fit@hcmus

Object-Oriented Programming

Design Pattern

Tran Duy Hoang

Design patterns

- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Pattern elements

- Name
  - A meaningful pattern identifier.
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

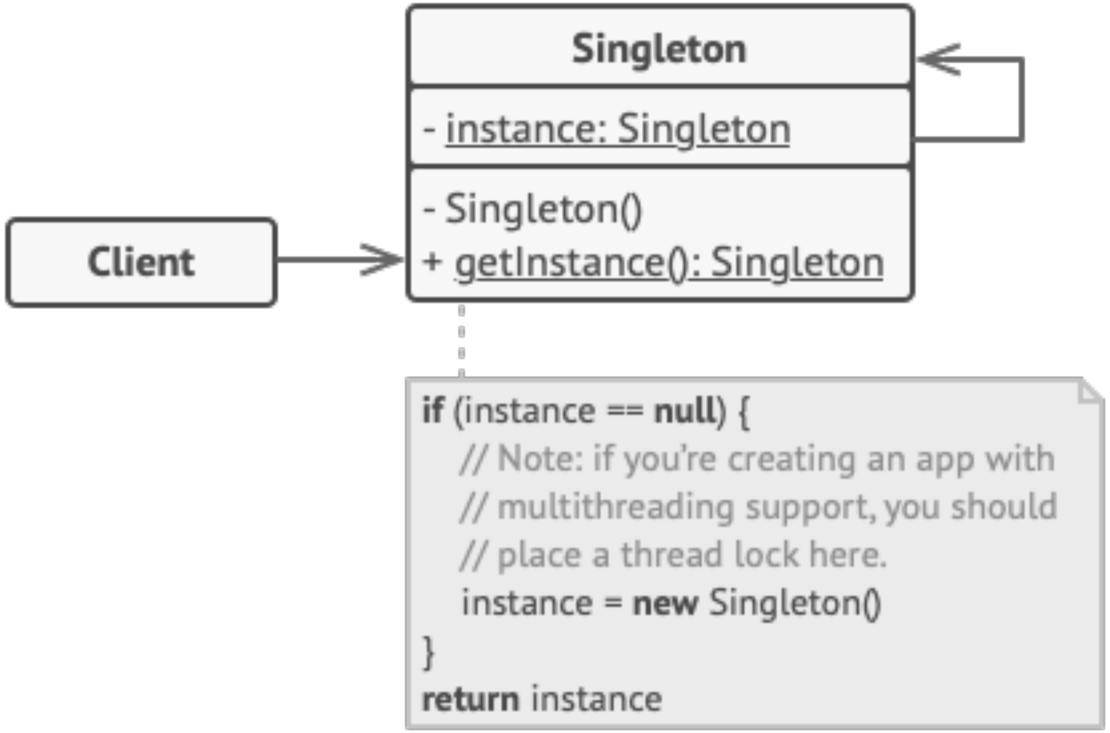
Purpose of Design patterns

- They are tried-and-true solutions
- They are simple to re-use
- They have a strong personalities
- They facilitate communication
- They eliminate the need for code refactoring
- They reduce the codebase's size

Singleton

- Intent
  - Ensure that a class has only one instance, while providing a global access point to this instance
- Problem
  1. Ensure that a class has just a single instance
  2. Provide a global access point to that instance
- Solution
  - Make the default constructor private
  - Create a static creation method that acts as a constructor

Singleton



```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

```
public class Config {

   private static Config instance;

   private Config() {
      // Some initialization code
   }

   public static Config getInstance() {
      if (instance == null) {
         instance = new Config();
      }
      return instance;
   }
}
```

Singleton

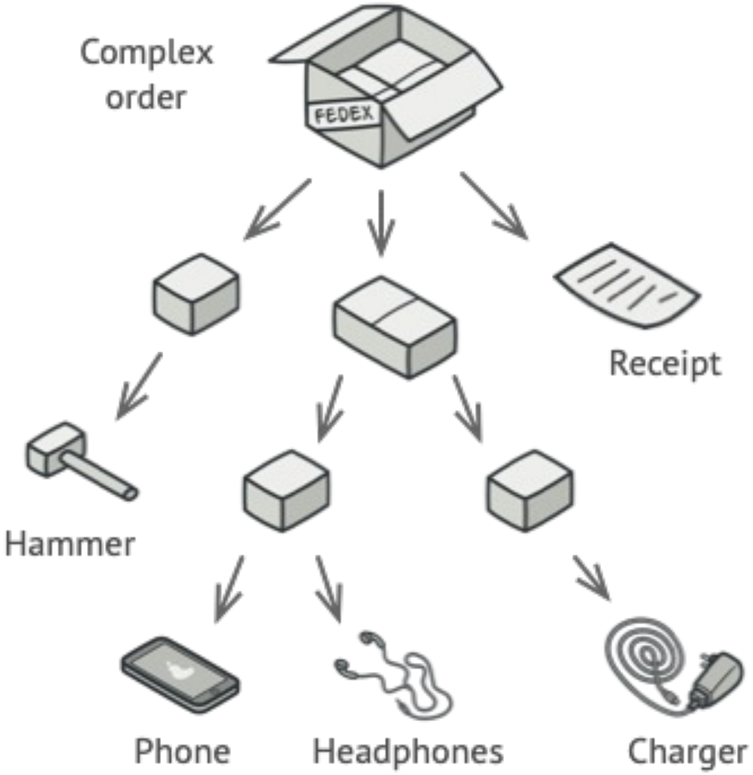
- Applicability
  - When a class in your program should have just a single instance available to all clients
    - E.g., a single database object shared by different parts of the program
  - When you need stricter control over global variables
    - E.g., configuration, logger, catching,…

Singleton

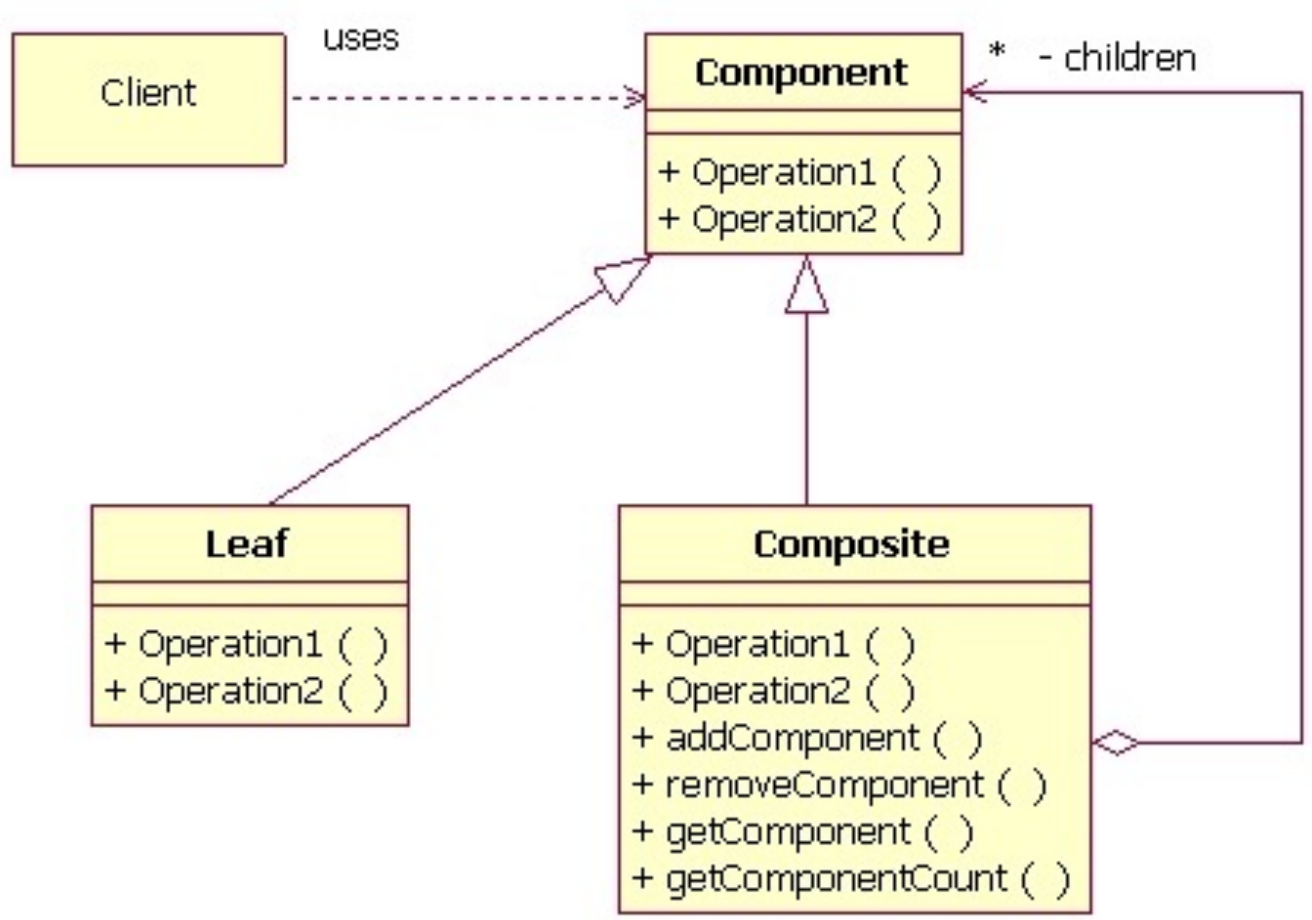
- Pros
  - You can be sure that a class has only a single instance
  - You gain a global access point to that instance
  - The singleton object is initialized only when it’s requested for the first time
- Cons
  - Violates the Single Responsibility Principle
  - Requires special treatment in a multithreaded environment

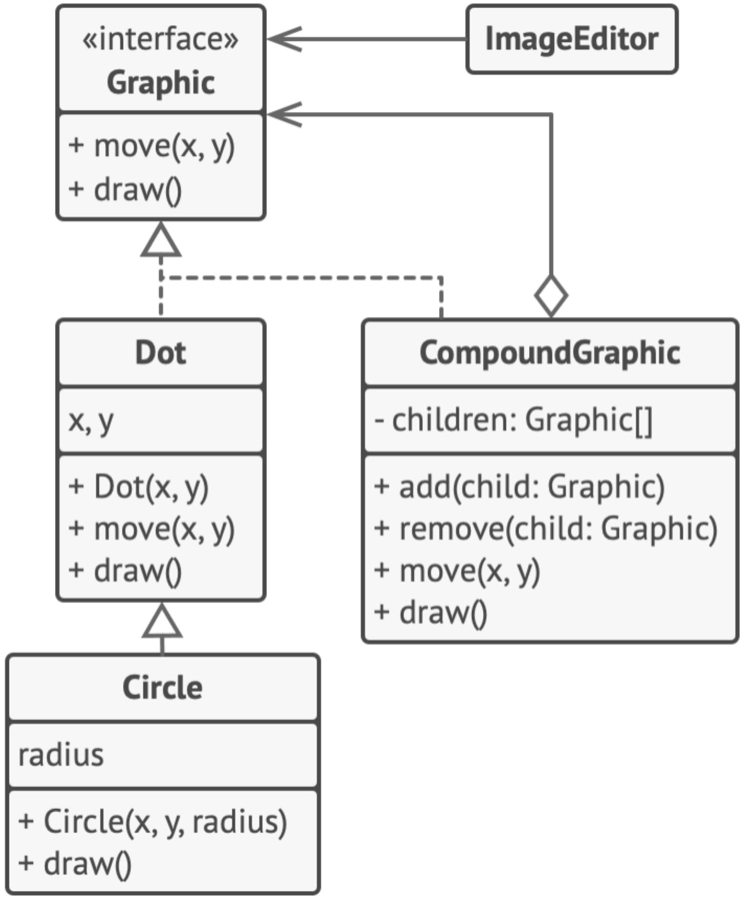
Composite

- Intent
  - Compose objects into tree structures and then work with these structures as if they were individual objects
- Problem
  - Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.
- Solution
  - The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.



Composite





The geometric shapes editor example.

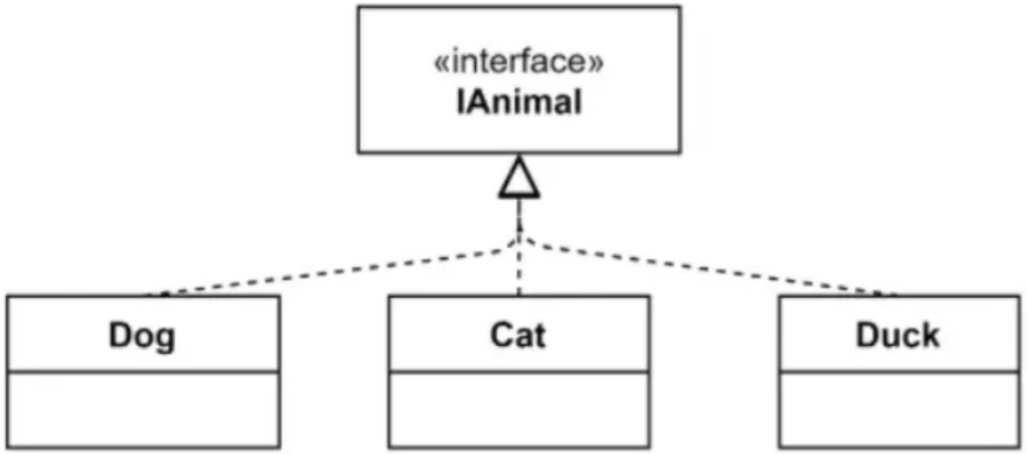
Composite

- Applicability
  - Use the Composite pattern when you have to implement a tree-like object structure.
  - Use the pattern when you want the client code to treat both simple and complex elements uniformly.

Factory Method

- Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Factory Method

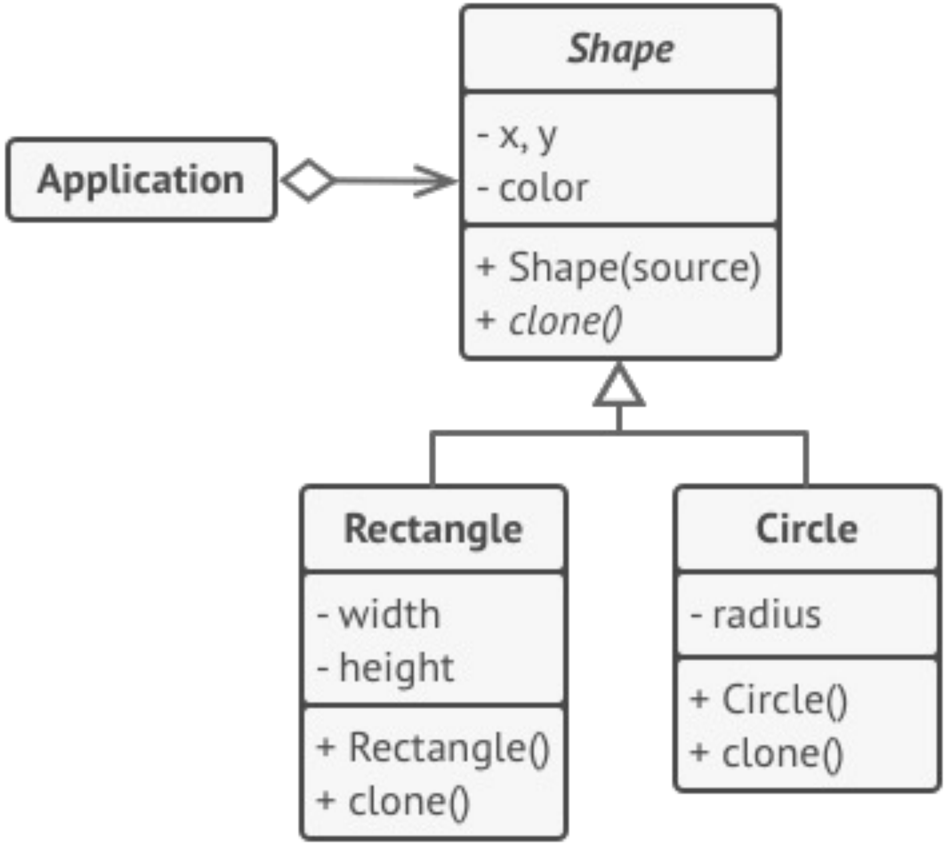


```
public class AnimalFactory {
    public static Animal* CreateAnimal(string animalType)
    {
        Animal* animal;
        switch (animalType) {
            case “Cat”:
                animal = new Cat();
                break;
            case “Dog”:
                animal = new Dog();
                break;
            case “Duck”:
                animal = new Duck();
                break;
        }
        return animal;
    }
}
```

Prototype

- Intent
  - Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
- Problem
  - Say you have an object, and you want to create an exact copy of it. How would you do it?
- Solution
  - The Prototype pattern delegates the cloning process to the actual objects that are being cloned.

Prototype

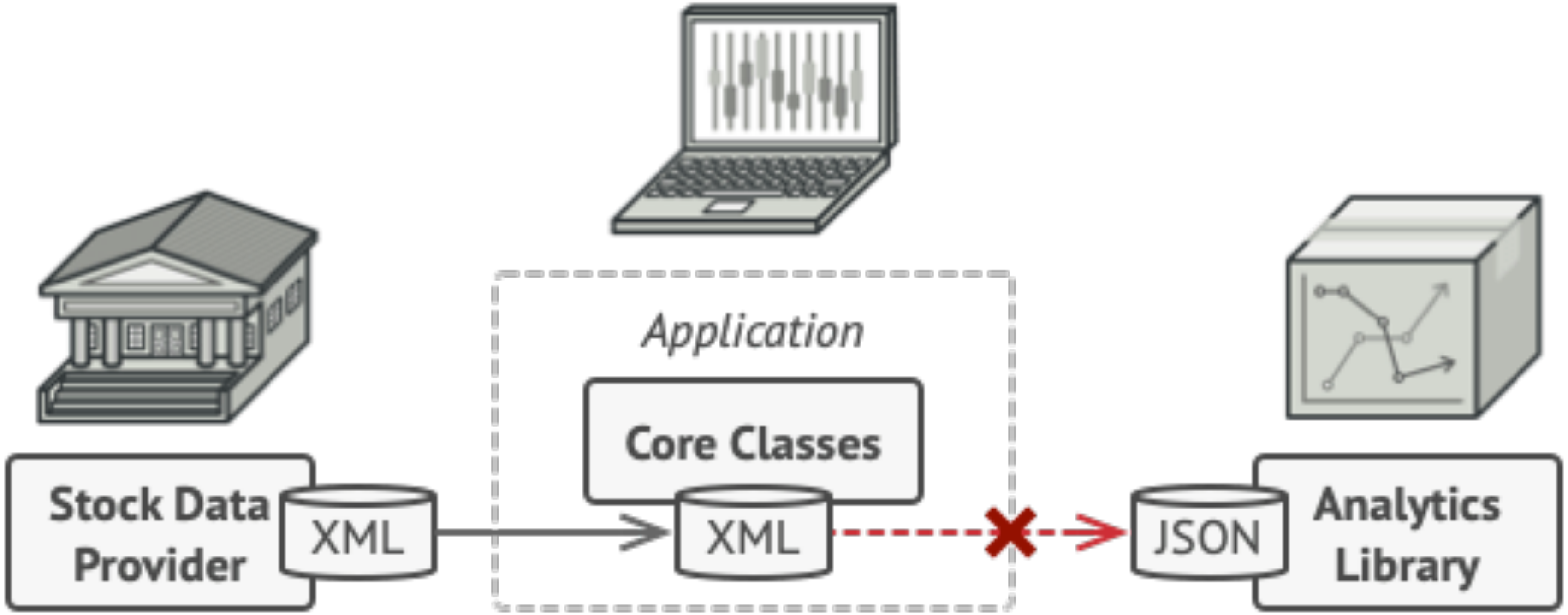


```
public class ShapeRegistry
{
private:
    vector<Shape*> prototypes;
public:
    addPrototype(Shape* shape) {}
    showPrototypes() {} // print id and info of prototypes
    getPrototypeById(int i) {
        return prototypes[i]->clone();
    }
}
```

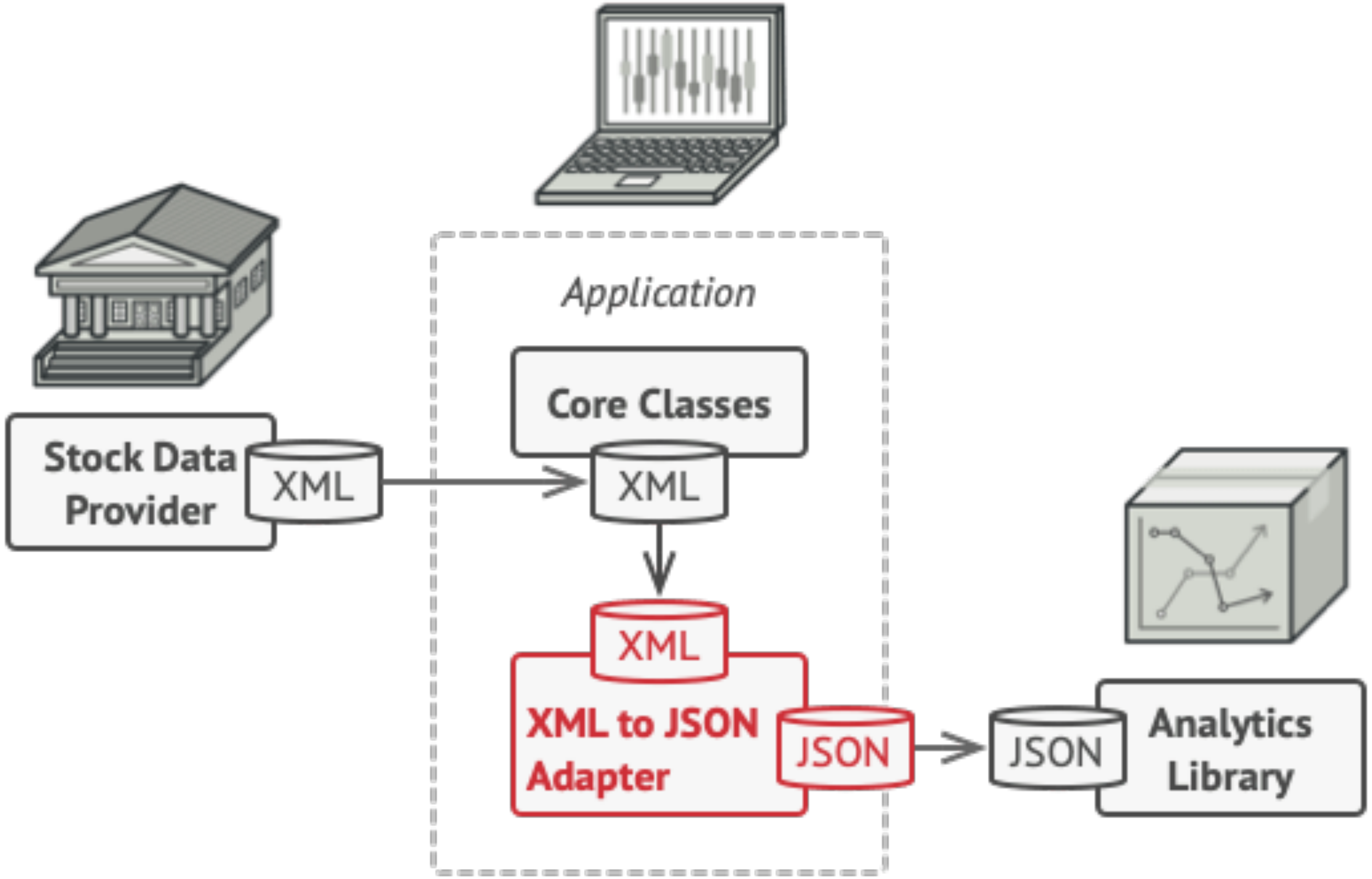
Adapter

- Intent
  - Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
- Problem
  - Imagine that you’re creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- Solution
  - You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.

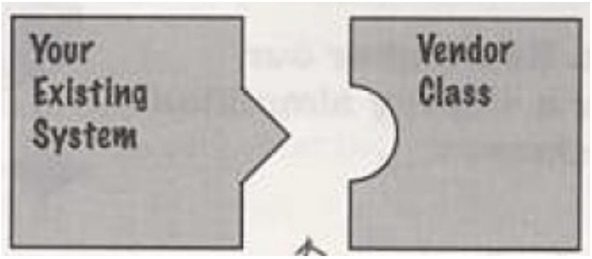
Adapter

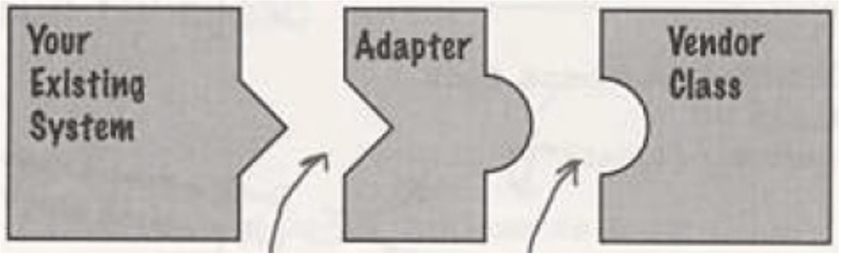

Stock Data Provider
XML
Application
Core Classes
XML
JSON
Analytics Library

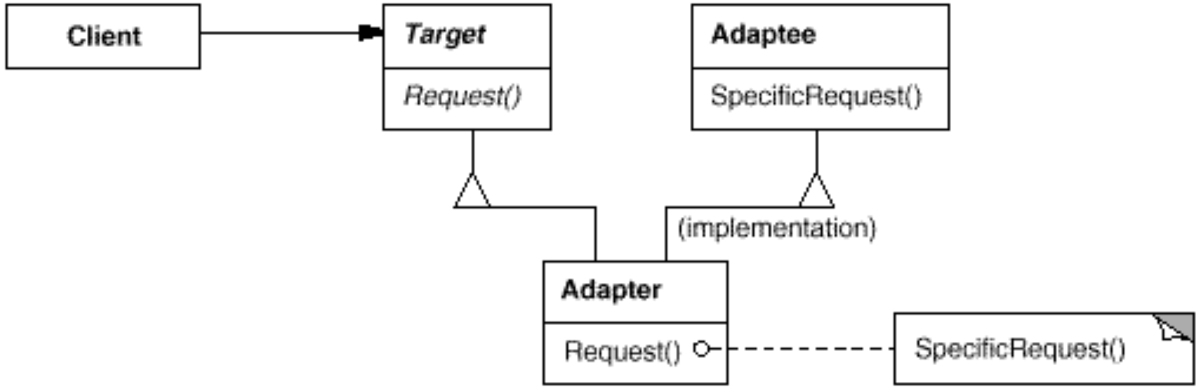
Adapter

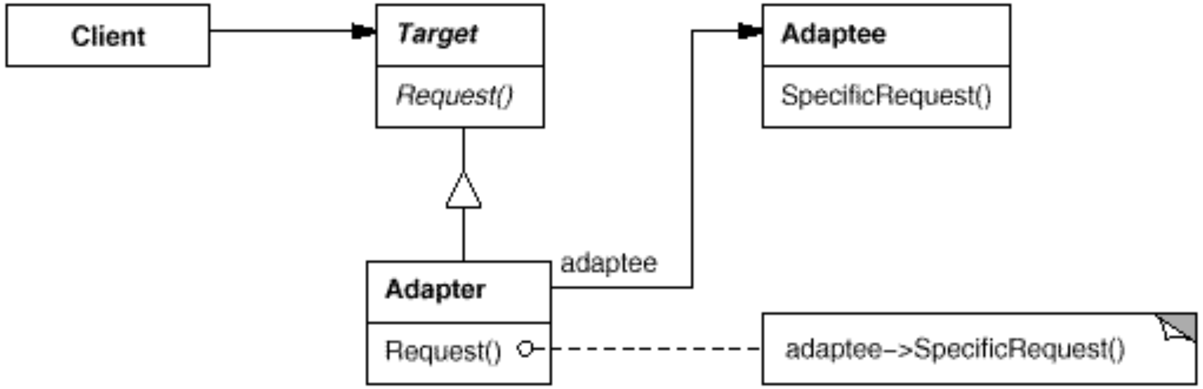

Stock Data Provider
XML
Application
Core Classes
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Adapter


Your Existing System
Vendor Class


Your Existing System
Adapter
Vendor Class


Client
Target
Request()
Adaptee
SpecificRequest()
(implementation)
Adapter
Request()
SpecificRequest()


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

Exercise 8.1

Implement a Logger class using the Singleton pattern. The logger maintains an **internal vector of messages**. It provides a **write()** method to append a new message to the log and a **print()** method to display all stored messages on the screen.

Exercise 8.2

- Build an expression evaluator using a composite tree.
  - Leaf: Number(value)
  - Composite: Add, Subtract, Multiply, Divide (each has left/right children).
  - Requirements:
    - evaluate() returns the numeric result.
    - toInfix() returns a parenthesized string like (3 + (2 * 5)).
    - Include division-by-zero handling.