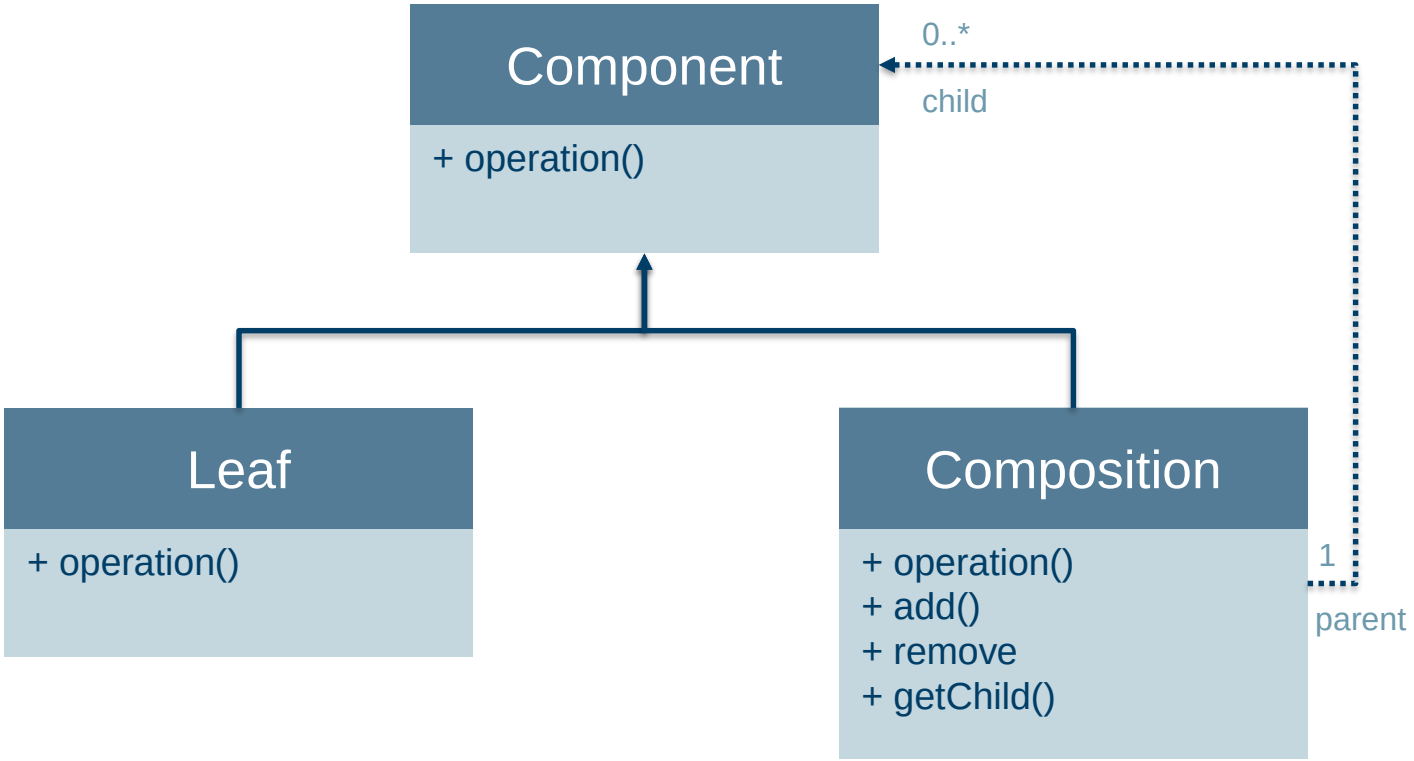
Null object pattern (Behavioral)



Composite pattern (Structural)

- The composite object allows you to treat a collection of objects the same as one object.
- Used for a tree structure of objects where all leaf-nodes have at least one operation to perform
- You define what a component should do and derive the composite and leaf from that

Composite pattern (Structural)



Summary

- Design patterns are templates for solving common design challenges
- Patterns were first described by the “Gang of four” in 1994
- Consistent use of patterns makes your software easier to maintain, more reliable and easier to understand
- Three types of patterns: Creational, structural and behavioral

Practice

Creating some patterns

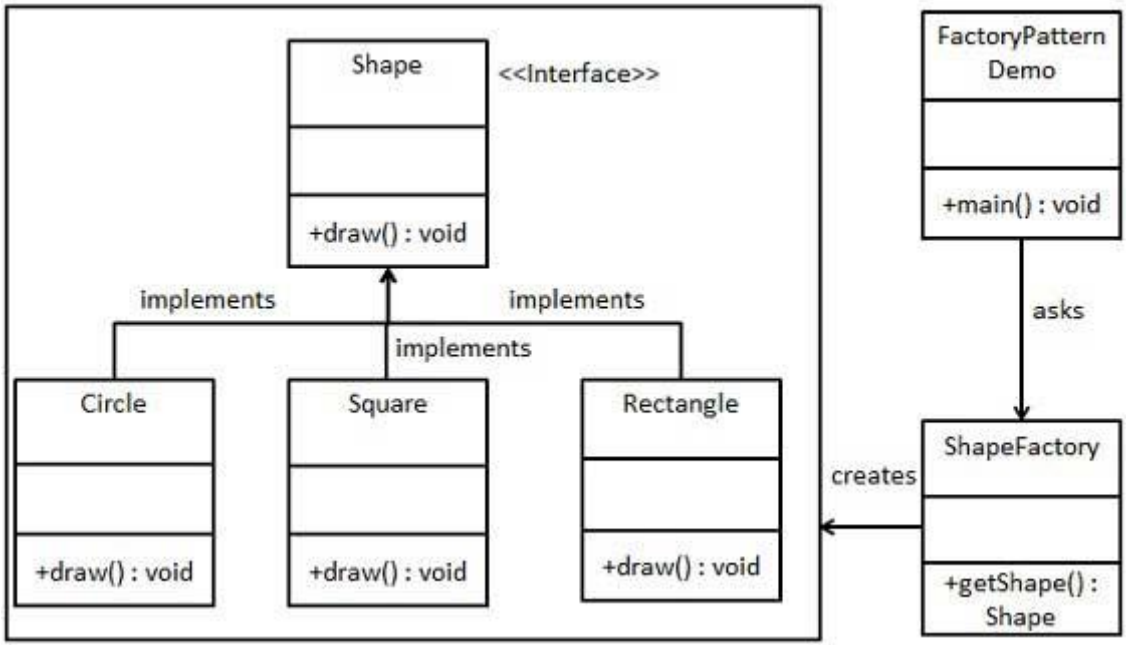
YOUR UTILITY IT PLATFORM

Introduction

- Language of choice: Java
  - Statically typed
  - Native support for interfaces, static methods, null type
  - Most commonly used in tutorials for design patterns

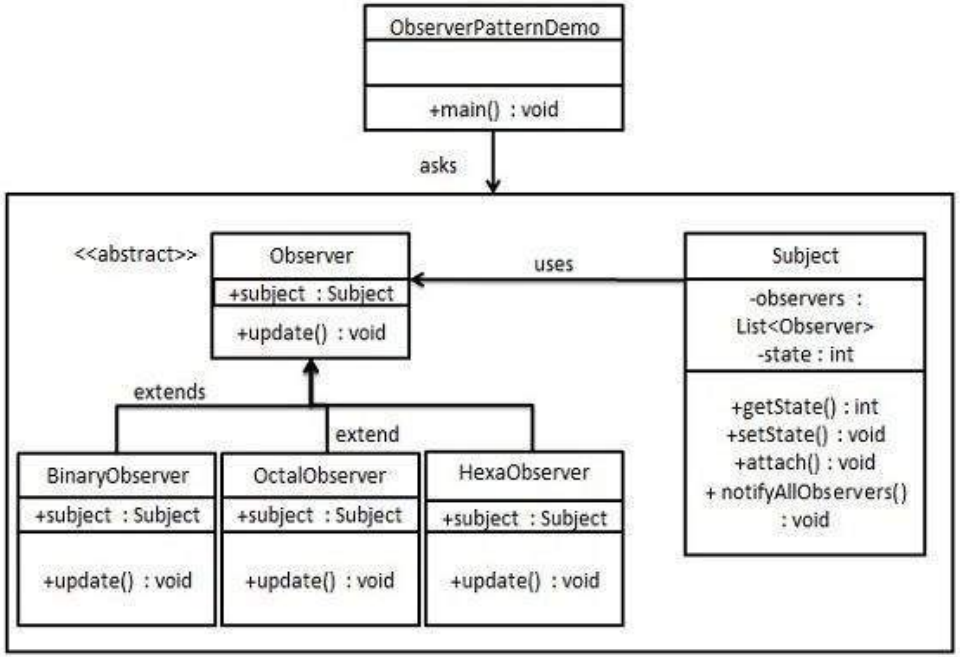
Exercise 1: Creating the Factory pattern (creational)

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



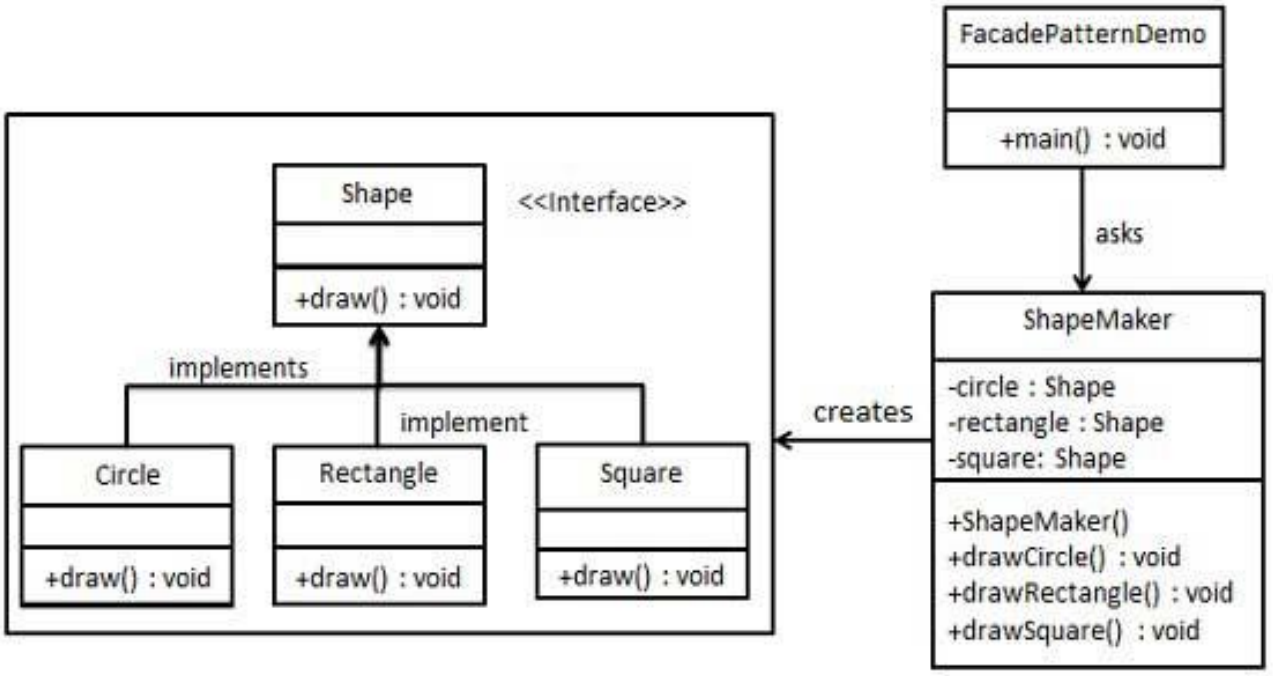
Exercise 2: Creating the Observer pattern (behavioral)

- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.



Exercise 3: Creating the Facade pattern (structural)

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.



Questions?

Contact Joost Meijer at info@itility.nl

YOUR UTILITY IT PLATFORM