[L9P2]

Déjà vu: Using patterns to solve recurring problems

FAQ: Why is this topic covered so late in the semester?
Answer: We want you to try a design without patterns first, and then, refactor the design to introduce patterns. That way, you can not only see the difference, but also get to practice refactoring.

Patterns

In software development, there are certain problems that crop up repeatedly.

- An example problem about software architecture: what is the best architecture for a given type of system?
- An example problem about the interaction between classes: how to lower the coupling between UI and Logic classes?

After repeated attempts at solving 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, this handout attempts to guide the reader on understanding patterns as a general concept by referring to a few representative examples to illustrate the concept of patterns. This handout focuses more on patterns in the area of software design (i.e. design patterns). Having understood the general idea of patterns, the reader should be able to pick up more patterns from other resources.

The common format to describe a pattern consists of the following components:

- **Context**: The situation or scenario where the design problem is encountered.
- **Problem**: The main difficulty to be resolved. The criteria for a good solution are also identified to evaluate solutions.
- **Solution:** The core of the solution. 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): Identifying the pros and cons of applying the pattern.
- **Other useful information** (optional): Code examples, known uses, other related patterns, etc.

Abstraction occurrence pattern

Context

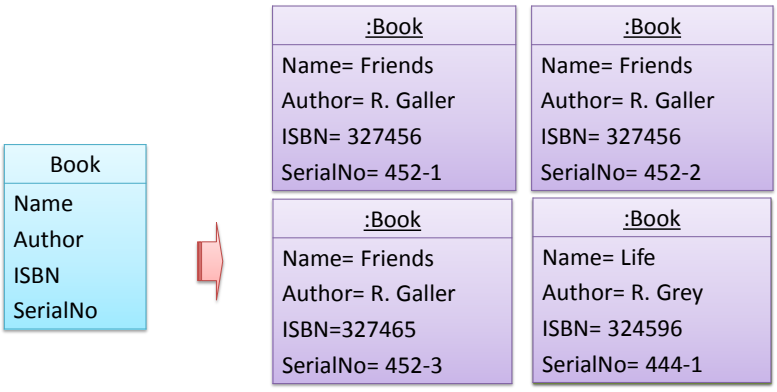
There is a group of similar entities that appears to be 'occurrences' (or 'copies') of the same thing, sharing lots of common information, but also differing in significant ways.

For example, in a library, there can be multiple copies of same book title. Each copy shares common information such as 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).

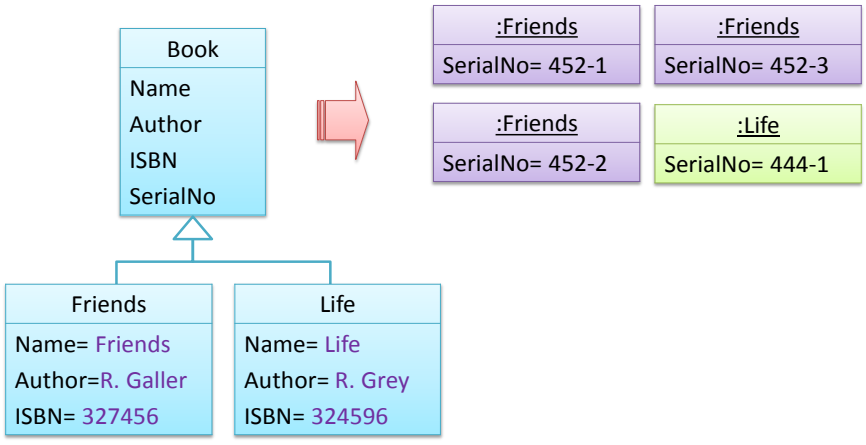
Problem

Representing the objects mentioned previously as a single class would be problematic (refer to anti-pattern description below). A better way to represent such instances is required, which should avoid duplicating the common information. Without duplicated information, inconsistency is avoided should these common information be changed.

Anti-patterns



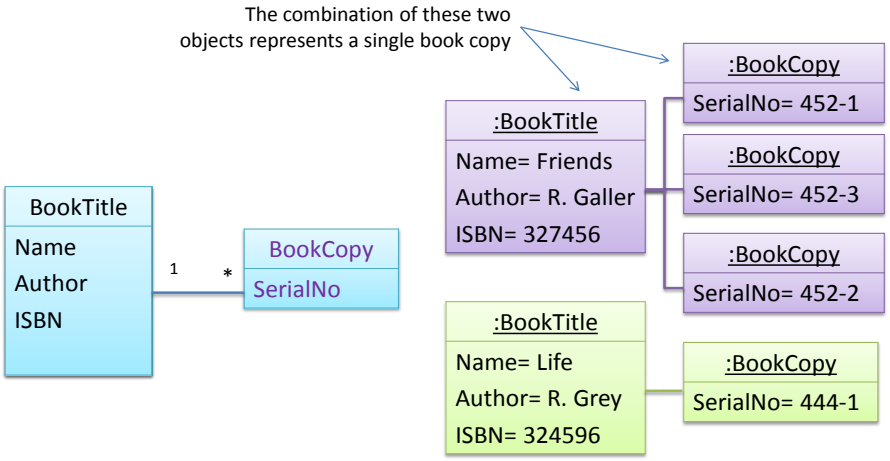
Take for example the problem of representing books in a library. Assume that there could be multiple copies of the same title, bearing the same ISBN number, but different serial numbers. The above diagram shows an inferior or incorrect design for this problem. It requires common information to be duplicated by all instances. This will not only waste storage space, but also creates a consistency problem. Suppose that after creating several copies of the same title, the librarian realized that the author name was wrongly spelt. 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 the existing copies to avoid inconsistency.



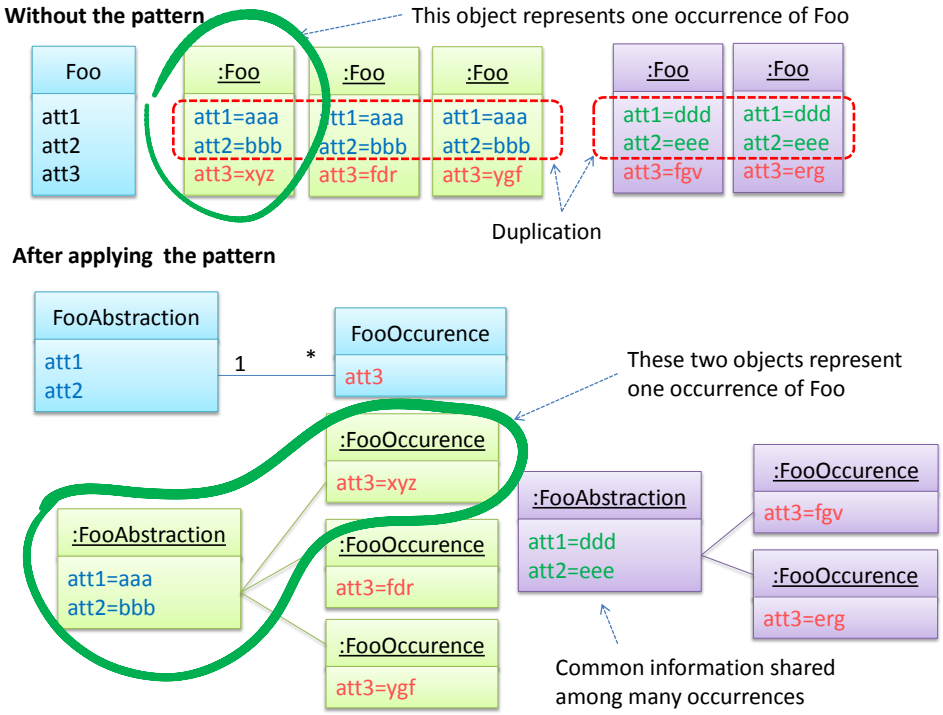
The design above segregates the common and unique information into a class hierarchy. Each book title is represented by a separate class with common data (i.e. Name, Author, ISBN) hard-coded in the class itself. This solution is worse than the first as each book title is represented as a class, resulting 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 the two situations *before* and *after* applying the pattern.



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. 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

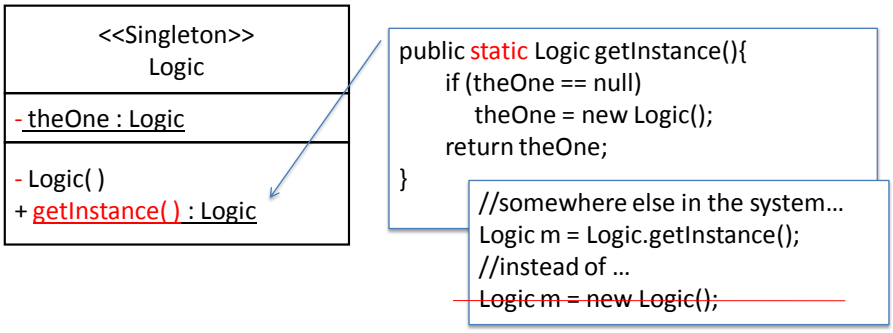
In most systems, it is common to restrict the number of instantiated objects of certain classes to just one (e.g. the main controller class of the system). These single instances are commonly known as *singletons*.

Problem

Prohibit the instantiation of more than one object from a singleton class, with the single instance easily shared among those who need it.

Solution

The key insight of the solution is that the *constructor* of the singleton class cannot be *public*. Since a *public constructor* will allow others to instantiate the class at will, a *private constructor* should be used instead. In addition, a public method is provided to access the *single instance*. This solution is described below.

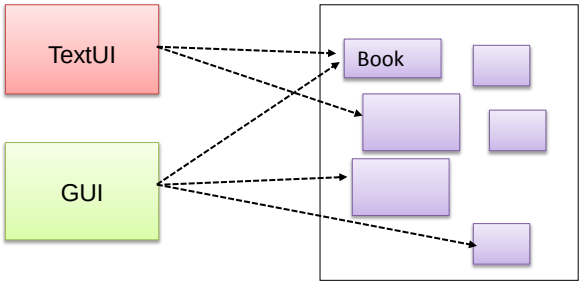


As shown, the solution makes the constructor private (note the "-" visibility marker for the constructor), which prevents instantiation from outside the class. The single instance of the singleton class is maintained by a private class-level variable. Access to this object is provided by a public class-level 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 return the single instance of the class.

Façade pattern

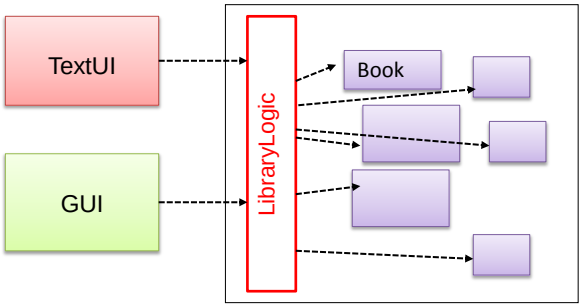
Context

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
Access to the component should be allowed without exposing its internal details. For example, the UI component should access the functionality of the Logic component without knowing that it contained a Book class within it.

Solution
Include a Façade[4] 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 diagram shows the application of the Façade pattern to the Library System example. In this example, the LibraryLogic class acts as the Façade class.

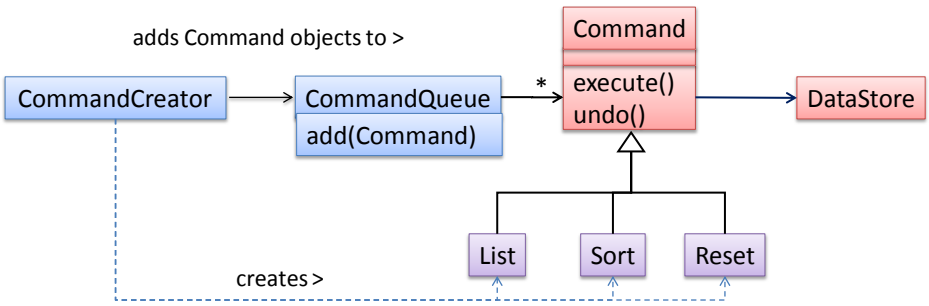


Command pattern

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

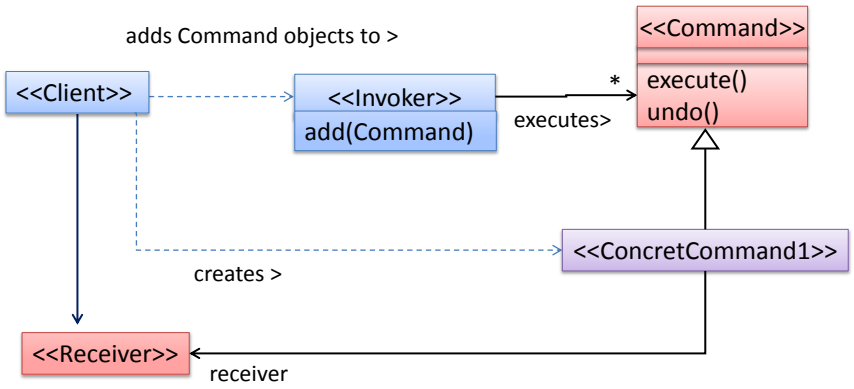
Problem
It is preferable that some part of the code execute these commands without having to know each command type. For example, there can be a CommandQueue object that is responsible for queuing commands and executing them without knowledge of what each command does.

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 performs the execute/undo operation on each of them without knowledge of the specific Command type. 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.

[4] Façade is a French word that means 'front of a building'



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. If a command is undoable, <<ConcreteCommand>> will store the state for undoing the command prior to invoking execute(). In addition, the <<ConcreteCommand>> object may have to be linked to any <<Receiver>> of the command before it is passed to the <<Invoker>>. Note that an application of the command pattern does not have to follow the structure given above. The essential element is to have a general <<Command>> object that can be passed around, stored, executed, etc.

Model-View-Controller (MVC) pattern

Context

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

Problem

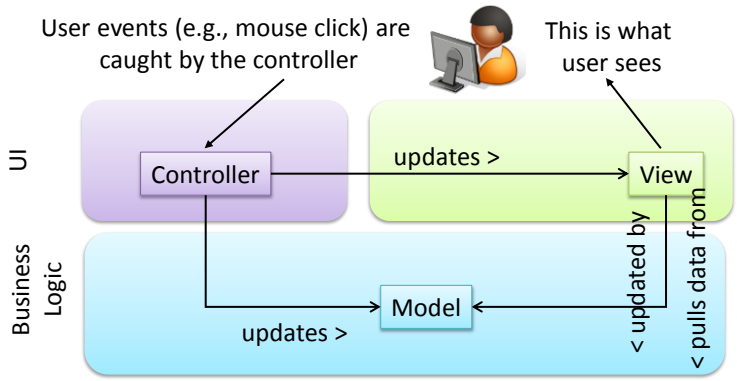
To reduce coupling resulting from the interlinked nature of the features described above.

Solution

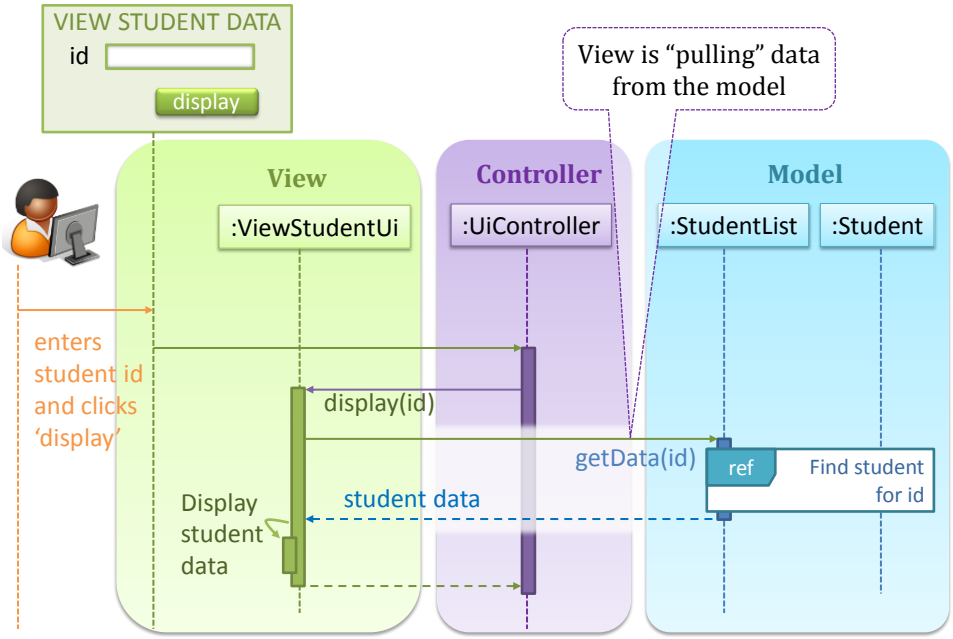
To decouple data, presentation, and control logic of an application by separating them into three different components: Model, View and Controller.

- View : Displays data, interacts with the 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 when necessary.
- Model :Stores and maintains data. Updates views if necessary.

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



Given below is a concrete example of MVC applied to a *student management system*. In this scenario, the user is retrieving data of one student.



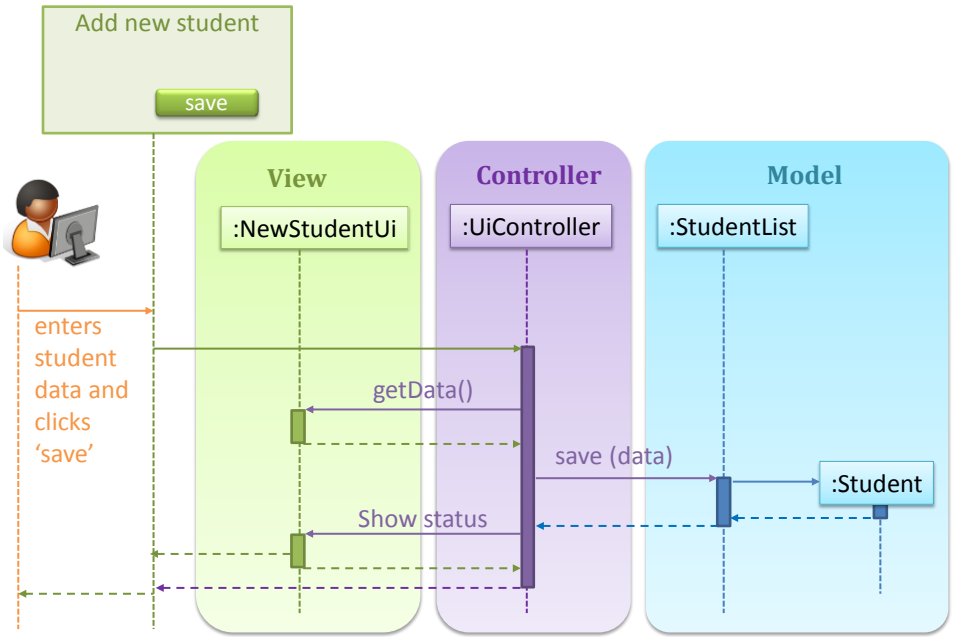
In the diagram above, when the user clicks on a button using the UI, the 'click' event is caught and handled by the UiController.

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

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.

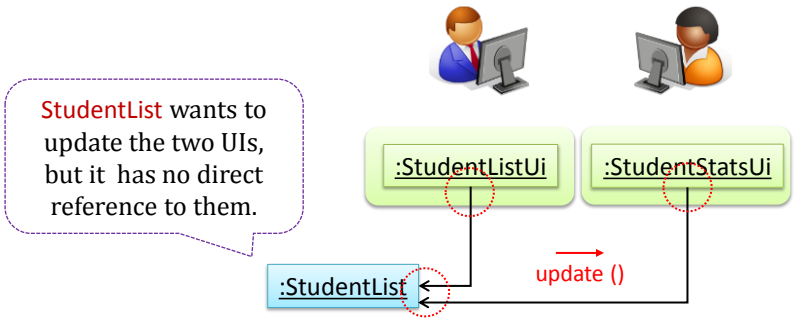


Now, assume the system has two additional *views* used in parallel by different users:

(a) StudentListUi : that accesses a list of students and

(b) StudentStatsUi : that generates 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.



However, the StudentList object has no knowledge about StudentListUi and StudentStatsUi (note the direction of the navigability) and has no way to inform those objects. This is an example of the type of problem addressed by the Observer pattern.

Context

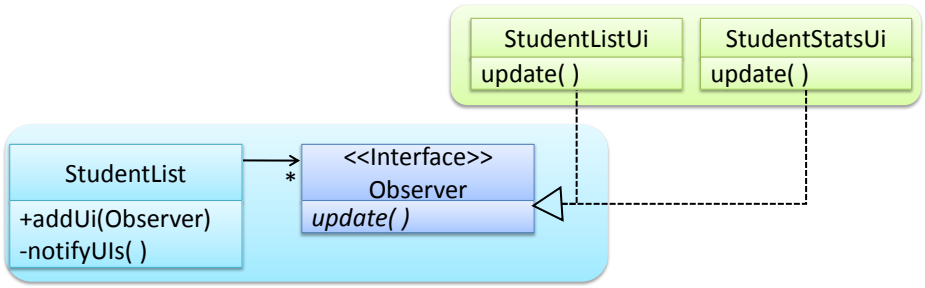
An object (possibly, more than one) is interested to get notified when a change happens to another object. That is, some objects want to 'observe' another object.

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. A concrete example is given below.



Here is the Observer pattern applied to the student management system.

During initialization of the system,

1. First, 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. This is also known as 'subscribing for updates'.

```
studentList.addUi(listUi);
studentList.addUi(statusUi);
```

Within the addUi operation of StudentList, all Observer objects subscribers are added to an internal data structure called observerList.

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

As such, whenever the data in StudentList changes (e.g. when a new student is added to the StudentList), all interested observers are updated by calling the notifyUIs operation.

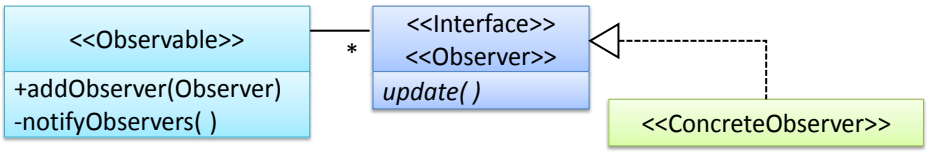
```
//StudentList class
public void notifyUIs() {
  for(Observer o: observerList) //for each observer in the list
      o.update();
}
```

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

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

Note that StudentList is unaware of the exact nature of the two UIs but still manages to communicate with them via an intermediary.

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 is the Controller notified when the "save" button is clicked? UI frameworks such as JavaFX has inbuilt support for the Observer pattern.

Beyond design patterns

The notion 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"*

Apparently, patterns and anti-patterns are found in the field of building architecture. This is because they are general concepts applicable to any domain, not just software design. In software engineering, there are many general types of patterns: Analysis patterns, Design patterns, Testing patterns, Architectural patterns, Project management patterns, and so on.

In fact, the abstraction occurrence pattern is more of an analysis pattern than a design pattern, while MVC is more of an architectural pattern.

New patterns can be created too. If a common problem needs to be solved frequently that leads to a non-obvious and better solution, it can be formulated as a pattern so that it can be reused by others. However, don't reinvent the wheel; the pattern might already exist.

The more patterns one acquires, the more 'experienced' he/she is. Exposing oneself to a multitude of patterns (at least the context and problem) is a must. Some patterns are domain-specific (e.g. patterns for distributed applications), some are created in-house (e.g. patterns in

the company/project) and some can be self-created (e.g. from past experience). However, most are common, and well known. As an 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**: Defining 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 come with overhead such as adding more classes or increasing the levels of abstraction. Use them only when they are needed. Before applying a pattern, make sure that:

- there is substantial improvement in the design, not just superficial.
- the associated tradeoffs are carefully considered. There are times when a design pattern is not appropriate (or an overkill).

Case study

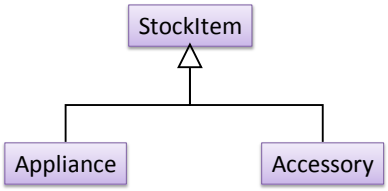
Next, let us look at a case study that shows how design patterns are used in the design of a class structure for a Stock Inventory System (SIS) for a shop. The shop sells appliances, and accessories for the appliances. SIS simply stores information about each item in the store.

> 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 the store.

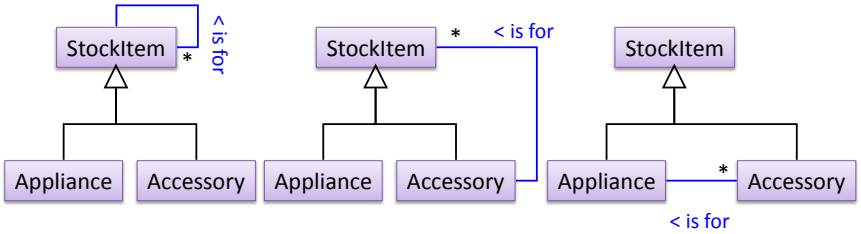
SIS can be accessed using multiple terminals. Shop assistants use their own terminals to access SIS, while the shop manager's terminal continuously displays a list of all items in store. In the future, it is expected that suppliers of items use their own applications to connect to SIS to get real-time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

A step by step explanation of the design is given below. Note that this is one out of many possible designs. Design patterns are also applied where appropriate.

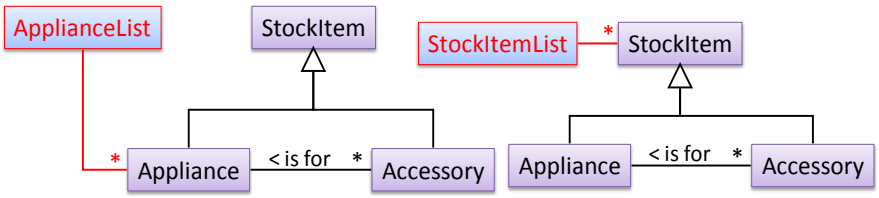
A StockItem can be an Appliance or an Accessory.



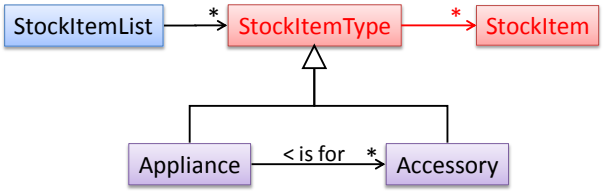
To track that each Accessory is associated with the correct Appliance, consider the following alternative class structures.



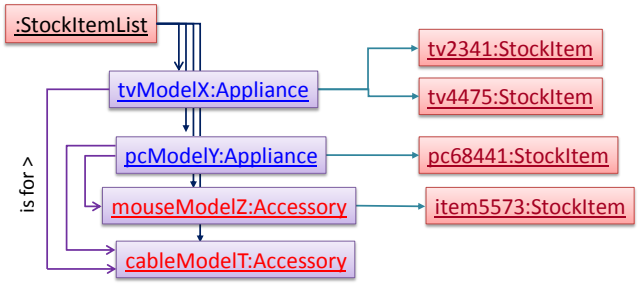
The third one seems more appropriate (the second one is suitable if accessories can have accessories). Next, consider between keeping a list of Appliances, and a list of StockItems. Which is more appropriate?



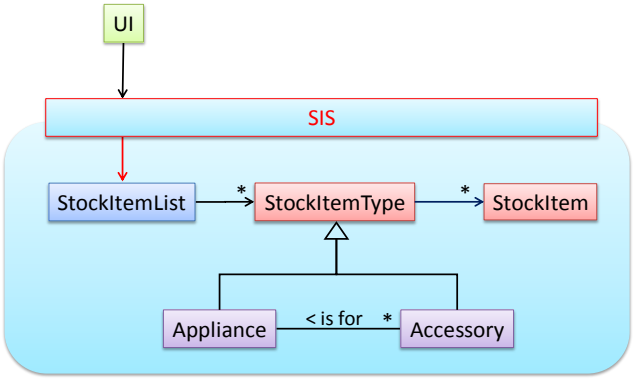
The latter seems more suitable because it can handle both appliances and accessories the same way. Next, an abstraction occurrence pattern is applied to keep track of StockItems.



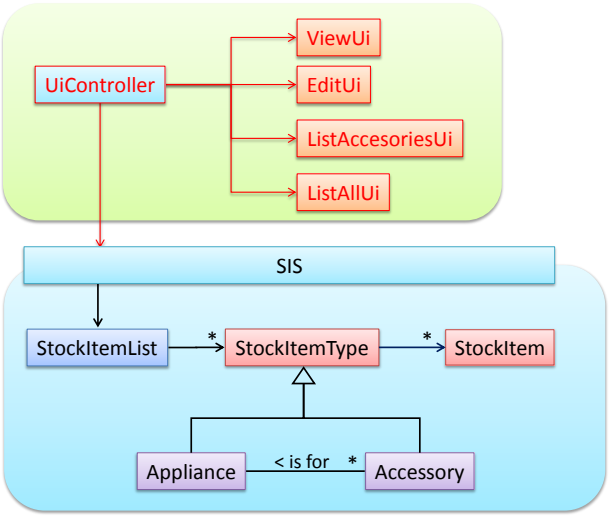
Note the inclusion of navigabilities. Here's a sample object diagram based on the class model created thus far.



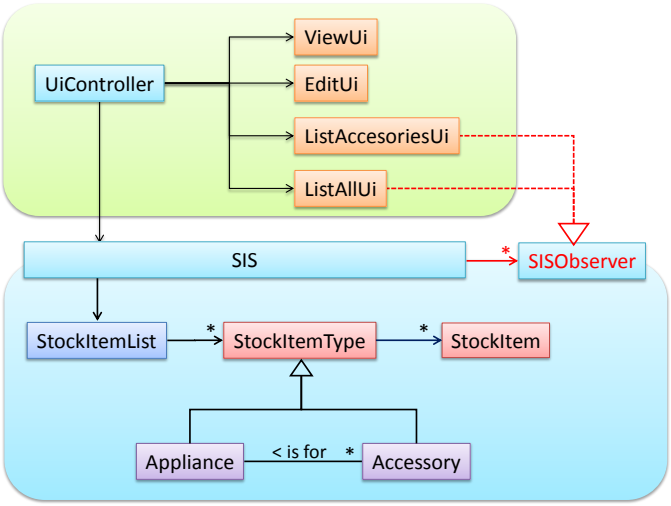
Next, apply the *façade pattern* to shield the SIS internals from the UI.



As UI consists of multiple views, the MVC pattern is applied here.



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



In addition, the Singleton pattern can be applied 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 that you are implementing a class called PassKey that can have only three instances at any time. These instances are to be called entryPassKey, exitPassKey and lockPassKey. They are to be created at system startup. All three instances are immutable (i.e. once created, cannot be changed). 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?

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]

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 the 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 : 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 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 player 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 styles (these architectural styles are patterns too) could be potentially useful when designing the architecture for this game.

a) Client-server

b) Transaction-processing

c) MVC

d) SOA (Service Oriented Architecture)

e) multi-layer (n-tier)

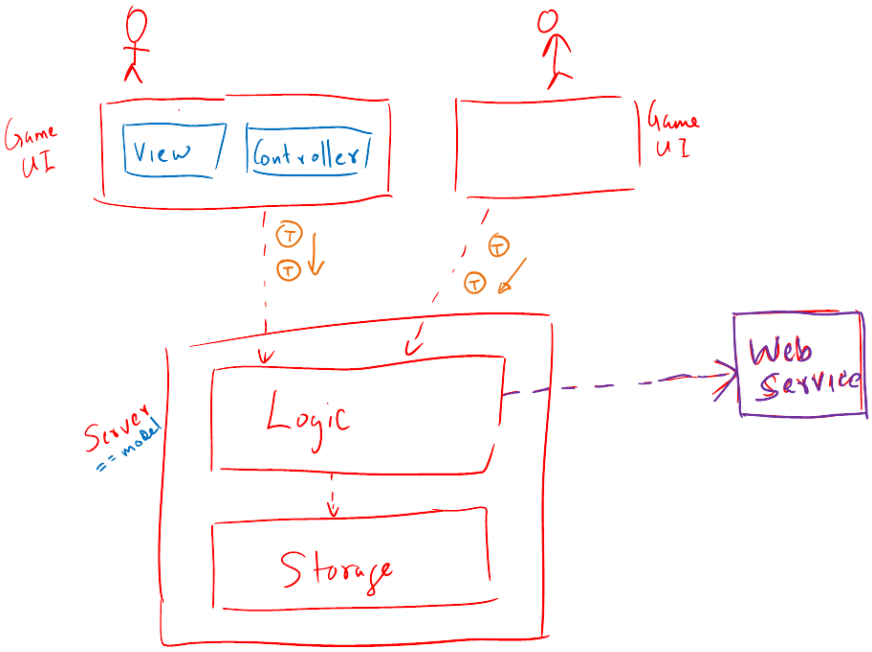
[A4]
a) Client-server – Clients can be the game UI running on player PCs. The server can be the game logic running on one machine.

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.



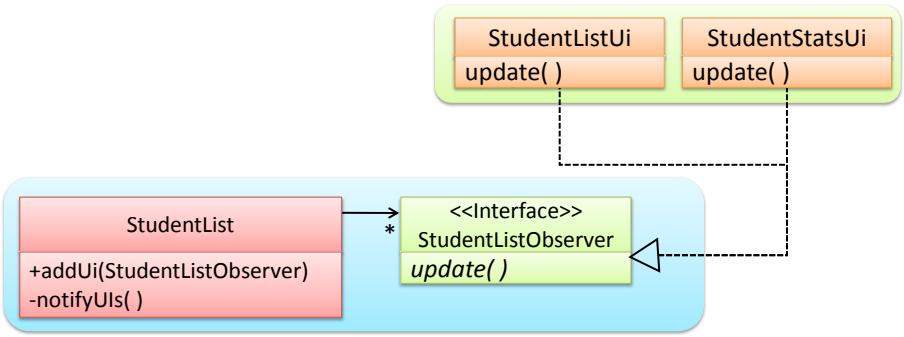
[Q5]

Explain how polymorphism is used in the Observer pattern.

[A5]

With respect to the general 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.



--- End of handout ---