Design Patterns

CSC207 Winter 2017

Design Patterns

A **design pattern** is a general description of the solution to a well-established problem using an arrangement of classes and objects.

Patterns describe the shape of code rather than the details.

They’re a means of communicating design ideas.

They are not specific to any one programming language.

You’ll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).

Gang of Four

First codified by the Gang of Four in 1995

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Original Gang of Four book described 23 patterns

- More have been added

- Other authors have written books

Book provides an overview of:

- **Name**
- **Problem**: when to use the pattern
  - motivation: sample application scenario
  - applicability: guidelines for when your code needs this pattern
- **Solution**:
  - structure: UML Class Diagram of generic solution
  - participants: description of the basic classes involved in generic solution
  - collaborations: describes the relationships and collaborations among the generic solution participants
  - sample code
- Consequences, Known Uses, Related Patterns, Anti-patterns

Iterator Design Pattern

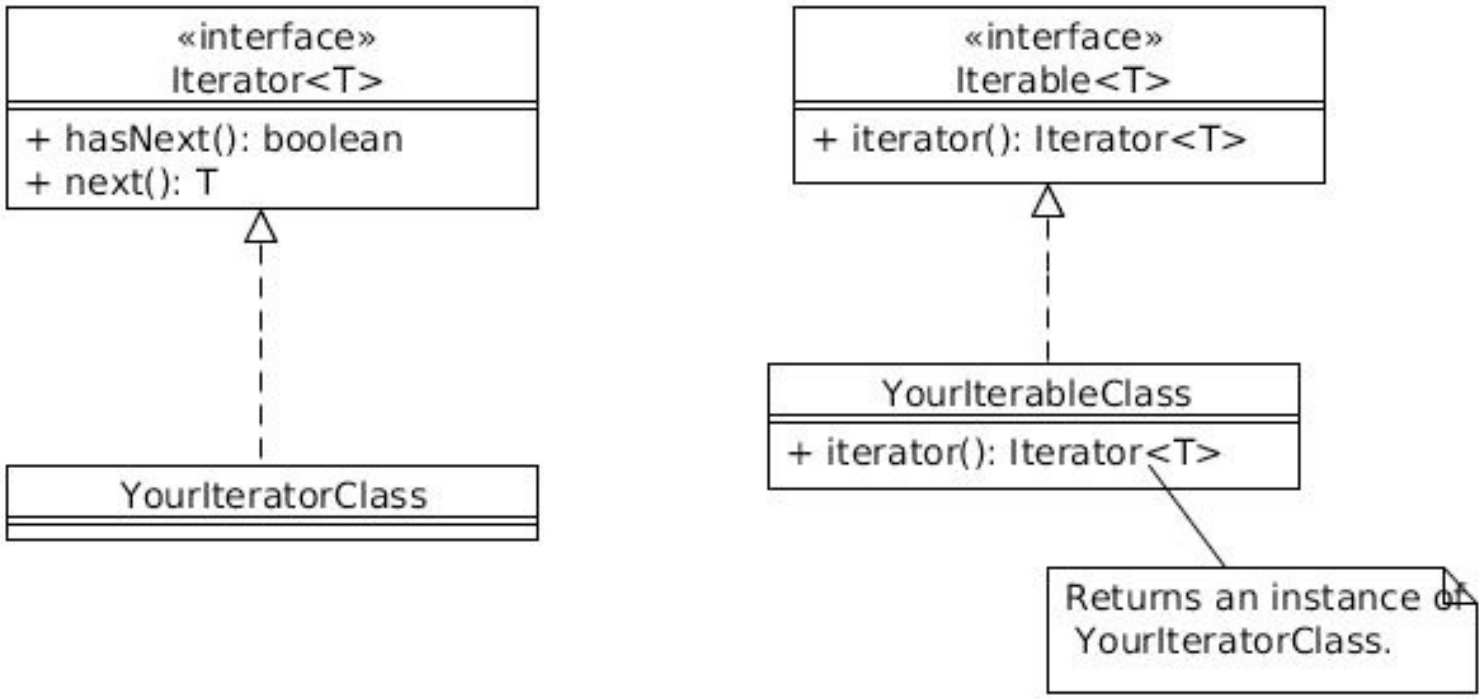
Context

- A container/collection object.

Problem

- Want a way to iterate over the elements of the container.
- Want to have multiple, independent iterators over the elements of the container.
- Do not want to expose the underlying representation: should not reveal how the elements are stored.

Iterator Design Pattern: Java

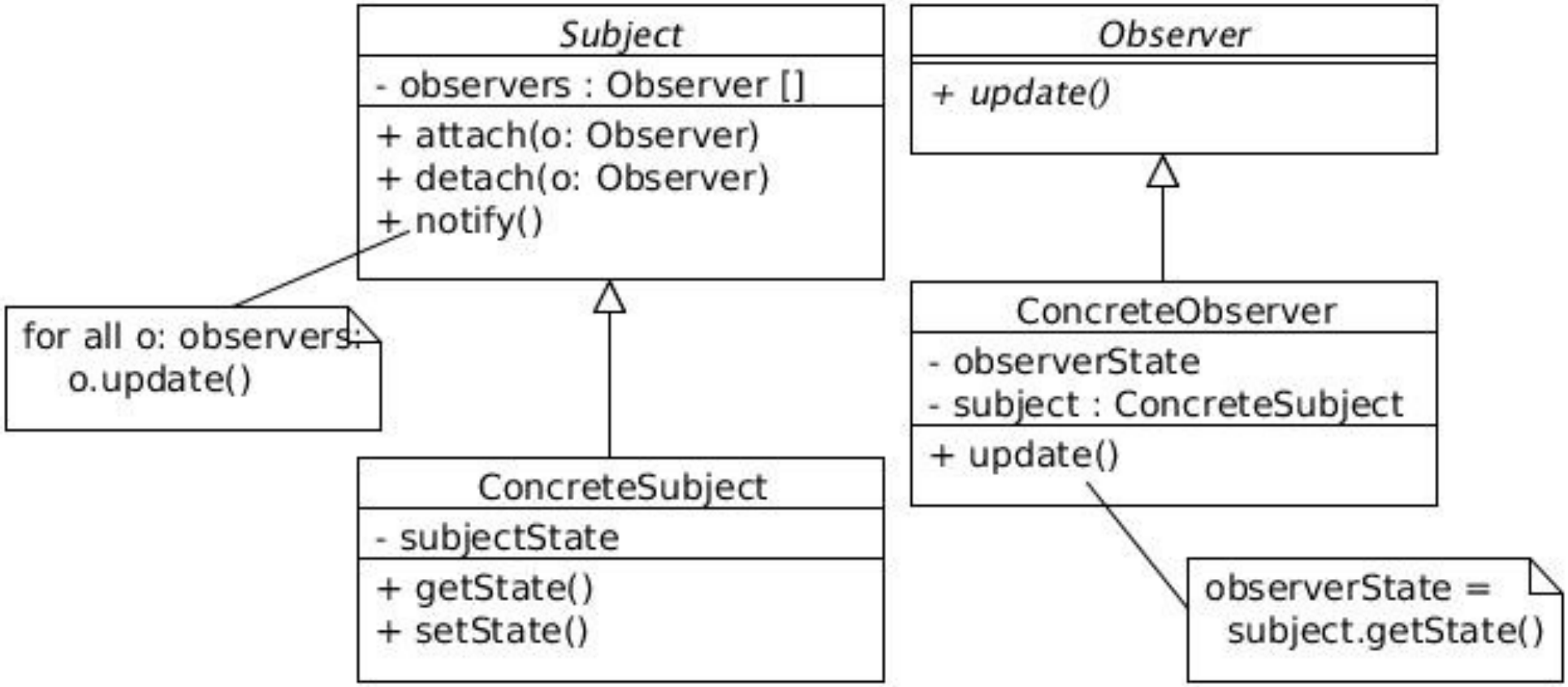


Observer Design Pattern

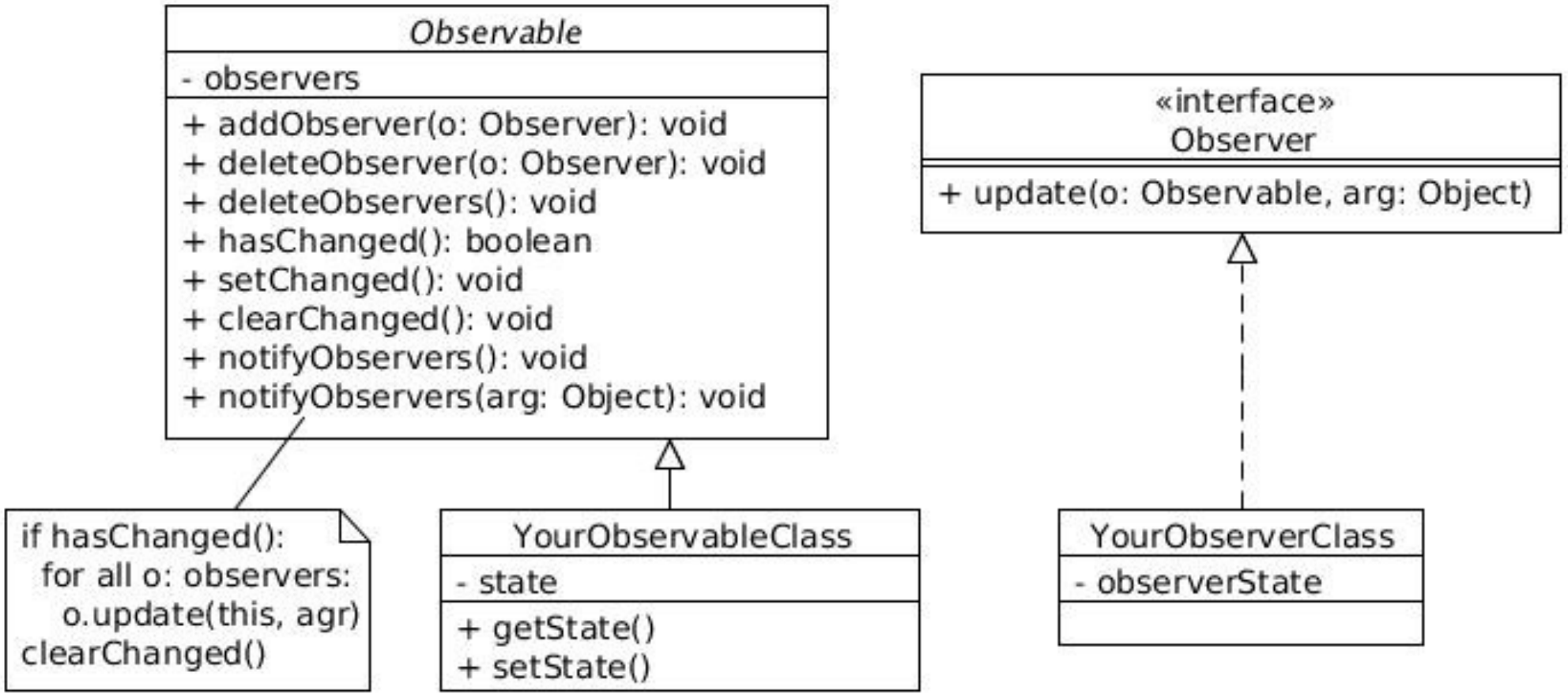
Problem:

- Need to maintain consistency between related objects.
- Two aspects, one dependent on the other.
- An object should be able to notify other objects without making assumptions about who these objects are.

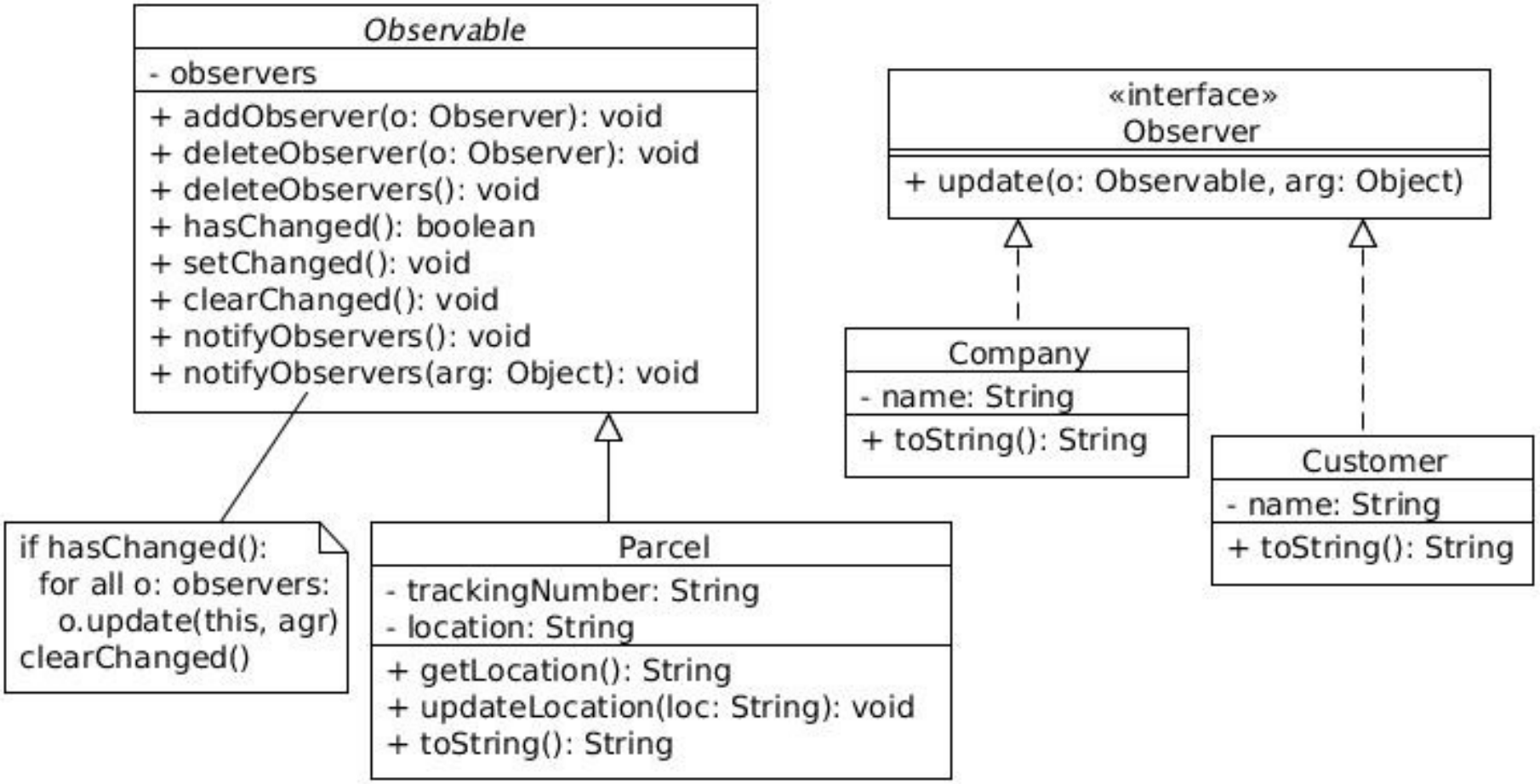
Observer: Standard Solution



Observer: Java Implementation



Observer: Example in Java



Uses of Observer in Java

In reality, people usually implement their own

- Usually can't or don't want to subclass from `Observable`
- Can't have your own class hierarchy and multiple inheritance is not available
- Has been replaced by the Java Delegation Event Model (DEM)
  - Passes event objects instead of update/notify

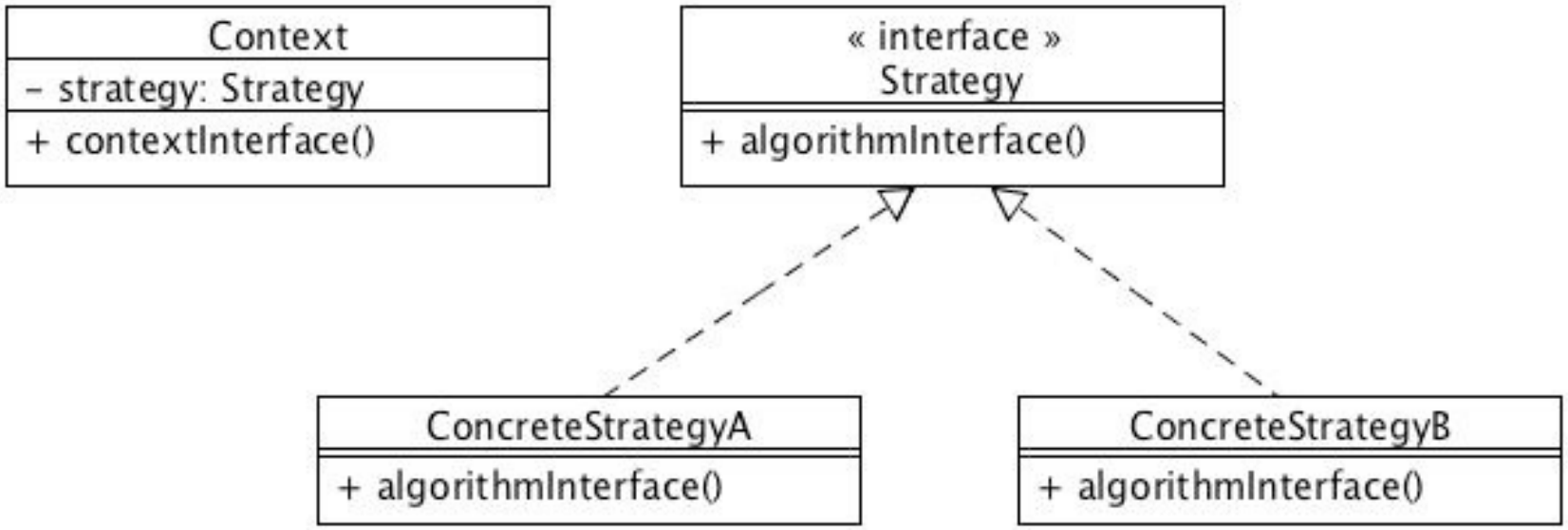
Listener is specific to GUI classes

Strategy Design Pattern

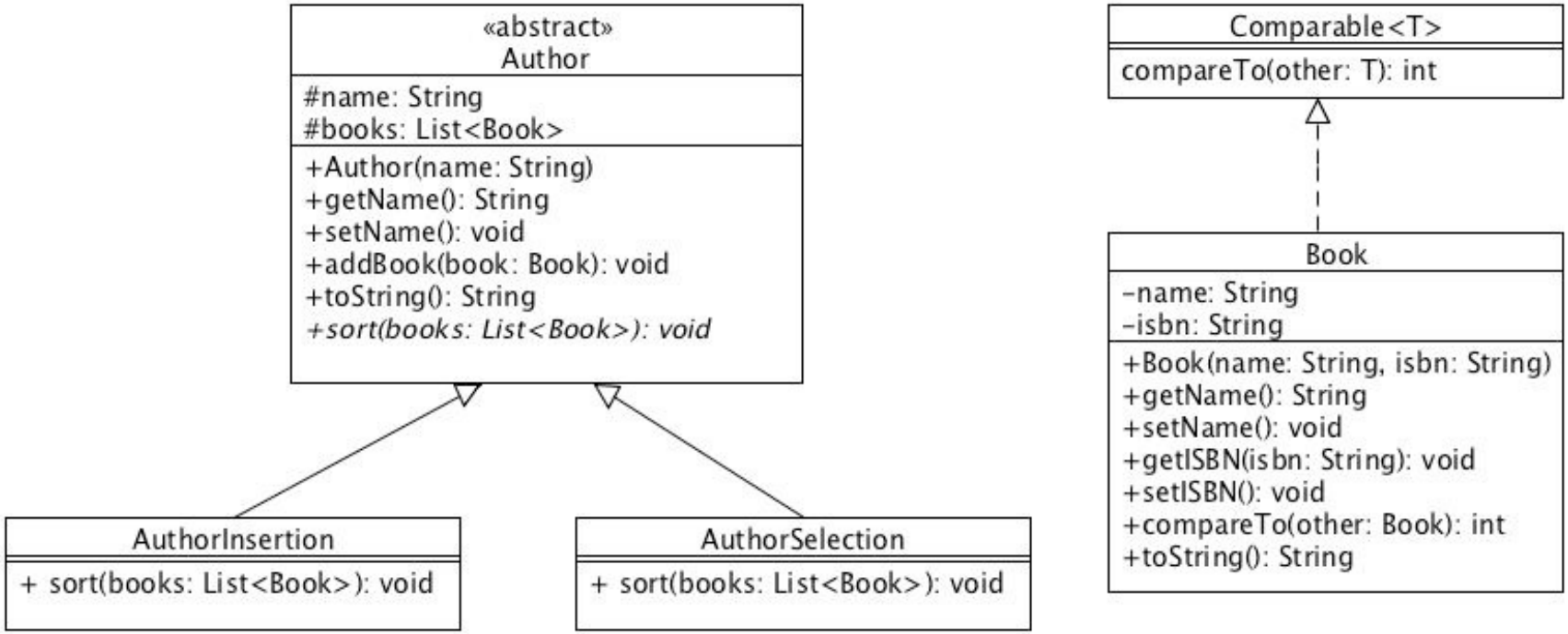
Problem:

- multiple classes that differ only in their behaviour (for example, use different versions of an algorithm)
- but the various algorithms should not be implemented within the class
- want the implementation of the class to be independent of a particular implementation of an algorithm
- the algorithms could be used by other classes, in a different context
- want to **decouple** — separate — the implementation of the class from the implementations of the algorithms

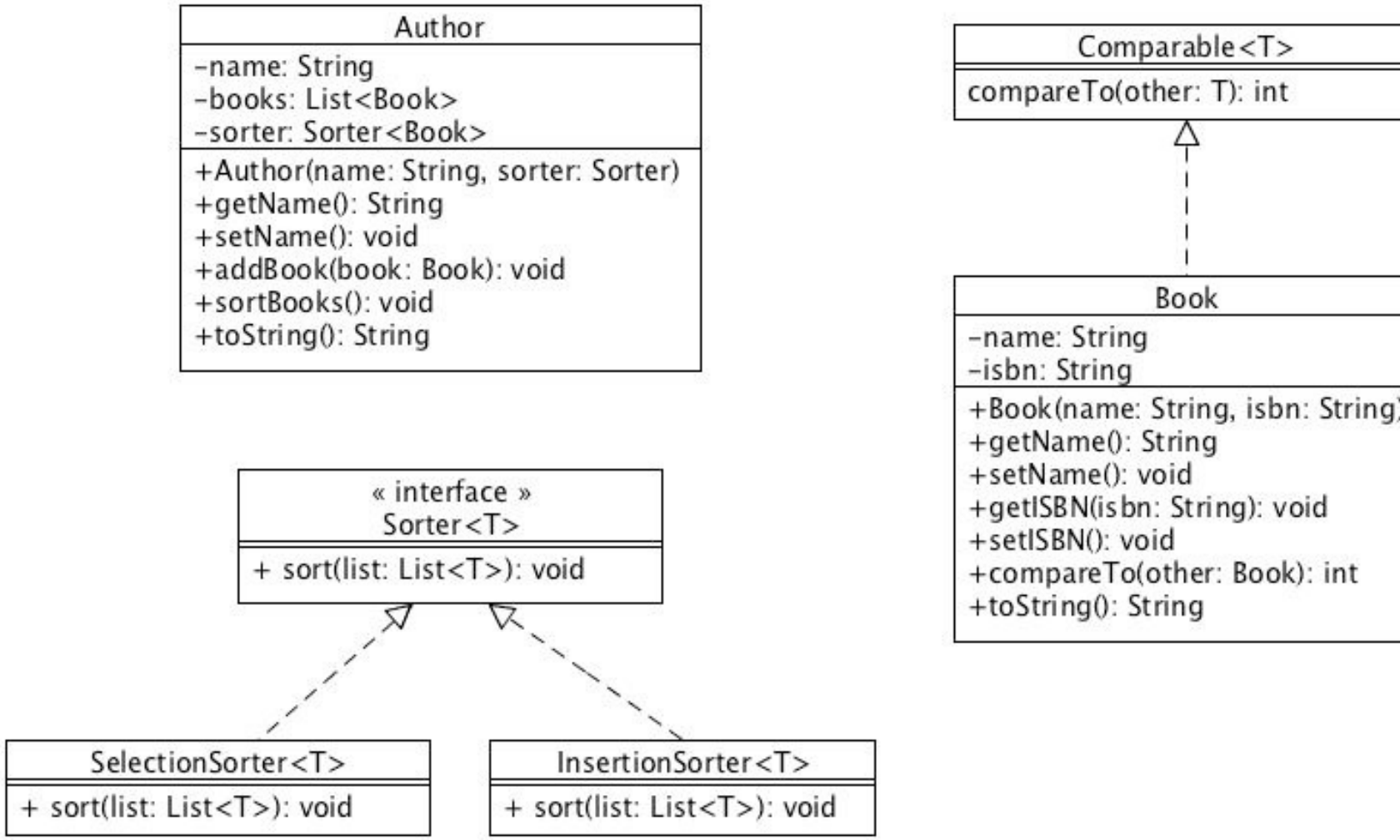
Strategy: Standard Solution



Example: without the Strategy pattern



Example: using the Strategy pattern



Loose coupling, high cohesion

These are two major goals of object-oriented design.

Coupling: the interdependencies between objects. The fewer couplings the better, because that way we can test and modify each piece independently.

Cohesion: how strongly related the parts are inside a class. High cohesion means that a class does one job, and does it well. If a class has low cohesion, then an object has parts that don't relate to each other.

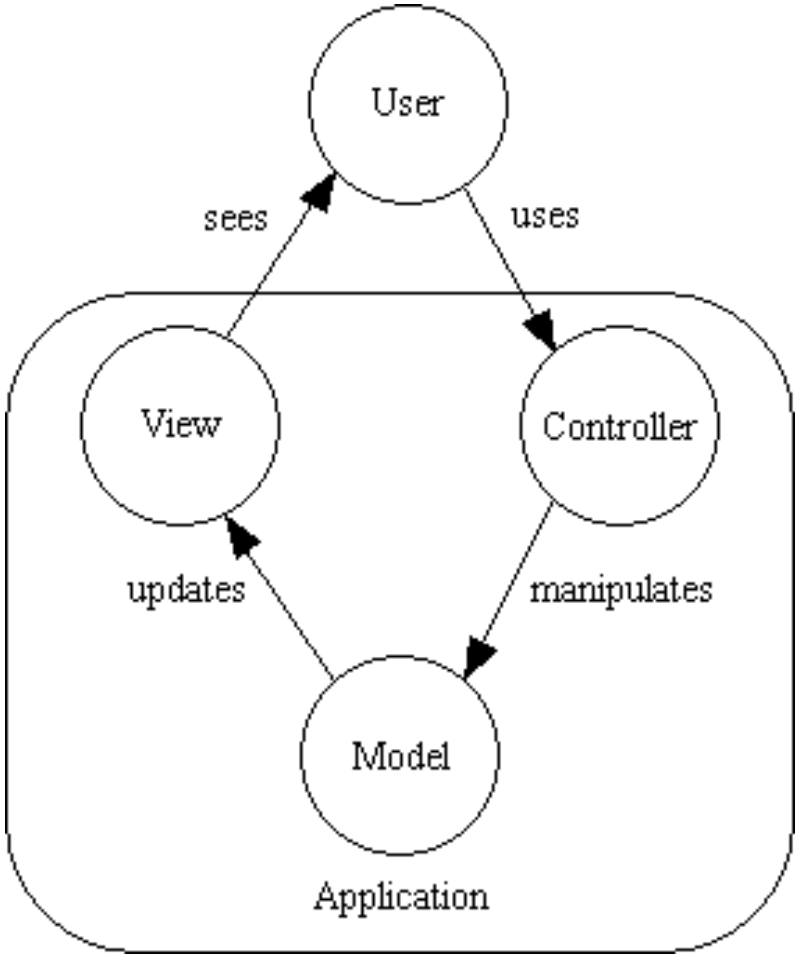
Design patterns are often applied to decrease coupling and increase cohesion.

Model-View-Controller (MVC) Design Pattern

- **Model**: data, rules that govern access to the data, interactions that update the data
    - Often approximates a real-world process
- **View**: presentation of that data that is responsive to changes in the model
    - With a Push Model, the View receives change notifications from the Model
    - With a Pull Model, the View is responsible for calling the model and retrieving the most current data.

- Controller: Interaction between User and View are translated by the Controller into actions for the Model to perform.
    - The controller can call different aspects of the View (close one window and/or open another)
- Graphic User Interfaces (GUI) often comprise the View, where the Controller is the non-visual front end (e.g., telling buttons what to do when they are clicked), and the Model is the data which the GUI displays and with which the User interacts via the Controller.

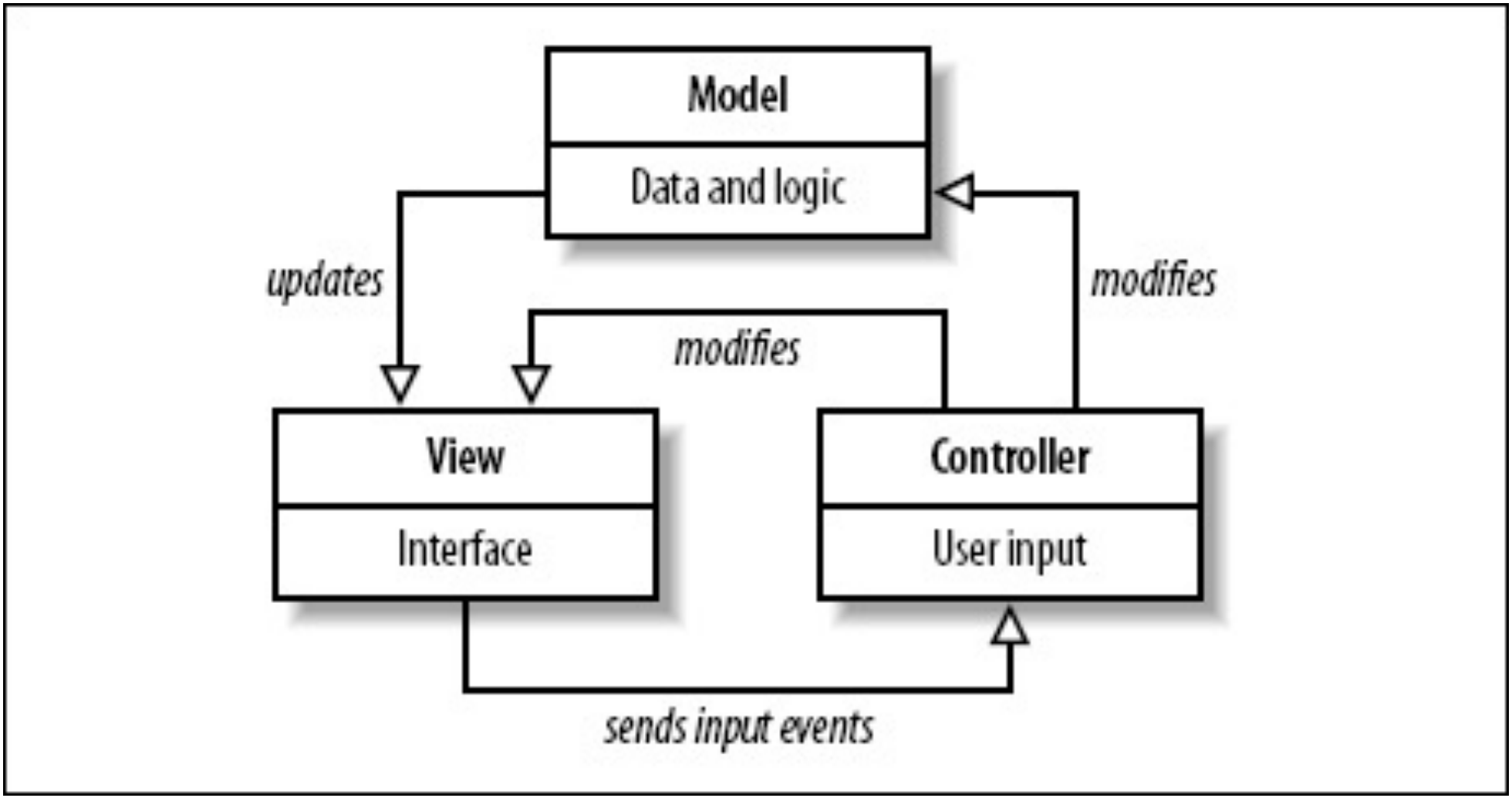
MVC Pattern: General Structure



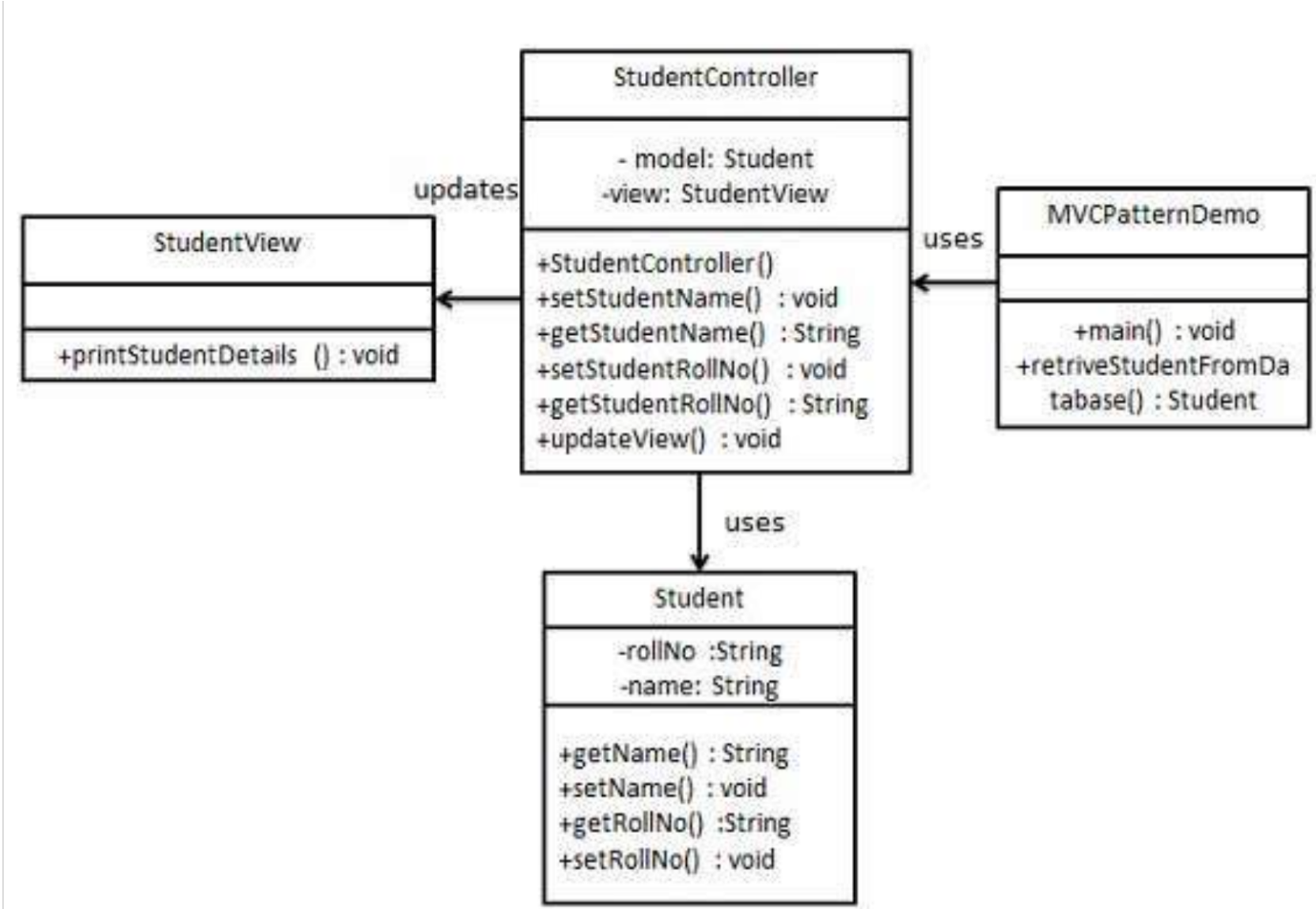
From:

https://www.tonymarston.net/php-mysql/model-view-controller-01.png

MVC: What's happening inside?



MVC: A Solution in Java



Code for this example can be found here:

https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

Dependency in Object Oriented Programming

- A "dependency" relationship between two classes (also called "using" relationship) means that any change to the second class will change the functionality of the first.
- For example: class `AddressBook` depends on class `Contact` because `AddressBook` contains instances of `Contact`.
- Other examples of dependencies: loggers, handlers, listeners

Dependency Injection Design Pattern

Problem:

- We are writing a class, and we need to assign values to the instance variables, but we don't want to hard-code the types of the values.

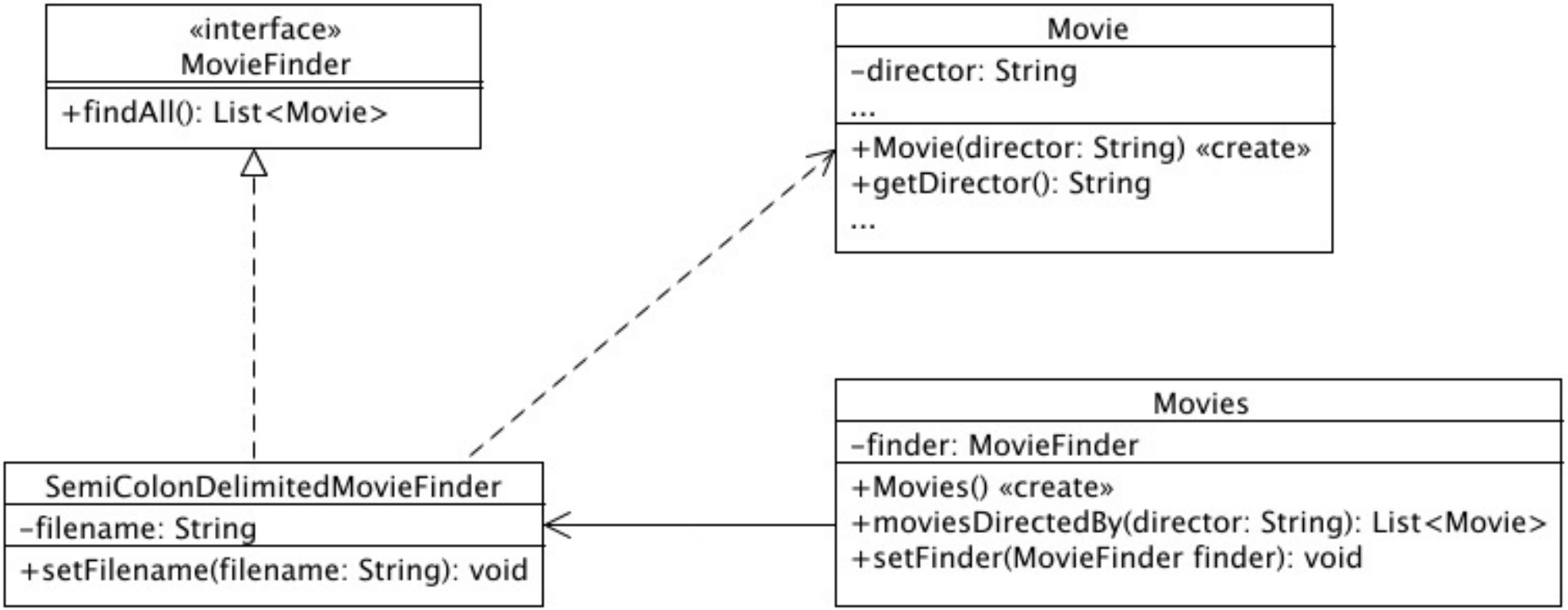
Dependencies in Java

- Using the `new` operator inside the first class can create an instance of a second class that cannot be used nor tested independently. This is called a "hard dependency".
- We often want to use and test the second class independently from the first and avoid hard dependencies. For example:

```
public class MyClass {
    private Student student;

    public MyClass(Student student){
          this.student = student;
    }
}
```

Dependency Injection Example



- Notice how the constructor for class `Movies` takes an argument of type `MovieFinder`.
- Any change to the `Movie` class will affect the `MovieFinder`, but not `Movies` directly.
- A dependency has been created between `Movie` and `MovieFinder`.

Code for this example can be found here:

http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dependency_injection.html

Dependency Injection Pattern Exercise

- Consider our previous example of class `Person` and its subclass `Student`.
- Create a class called `Email` that contains instance variables of type String called: `to`, `from`, `cc`, `subject`, and `message`. It should also contain `printToScreen()` and `printToFile()` methods.
- Class `Person` should have the ability to send an `Email` object to another `Person` object. But that will require our first instance of `Person` to obtain the second `Person` object's name or email address.

- Without Dependency Injection:
    - The sender object will have to use reflection to gather their own name/email address, and somehow obtain the receiver object's name/email address, etc.
    - The code for creating and sending an email will have to be inside class `Person`.
    - Any change to the `Email` class will impact the code in class `Person`.

- With Dependency Injection:
    - We can create a class called `EmailService` that gathers the information required to compose and send `Email` objects.
    - The constructor for `EmailService` can take in the entire `Person` sender object and `Person` receiver object and retrieve the values of their respective `name` and `email` fields, in order to create a new instance of `Email`.
    - Any change to the `Email` class will be contained by the `EmailService` and shielded from class `Person`.

Factory Design Pattern

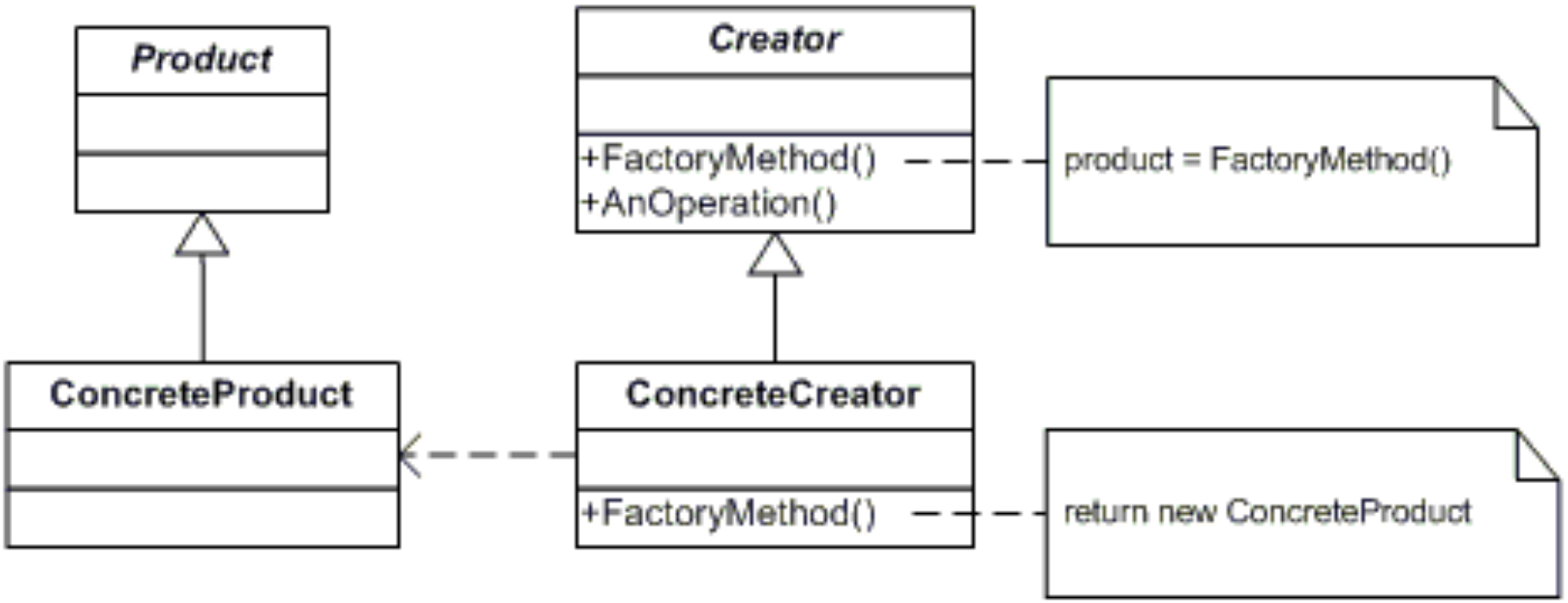
Problem:

- A class cannot anticipate the class of the object it must create.
- A class wants its subclasses to specify the object it creates.
- Classes delegate responsibility to one of several helper subclasses and you want to localize knowledge of which helper subclass is the delegate.

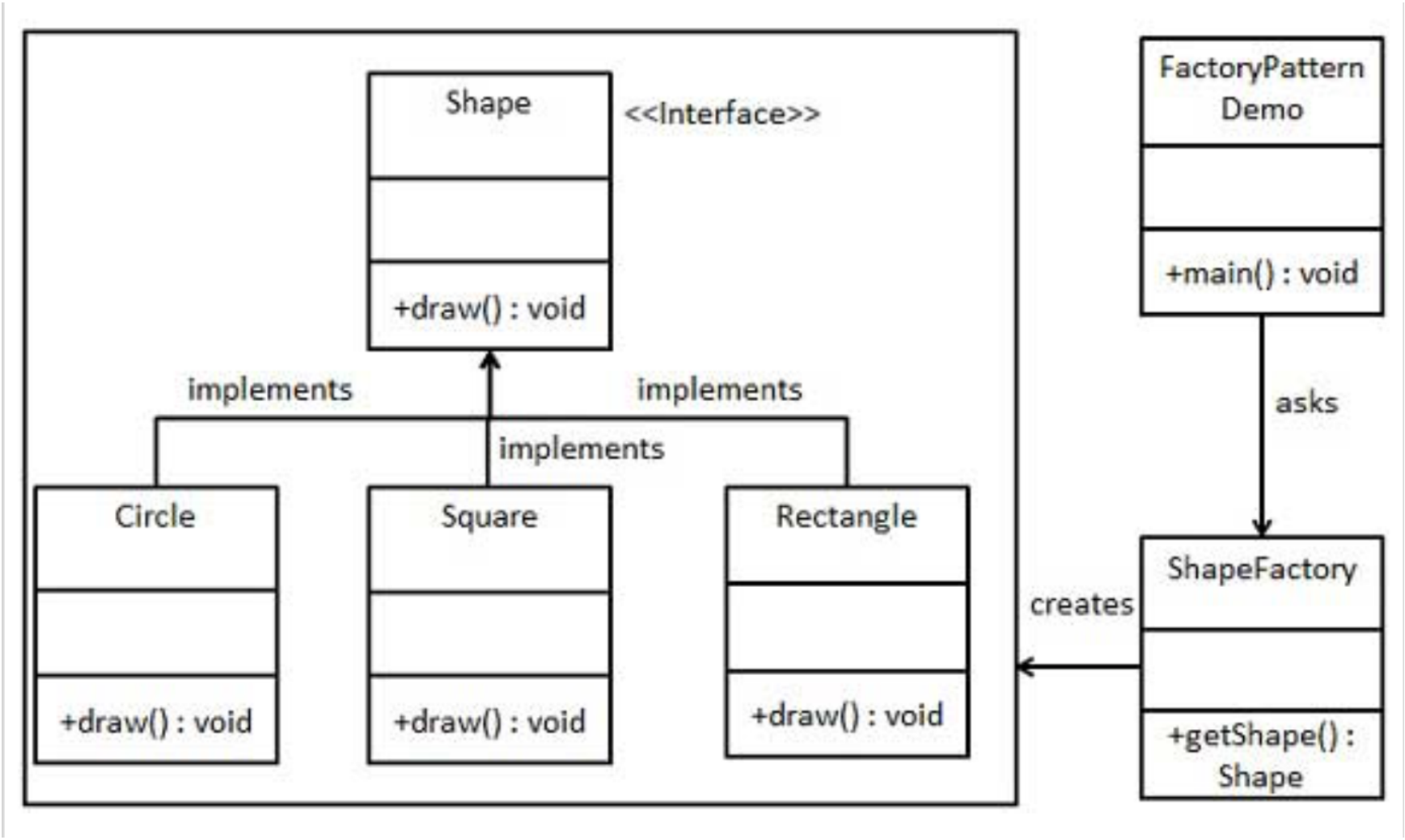
When to use Factory Method Design Pattern

- One class wants to interact with many possible related objects.
- We want to obscure the creation process for these related objects.
- At a later date, we might want to change the types of the objects we are creating.

Factory Method: A Standard Solution



Factory Pattern: An example

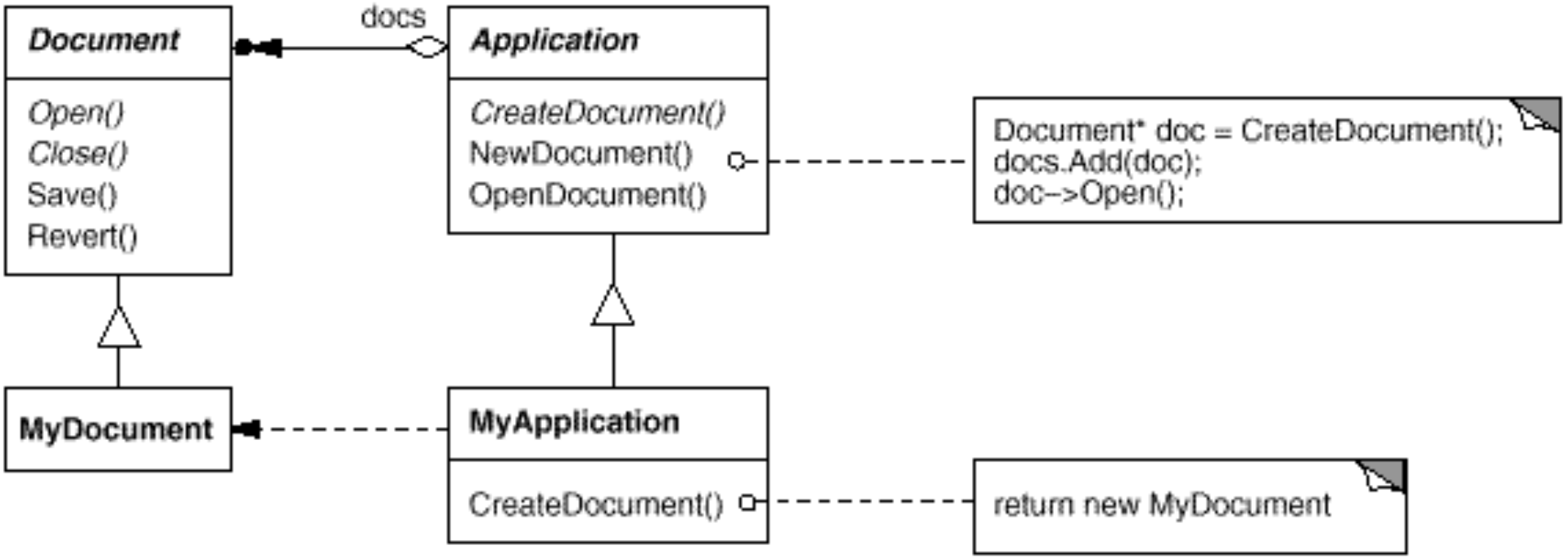


Code for this example can be found here:

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

We can easily create new subclasses of `Shape` and use the `ShapeFactory` to instantiate them without changing the rest of the code.

Factory Pattern: Another Example



For implementation of the Document Example, see:

http://stg-tud.github.io/eise/WS15-SE-18-Factory_Method_and_Abstract_Factory_Design_Pattern.pdf