[Handout for L9P1] Déjà vu: Using patterns to solve recurring problems

Patterns

One major theme in problem solving is to identify commonalities in similar problems, and search for a general solution. Once a general solution is found, all future occurrences of the problem can be resolved with less effort and better outcomes. For example, the stack data structure can be used to solve problems that exhibit LIFO (Last in first out) ordering, Dijkstra's algorithm can help to identify the shortest path from a single node, etc.

In software development, there are certain problems which frequently crop up. The problems could be about the software architecture e.g. what is the best architecture for a given type of system, or about the interaction between classes e.g. how do I lower the coupling of these classes, or similar. After years of addressing such problems, better solutions are discovered and refined over time. These solutions are collectively known as *patterns*, a term popularized by the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by the so called "Gang of Four" (GoF) Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Instead of listing a large number of patterns in this handout, here we attempt to guide you on <u>how to understand them</u>. Furthermore, we focus more in patterns in the area of software design, i.e. design patterns. Since patterns are described with a specific format, you should be able to pick up more of them after reading through the few examples given here. The common format to describe a pattern consists of the following information:

- **Context**: The situation or scenario where the design problem is encountered.
- **Problem**: The main difficulty to be resolved in the problem. Criteria for a good solution are also identified to evaluate solutions.
- **Solution:** Core of the solution is described. It is important to note that the solution presented only includes the most general constraints, which may need further refinement for a specific context.
- **Anti-patterns** (optional)**:** Commonly used solutions, which are usually incorrect and/or inferior to the Design Pattern.
- **Consequences** (optional): What are the pros and cons of applying the pattern
- **Other useful information** (optional): Code examples, known uses, other related patterns, etc.

Abstraction-occurrence pattern

Context

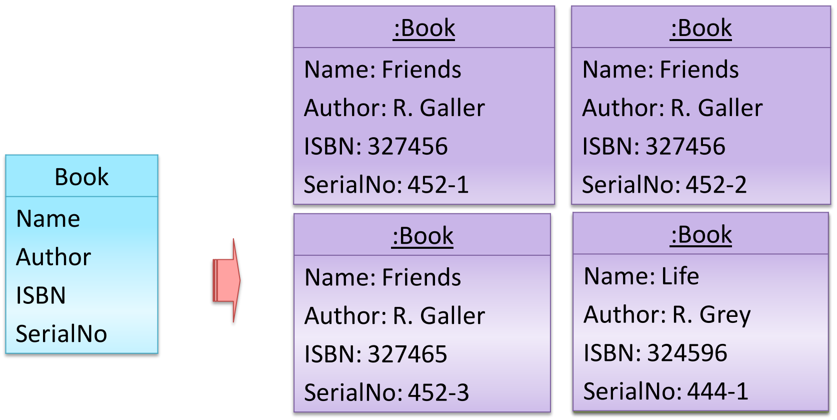
There is a group of similar entities that appears to be 'occurrences' (or 'copies') of the same thing, sharing lots of common information, but also differing in significant way.

For example, in a library, there can be multiple copies of same book title. Each copy shares common information like book title, author, ISBN etc. However, there are also significant differences like purchase date and barcode number (assumed to be unique for each copy of the book). Other examples include episodes of the same TV series and stock items of the same product model (e.g. TV sets of the same model).

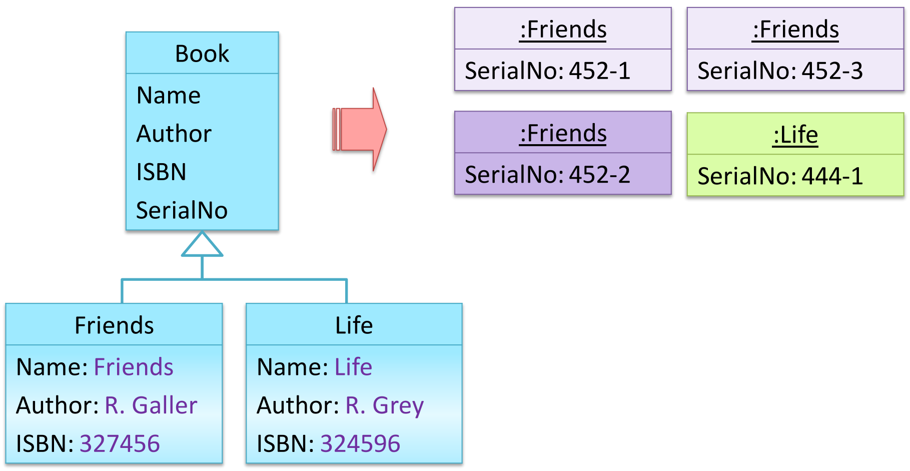
Problem

Representing those objects mentioned previously as a single class would be problematic (see Anti-pattern description below). We need a better representation to represent such instances, which should avoid duplicating the common information. Without duplicated information, it is also easier to avoid inconsistency should these common information changes.

Anti-patterns

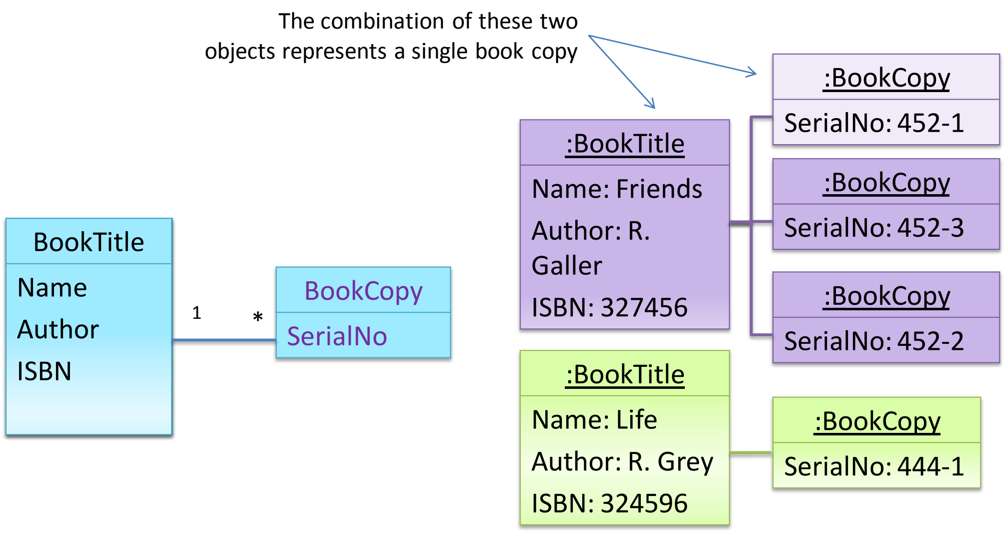


Let us take the problem of representing books in a library. Assume there could be multiple copies of the same title, bearing different serial numbers but the same ISBN number. The above diagram shows an inferior or incorrect design for this problem. It requires common information to be duplicated by all instances. This will not only waste precious storage space, but also creates a consistency problem. Imagine that after creating several copies of the same title, the librarian realized that the author name is spelt wrongly. To correct this mistake, the system needs to go through every copy of the same title to make the correction. Also, if a new copy of the title is added later on, the librarian has to make sure that all information entered is the same as those previous copies to avoid inconsistency.

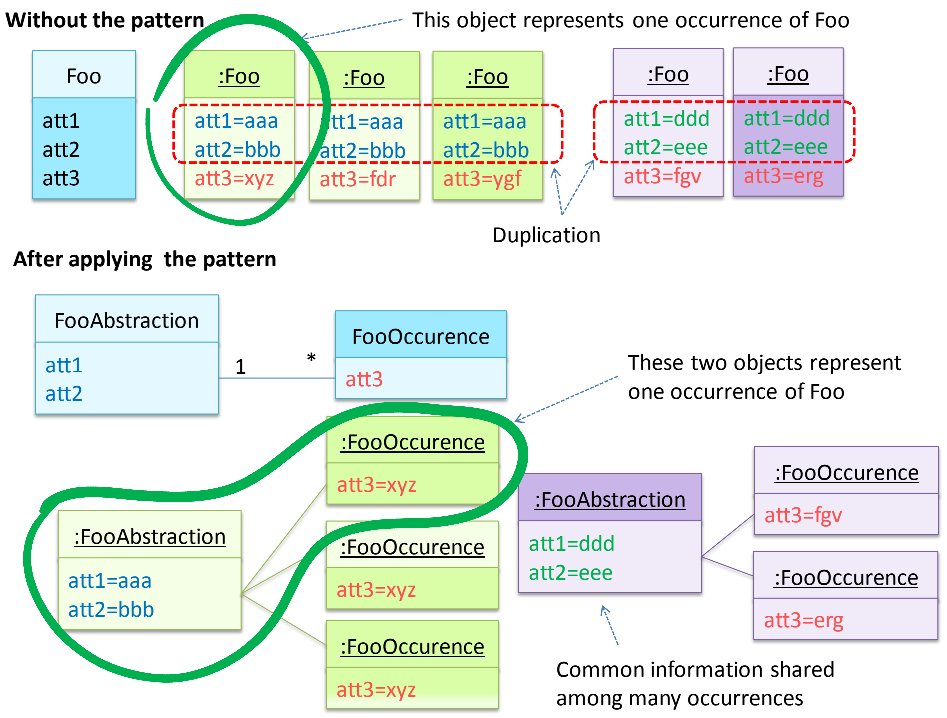


The design above segregates the common and unique information into a class hierarchy. Each book title is represented by a separate class and common data (i.e. Name, Author, ISBN) are hard-coded in the class itself. This solution is worse than the first because each book title will have to be represented as a class. This could results in thousands of classes. Every time the library buys new books, the source code of the system will have to be updated with new classes.

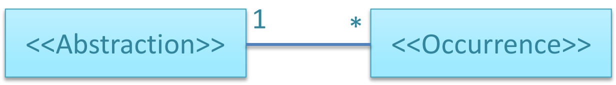
Solution
The solution is to let a book copy be represented by two objects instead of one, as given below.



In this solution, the common and unique information are separated into two classes to avoid duplication. Given below is another example that contrasts situations *before* and *after* applying the pattern. An *association* is be used to connect the two classes instead.



The general idea can be found in the following class diagram:



The <<Abstraction>> class should hold all common information, and the unique information should be kept by the <<Occurrence>> class. Note that 'Abstraction' and 'Occurrence' are not class names, but roles played by each class. You can think of this diagram as a *meta-model* (i.e. a 'model of a model') of the BookTitle-BookCopy class diagram given above.

Singleton pattern

Context

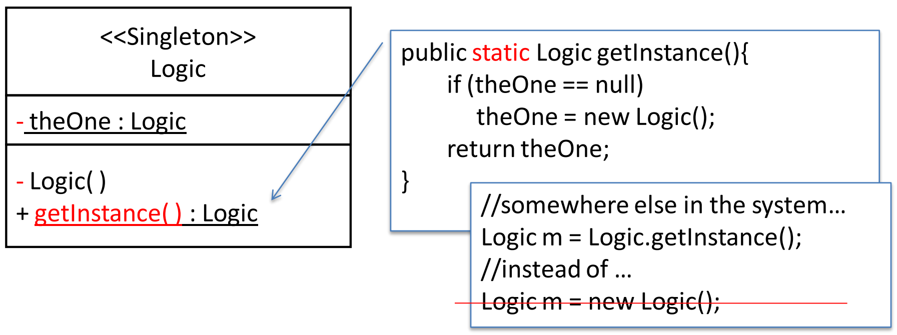
In most systems, it is common to find certain classes for which we wish to restrict the number of instantiated objects to just one (e.g. the main controller class of the system). These single instances are commonly known as *singleton*.

Problem

We want to ensure that it is *impossible* to instantiate more than one object from a singleton class and the single instance is easily shared among those who need it.

Solution

The key point of the solution is to realize that *constructor* of the singleton class cannot be *public*. As *public constructor* will allow others to instantiate the class at will. A *private constructor* should be used instead. In addition, we can provide a public method to access the *single instance*. This solution is described below.



```
public static Logic getInstance(){
      if (theOne == null)
          theOne = new Logic();
      return theOne;
}
```

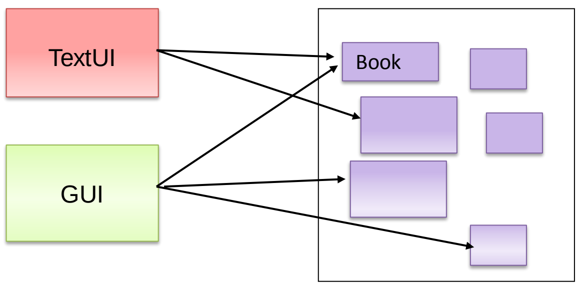
```
//somewhere else in the system...
Logic m = Logic.getInstance();
//instead of ...
~~Logic m = new Logic();~~
```

As shown, the solution class design has a private constructor (note the "-" visibility marker for the constructor), which prevents instantiation from outside the class. The single instance of the singleton class is maintained by a private class-level variable. Access to this object is provided by a public class-level operation getInstance(). In the skeleton code above, getInstance()instantiates a single copy of the singleton class when it is executed for the first time. Subsequent calls to this operation instead return the single instance of the class.

Façade pattern

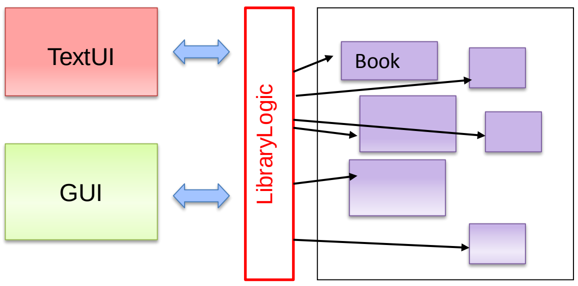
Context

Components need to access functionality deep inside other components. For example, the UI component of a Library system might want to access functionality of the Book class contained inside the Logic component.



Problem
We need to allow access without exposing internal details of component. For example, the UI component should access the functionality from the Logic component without knowing that it contained a Book class inside.

Solution
We put in a class – called a Façade class – that sits between the component internals and users of the component such that all access to the component happens through the Façade class. The following class diagrams show the general concept and also an application to the Library System example.



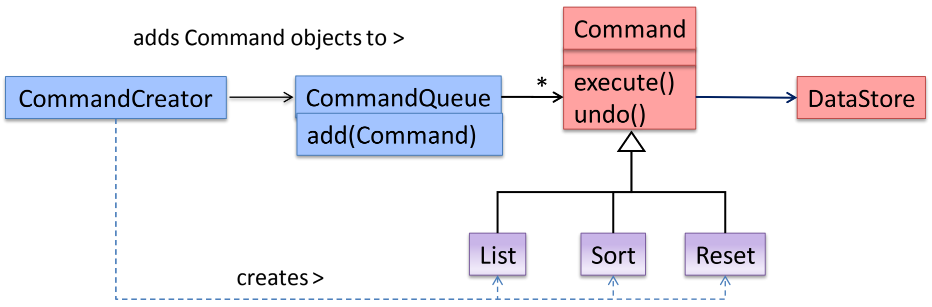
In the new design for the Minesweeper we discussed previously, the Logic class acts as the façade, shielding classes inside the MSLogic component from TextUI and ATD.

Command pattern

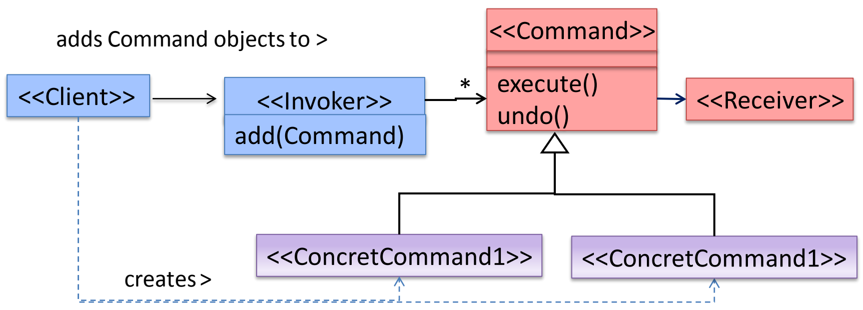
Context
A system has to execute a number of commands, each doing a different task. For example, a system might have to support Sort, List, Reset commands.

Problem
Some part of the code prefers to execute these commands without having to know each command type. For example, there can be a CommandQueue object that is responsible for queuing commands and executing them without any need to know what each command does.

Solution
In the example solution below, the CommandCreator creates List, Sort, and Reset Command objects and adds them to the CommandQueue object. The CommandQueue object treats them all as Command objects and executes/undo them without knowing the specific type of the Command. When executed, each Command object will access the DataStore object to carry out its task. The Command class can also be an abstract class or an interface.



The general form of the solution is as follows.



The <<Client>> creates a <<ConcreteCommand>> object and passes it to the <<Invoker>>. The <<Invoker>> object treats all commands as a general <<Command>> type. <<Invoker>> issues a request by calling execute() on the command. When commands are undoable, <<ConcreteCommand>> stores state for undoing the command prior to invoking execute(). Note that the <<ConcreteCommand>> object may have to be linked to any <<Receiver>> of the command before it is passed to the <<Invoker>>.

Model-View-Controller (MVC) pattern

Context

Most application support storage/retrieval of information, displaying of information to the user (often, using multiple UIs having different formats), and changing stored information based on external inputs.

Problem

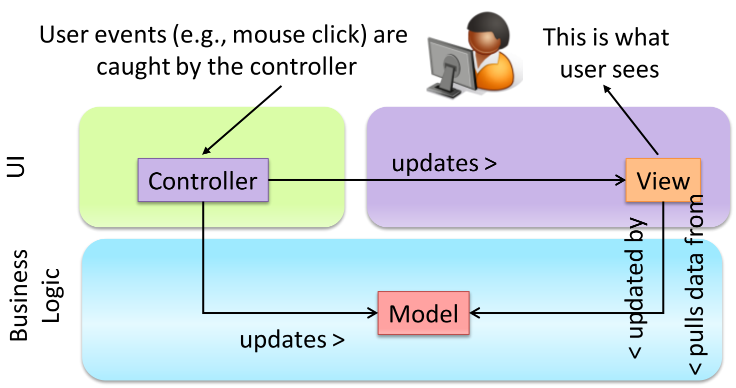
We would like to reduce coupling that result from the interlinked nature of various aspects described above.

Solution

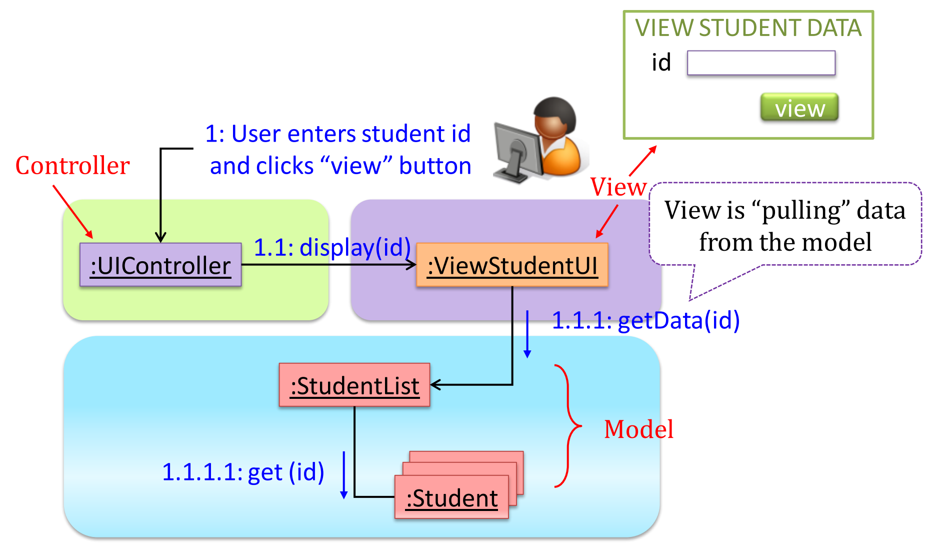
Decouple data, presentation, and control logic of an application by separating them into three different components model, view and controller.

- view : Displays data, interacts with user, and pulls data from the model if necessary.
- controller : Detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view as necessary.
- model :Stores and maintains data. Updates views if necessary.

The relationship between the components can be seen in the diagram below. Typically, the UI is the combination of view and controller.



Now, let's look at a concrete example: Given below is a how MVC appears in a *student management system*. In this scenario, the user is retrieving data of one student.

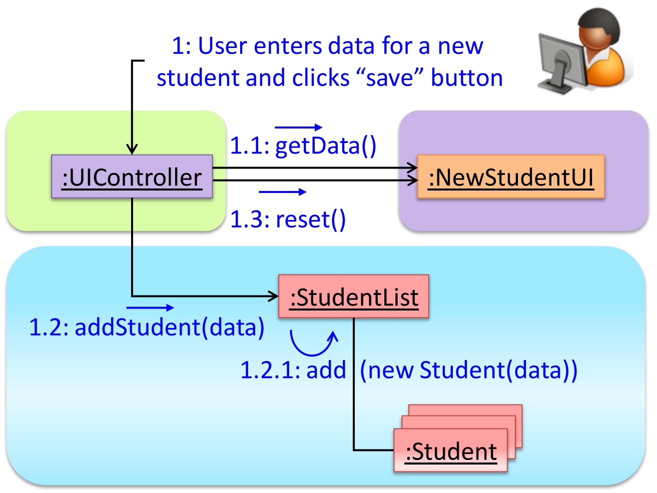


Note that in a simple UI where there's only one view, Controller and View can be combined as one class.

Note: There are many variations of the MVC model used in different domains. For example, the one used in a desktop GUI could be different from the one used in a Web application.

Observer pattern

Here is another scenario from the same student management system where the user is adding a new student to the system.

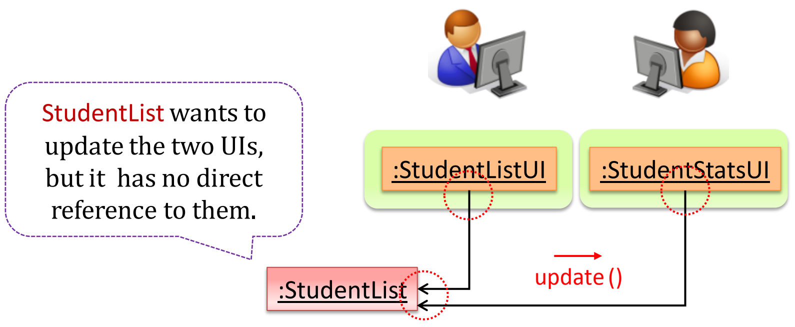


Now, assume the system has two more *views* used in parallel by different users:

(a) StudentListUI : to access a list of students and

(b) StudentStatsUI : to generate statistics of current students.

When a student is added to the database using NewStudentUI shown above, both StudentListUI and StudentStatsUI should get updated automatically, as shown below.


StudentList wants to update the two UIs, but it has no direct reference to them.
:StudentListUI
:StudentStatsUI
:StudentList
update ()

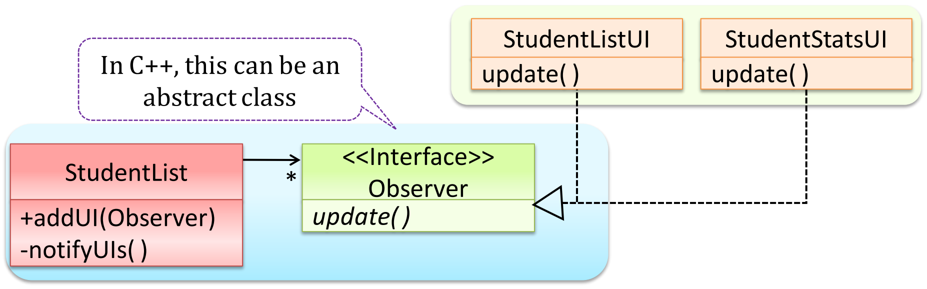
However, StudentList object has no knowledge about StudentListUI and StudentStatsUI (note the direction of the navigability) and has no way to inform those objects. That is one example of the problem addressed by the Observer pattern.

Context
An object (possibly, more than one) is interested to get notified when a change happens to another object. That is, some objects want to 'observe' another object.

Problem
A bidirectional link between the two objects is not desirable. However the two entities need to communicate with each other. That is, the 'observed' object does not want to be coupled to objects that are 'observing' it.

Solution
The *Observer* pattern shows us how an object can communicate with other objects while avoiding a direct coupling with them.

The solution is to force the communication through an interface known to both parties. Let us first look at a concrete example before we learn the generic form of the pattern.



Here is the Observer pattern applied to our student management system. This is how it works:

During initialization of the system,

1. First, we create the relevant objects.

```
StudentList studentList = new StudentList();
StudentListUI listUI = new StudentListUI();
StudentStatusUI statusUI = new StudentStatsUI();
```

2. Next, the two UIs indicate to the StudentList that they are interested in being updated whenever StudentList changes.

```
studentList.addUI(listUI);
studentList.addUI(statusUI);
```

Inside the addUI operation of StudentList, all Observer objects received are added to an internal data structure called observerList.

```
//StudentList class
public void addUI(Observer o) {
   observerList.add(o);
}
```

Later, whenever the StudentList data changes (e.g. when a new student is added to the StudentList), it will update all interested observers by calling its own notifyUIs operation.

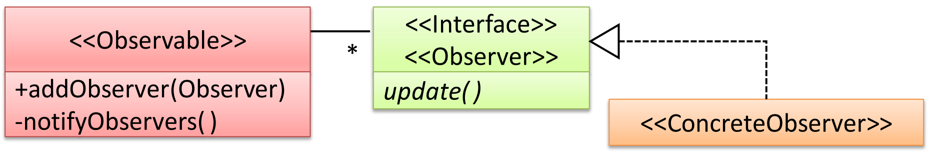
```
//StudentList class
public void notifyUIs() {
   for(Observer o: observerList)
      o.update();
}
```

UIs can pull data from the StudentList whenever the update operation is called.

```
//StudentListUI class
public void update() {
   //refresh UI by pulling data from StudentLits
}
```

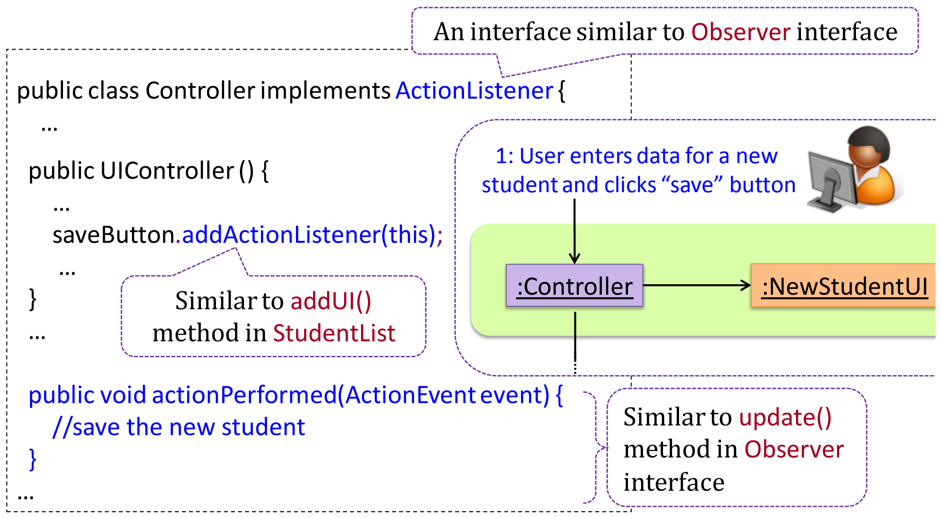
Note how StudentList does not even know of the two UIs but still manages to communicate with them via an interface.

Here is the generic description of the observer pattern:



- <<Observer>> is an interface: any class that implements it can observe an <<Observable>>. Any number of <<Observer>> objects can observe (i.e. listen to changes of) the <<Observable>> object.
- The <<Observable>> maintains a list of <<Observer>> objects. addObserver(Observer) operation adds a new <<Observer>> to the list of <<Observer>> s.
- Whenever there is a change in the <<Observable>, the notifyObservers( ) operation is called that will call the update() operation of all <<Observer>> s in the list.

In a GUI application, how does your Controller gets notified when the “save” button is pushed? UI frameworks such as Java SWING framework has inbuilt support for the Observer pattern. in-built Button → “Observable” object; your Controller→ “Observer” object. The figure below gives an example of such an inbuilt observer pattern being put into use.



```
public class Controller implements ActionListener {
  ...
 public UIController () {
   ...
   saveButton.addActionListener(this);
   ...
 }
 ...
 public void actionPerformed(ActionEvent event) {
   //save the new student
 }
...
```

An interface similar to Observer interface
1: User enters data for a new student and clicks “save” button
:Controller
:NewStudentUI
Similar to addUI() method in StudentList
Similar to update() method in Observer interface

Beyond design patterns

The idea of capturing design ideas as "patterns" is usually attributed to Christopher Alexander. He is a building architect noted for his theories about design. His book *Timeless way of building* talks about “design patterns” for constructing buildings.

Here is a sample pattern from that book:

> *When a room has a window with a view, the window becomes a focal point: people are attracted to the window and want to look through it. The furniture in the room creates a second focal point: everyone is attracted toward whatever point the furniture aims them at (usually the center of the room or a TV). This makes people feel uncomfortable. They want to look out the window, and toward the other focus at the same time. If you rearrange the furniture, so that its focal point becomes the window, then everyone will suddenly notice that the room is much more “comfortable”*

Just like patterns and anti-patterns are found in the field of building architecture, they are general concepts applicable to any activity. In software engineering, we can have many types of patterns: Analysis patterns, Design patterns, Testing patterns, Architectural patterns, Project management patterns, and so on.

The abstraction occurrence pattern is more of an analysis pattern than a design pattern and MVC is more of an architectural pattern.

You can create your own patterns too. If you find yourself solving a similar problem many times and eventually hit-upon a non-obvious, and better, solution, you can formulate it as a pattern so that it can be reused by others. However, don’t reinvent the wheel; may be the pattern already exists.

The more patterns you know, the more ‘experienced’ you are. Therefore, try to learn more patterns (at least the context and problem). Where to find them? Some are domain-specific (e.g. patterns for distributed applications), some are made in-house (e.g. patterns in your company/project) and some can be your own (e.g. from your past experience). However, most are common, and well known. For example, GoF book [1] contains 23 design patterns:

- **Creational**: About object creation. They separate the operation of an application from how its objects are created
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structural**: About the composition of objects into larger structures while catering for future extension in structure.
    - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- **Behavioral**: Define how objects interact and how responsibility is distributed among them.
    - Chain of Responsibility, Command, Interpreter, Template Method, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor

When using patterns, be careful not to overuse them. Do not throw patterns at a problem at every opportunity. Patterns have overhead such as adding more classes or increasing the levels of abstraction; Use only whey you need them. Before applying a pattern, make sure that:

- It does improve the design substantially, not just superficially.
- You have studied the tradeoff. There are times where design pattern is not appropriate (or an overkill).

A case study

Let us design a class structure for a Stock Inventory System (SIS), to be used in a shop that sells electronic appliances. The shop sells appliances, and accessories for those appliances. SIS simply stores information about each item in store.

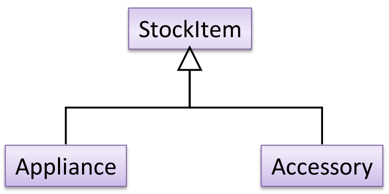
> Use cases: create a new item, view information about an item, modify information about an item, view all available accessories for a given appliance, list all items in store.

SIS can be accessed using multiple terminals: Shop assistants will use their own terminals to access SIS, while the shop manager's terminal continuously displays a list of all items in store. In the future, we expect suppliers of items to use their own application to connect to SIS to get real-time information about current stock status. User authentication is not required for the current version, but may be required in the future.

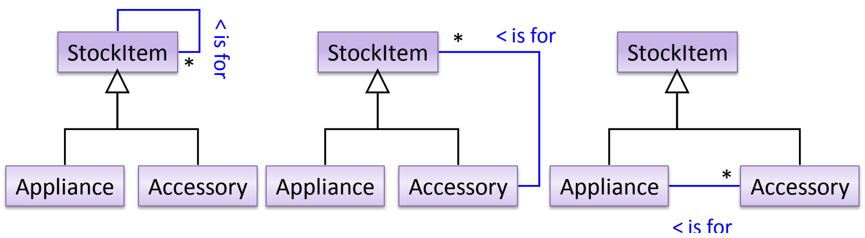
Let us also apply design patterns where appropriate.

Here is a step by step explanation of the design. Note that this is one of the many possible designs.

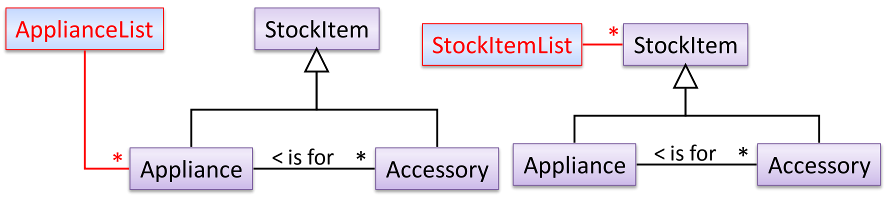
A StockItem can be an Appliance or an Accessory.



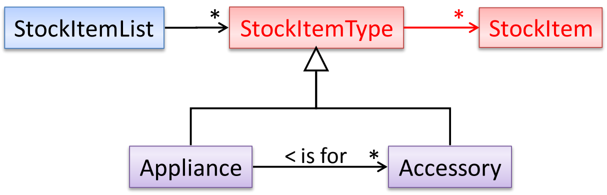
We need to track which Accessory is for which Appliance. Which one is more appropriate?



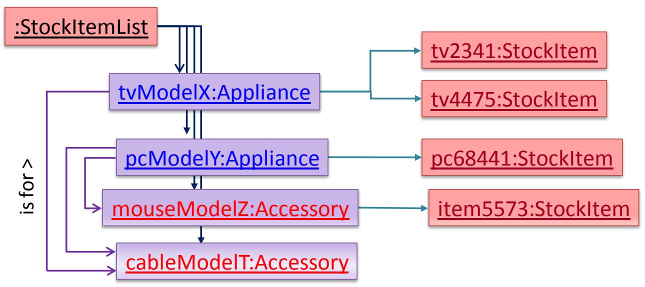
The third one seems more appropriate. Next, should we keep a list of Appliances or keep a list of StockItems?



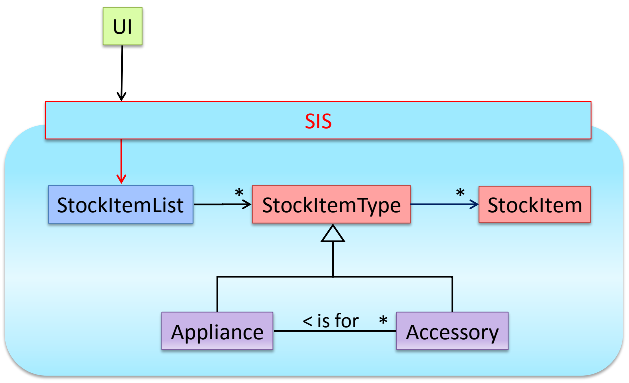
The latter seems more appropriate. Next, we can apply the abstraction occurrence pattern to keep track of StockItems.



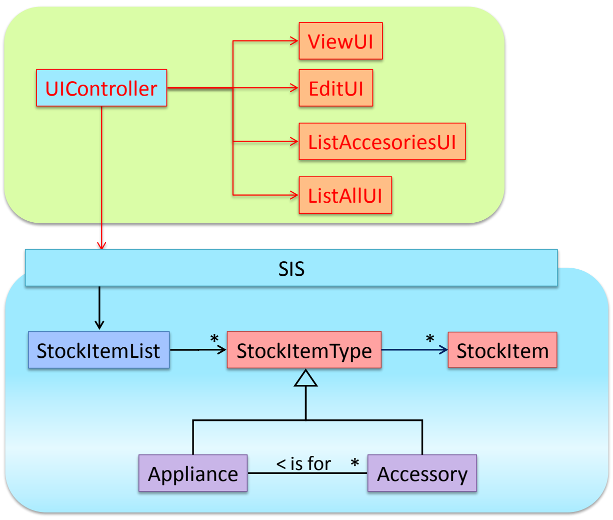
Note that we have also added navigabilities too. Here's a sample object diagram for the class model we have come up with so far.



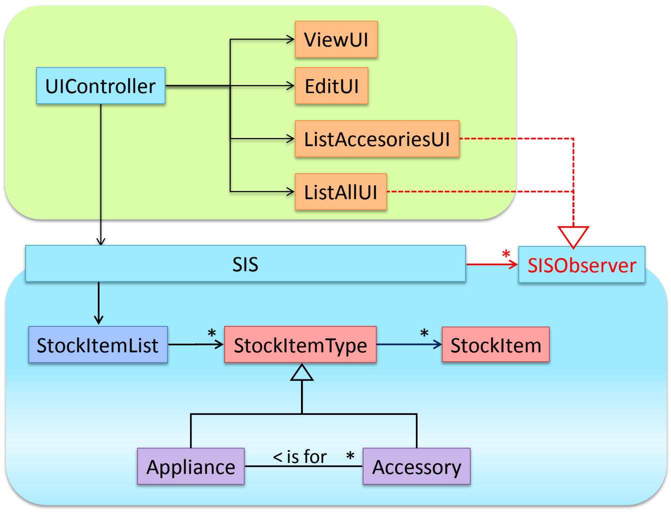
We can apply the *façade pattern* to shield the SIS internals from the UI.



UI consists of multiple views. We can apply MVC pattern here.



Some views need to be updated when the data change. We can apply Observer pattern here.



In addition, we can apply the Singleton pattern to the façade class.

References

[1] *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

Worked examples

[Q1]

Assume you are implementing a class called PassKey that can have only 3 instances at any time. These instances are to be called entryPassKey, exitPassKey, lockPassKey. They are to be created at system startup. All three instances are immutable. As you know, Singleton pattern can limit the number of instances to only one. Explain how you will extend the idea behind the Singleton pattern to implement the Passkey class. Show code examples where necessary.

[A1]

There are other acceptable variations.

```
class PassKey{
        private static PassKey entryPassKey = new PassKey(); //this way, the object is created as system startup
        private static PassKey exitPassKey = new PassKey();
        private static PassKey lockPassKey = new PassKey();
        private void PassKey(){…} //private constructor
        public static PassKey getEntryPassKey() {
         return entryPassKey; //no need to check whether it is null
        }
        //getExitPassKey() and getLockPassKey() are similar.
        …
}
```

[Q2]
Which pairs of classes are likely to be the <<Abstraction>> and the <<Occurrence>> of the abstraction occurrence pattern?

One of the key things to keep in mind is that the <<Abstraction>> does not represent a real entity. Rather, it represents some information common to a set of objects. A single real entity is represented by an object of <<Abstraction>> type and <<Occurrence>> type.

Before applying the pattern, some attributes have the same values for multiple objects. For example, w.r.t. the BookTitle-BookCopy example given in this handout, values of attributes such as book_title, ISBN are exactly same for copies of the same book.
After applying the pattern, the Abstraction and the Occurrence classes together represent one entity. It is like one class has been split into two. For example, a BookTitle object and a BookCopy object combines to represent an actual Book.

i. CarModel, Car. (Here CarModel represents a particular model of a car produced by the car manufacturer. E.g. BMW R4300)
ii. Car, Wheel
iii. Club, Member
iv. TeamLeader, TeamMember
v. Magazine (E.g. ReadersDigest, PCWorld), MagazineIssue

[A2]
i. CarModel, Car : Yes
ii. Car, Wheel : No. Wheel is a 'part of' Car. A wheel is not an occurrence of Car.
iii. Club, Member: No. this is a 'part of' relationship.
iv. TeamLeader, TeamMember: No. A TeamMember is not an occurrence of a TeamLeader or vice versa.
v. Magazine, MagazineIssue: Yes.

[Q3]
Given below are some common elements of a design pattern. Using similar elements, describe a pattern that is not a design pattern. It must be a pattern you yourself have noticed, not a pattern already documented by others. You may also give a pattern not related to software.

Some examples:

- A pattern for testing textual UIs.
- A pattern for striking a good bargain at a mall such as Sim-Lim Square.

Elements of a pattern: Name, Context, Problem, Solution, Anti-patterns (optional), Consequences (optional), other useful information (optional).

[A3]
No answer provided.

[Q4]
Assume you are designing a multiplayer version of the Minsweeper game where any number of players can play the same Minefield. Players use their own PCs to play the game. A players scores by deducing a cell correctly before any of the other players do. Once a cell is correctly deduced, it appears as either marked or cleared for all players. Comment on how each of the following architectural patterns (also called architectural *styles*) could be potentially useful when designing the architecture for this game.

a) Client-server

b) Transaction-processing

c) MVC

d) SOA

e) multi-layer (n-tier)

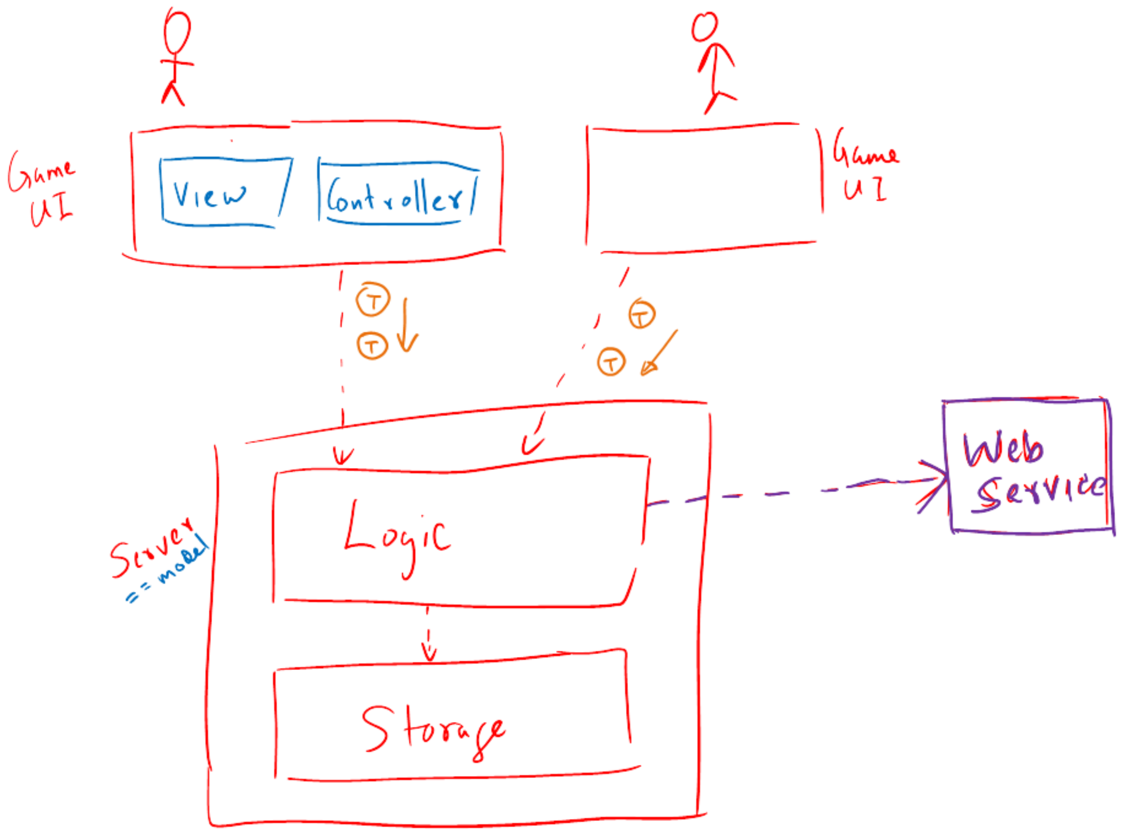
[A4]
a) Client-server – Clients can be the game UI running on player PCs. The server can be the game logic running on one machine.

b) Transaction-processing – Each player action can be packaged as transactions (by the client component running on the player PC) and sent to the server. Server processes them in the order they are received.

c) MVC – Game UI (running on player machines) can have the view and the controller part while the server can be considered as the model.

d) SOA – The game can access a remote web services for things such as getting new puzzles, validating puzzles, charging players subscription fees, etc.

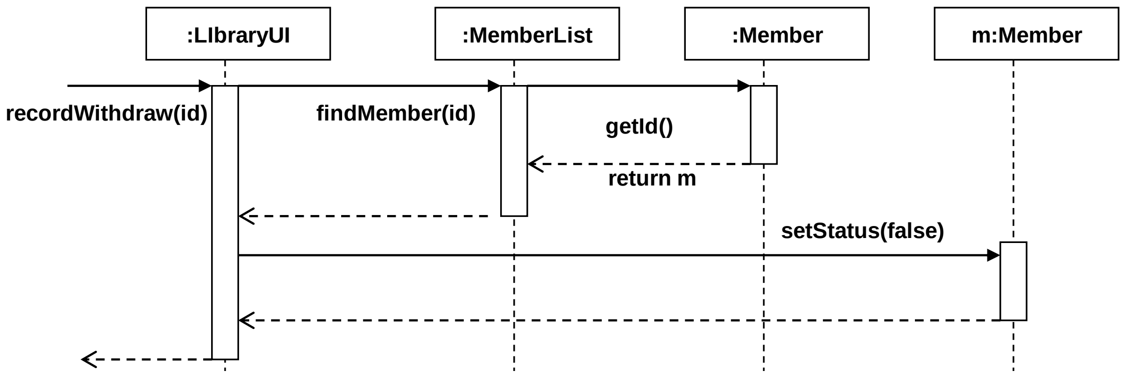
e) Multi-layer – The server component can have two layers: logic layer and the storage layer.



[Q5]

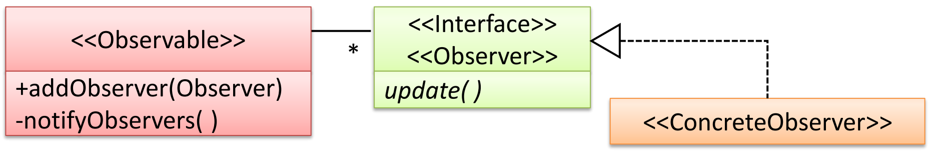
a) Explain how polymorphism is used in the Observer pattern.

b) Given below is a sequence diagram from a library system. It shows the interaction related to a member withdrawing his/her membership from the library. Redraw this diagram after applying the Façade pattern to the system. The objective of applying the pattern is to shield the UI layer from the complexities of the business logic layer. Note that the 'getId()' method is supposed to be invoked in a loop until we find the member whose id matches the id given as the parameter. The loop has been removed to simplify the diagram as it is not relevant to the question.



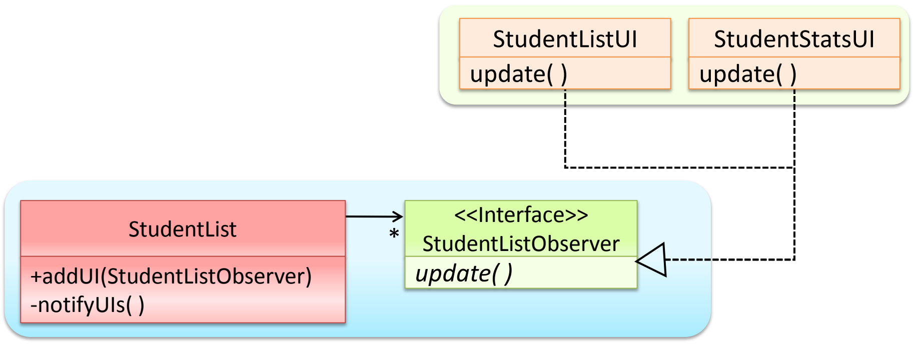
[A5]

(a)

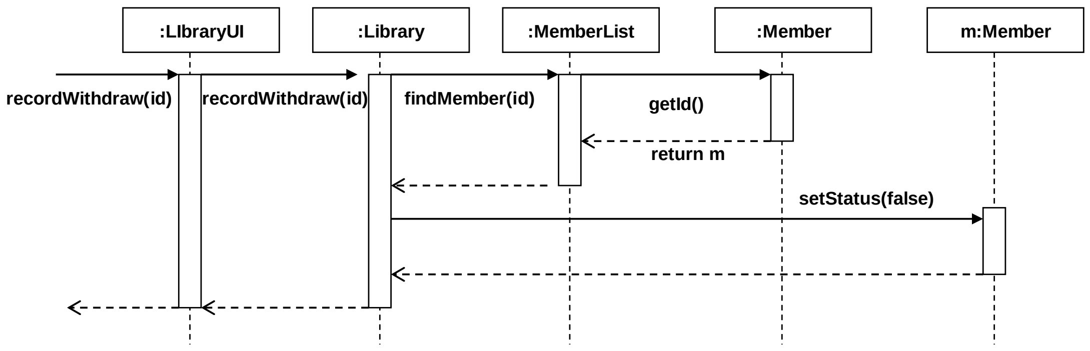


With respect to the canonical form of the Observer pattern given above, when the Observable object invokes the notifyObservers() method, it is treating all ConcreteObserver objects as a general type called Observer and calling the update() method of each of them. However, the update() method of each ConcreteObserver could potentially show different behavior based on its actual type. That is, update()method shows polymorphic behavior.

In the example give below, the notifyUIs operation can result in StudentListUI and StudentStatsUI changing their views in two different ways.



(b) In the example below, Library is the Façade class. Note how the LibraryUI is simply forwarding the recordWithdraw(id) operation to the Façade class without breaking it to two steps as was done in the original diagram. This is because UI should limit itself to its job of 'interacting with the user' while the logic aspect of operations should be done elsewhere in the system.



---End of Document---