Software design patterns

Introduction

What

★★☆☆ Can explain design patterns

> **Design pattern**: An *elegant reusable solution* to a *commonly recurring problem* within a *given context* in software design.

In software development, there are certain problems that recur in a certain context.

Some examples of recurring design problems:

Design Context	Recurring Problem
Assembling a system that makes use of other existing systems implemented using different technologies	What is the best architecture?
UI needs to be updated when the data in the application backend changes	How to initiate an update to the UI when data changes without coupling the backend to the UI?

After repeated attempts at solving such problems, better solutions are discovered and refined over time. These solutions are known as design patterns, **a term popularized by the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by the so-called "Gang of Four" (GoF)** written by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Exercises

Format

★★☆☆ Can explain design patterns format

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.
- **Solution**: The core of the solution. It is important to note that the solution presented only includes the most general details, 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.

Exercises

Singleton pattern

What

★★☆☆ Can explain the Singleton design pattern

Context

Certain classes should have no more than just one instance (e.g. the main controller class of the system). These single instances are commonly known as *singletons*.

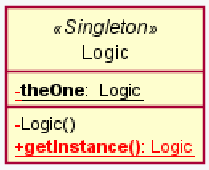
Problem

A normal class can be instantiated multiple times by invoking the constructor.

Solution

Make the constructor of the singleton class `private`, because a `public` constructor will allow others to instantiate the class at will. Provide a `public` class-level method to access the *single instance*.

Example:



Exercises

Implementation

★★☆☆ Can apply the Singleton design pattern

Here is the typical implementation of how the Singleton pattern is applied to a class:

```
class Logic {
    private static Logic theOne = null;

    private Logic() {
        ...
    }

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

Notes:

- The constructor is `private`, 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()` which 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.

If `Logic` was not a Singleton class, an object is created like this:

```java
Logic m = new Logic();
```

But now, the `Logic` object needs to be accessed like this:

```java
Logic m = Logic.getInstance();
```

Evaluation

★★★☆ 🏆 **Can decide when to apply Singleton design pattern**

Pros:

- easy to apply
- effective in achieving its goal with minimal extra work
- provides an easy way to access the singleton object from anywhere in the code base

Cons:

- The singleton object acts like a global variable that increases coupling across the code base.
- In testing, it is difficult to replace Singleton objects with stubs (static methods cannot be overridden).
- In testing, singleton objects carry data from one test to another even when you want each test to be independent of the others.

Given that there are some significant cons, it is recommended that you apply the Singleton pattern when, in addition to requiring only one instance of a class, there is a risk of creating multiple objects by mistake, and creating such multiple objects has real negative consequences.

Abstraction occurrence pattern

What

★★★★ 🏆 **Can explain the Abstraction Occurrence design pattern**

Context

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

📦 In a library, there can be multiple copies of the 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).

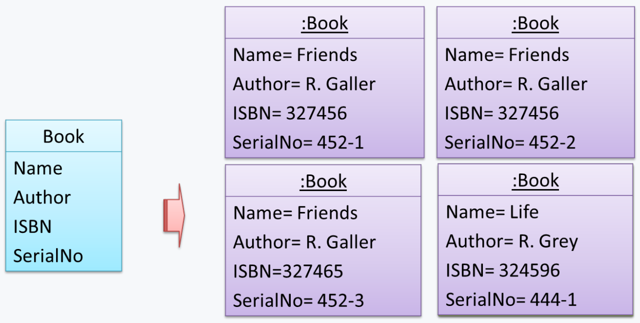
Other examples:

- Episodes of the same TV series
- 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 because it results in duplication of data which can lead to inconsistencies in data (if some of the duplicates are not updated consistently).

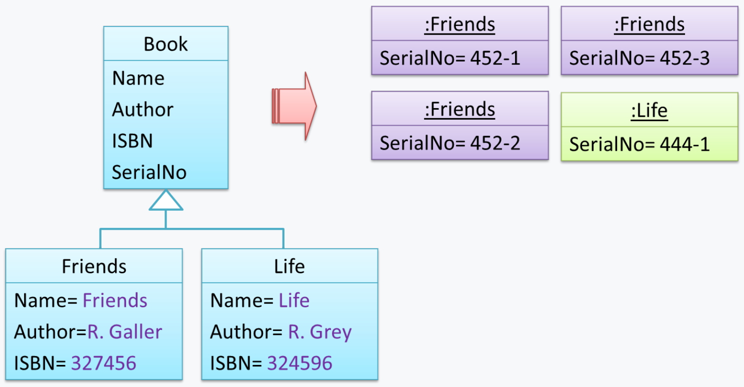
📦 Take for example the problem of representing books in a library. Assume that there could be multiple copies of the same title, bearing the same ISBN number, but different serial numbers.



The above solution 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 (or the system) has to make sure that all information entered is the same as the existing copies to avoid inconsistency.

Anti-pattern

📦 Refer to the same Library example given above.

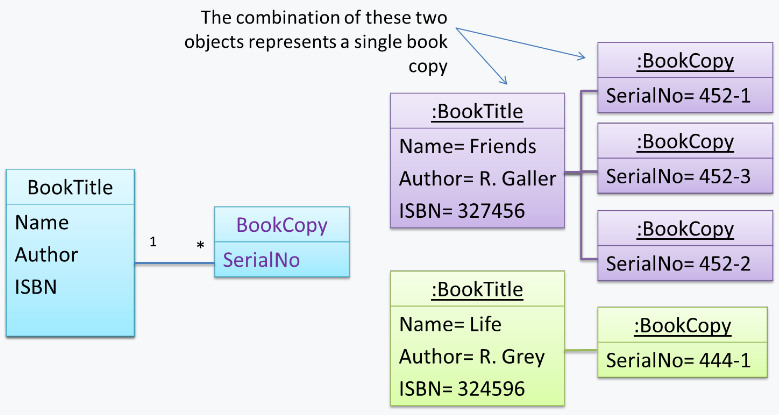


The design above segregates the common and unique information into a class hierarchy. Each book title is represented by a separate class with common data (i.e. Name, Author, ISBN) hard-coded in the class itself. This solution is problematic because each book title is represented as a class, resulting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

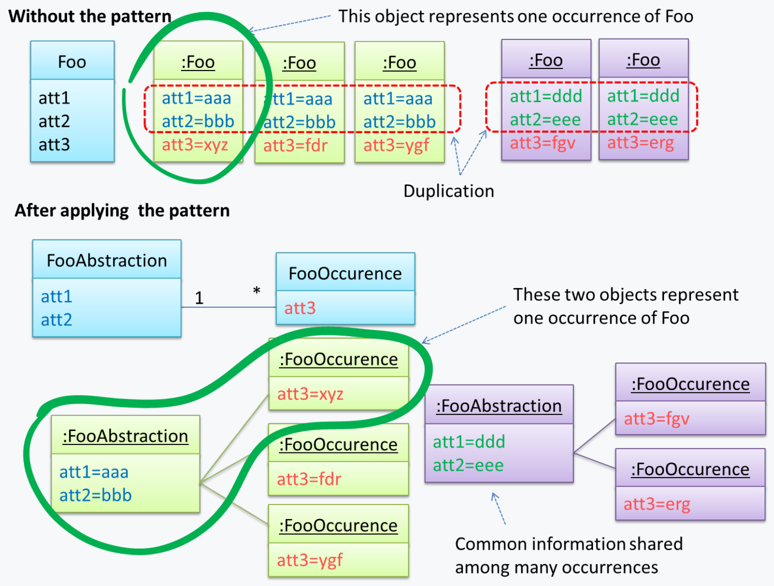
Solution

Let a copy of an entity (e.g. a copy of a book) be represented by two objects instead of one, separating the common and unique information into two classes to avoid duplication.

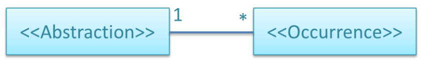
📦 Given below is how the pattern is applied to the Library example:



Here's a more generic example:



The general solution:



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.

Exercises

Facade pattern

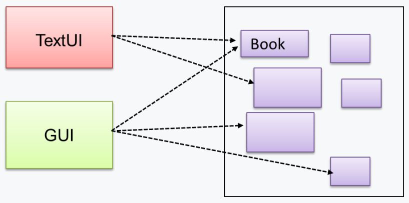
What

★★★☆ Can explain the Facade design pattern

Context

Components need to access functionality deep inside other components.

The `UI` component of a `Library` system might want to access functionality of the `Book` class contained inside the `Logic` component.



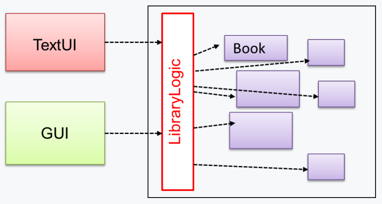
Problem

Access to the component should be allowed without exposing its internal details. e.g. the `UI` component should access the functionality of the `Logic` component without knowing that it contains a `Book` class within it.

Solution

Include a Façade class that sits between the component internals and users of the component such that all access to the component happens through the Facade class.

The following class diagram applies the Facade pattern to the `Library System` example. The `LibraryLogic` class is the Facade class.



Exercises

Command pattern

What

★★★☆ Can explain the Command design 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

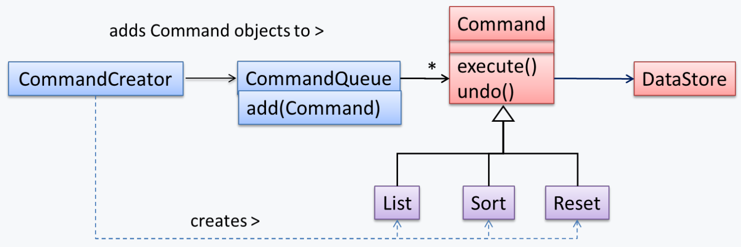
It is preferable that some part of the code executes these commands without having to know each command type. e.g., there can be a `CommandQueue` object that is responsible for queuing commands and executing them without knowledge of what each command does.

Solution

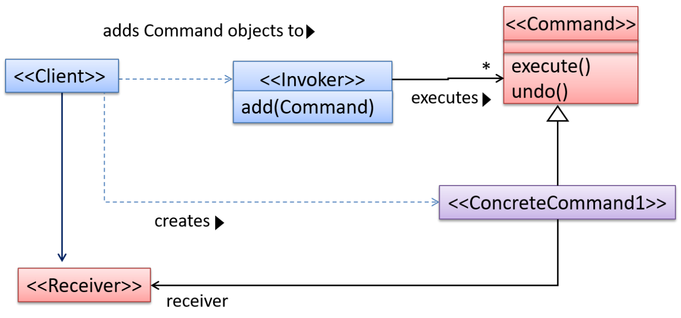
The essential element of this pattern is to have a general `<<Command>>` object that can be passed around, stored, executed, etc without knowing the type of command (i.e. via polymorphism).

Let us examine an example application of the pattern first:

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.



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.

Model view controller (MVC) pattern

What

★★☆☆ Can explain the Model View Controller (MVC) design 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

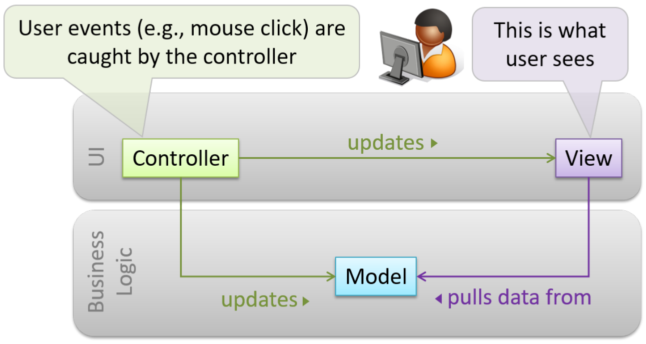
The high coupling that can result from the interlinked nature of the features described above.

Solution

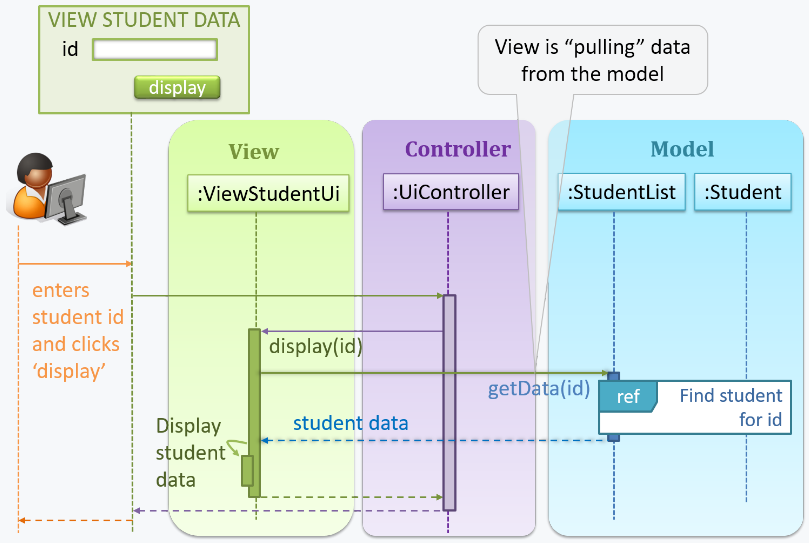
Decouple data, presentation, and control logic of an application by separating them into three different components: *Model*, *View* and *Controller*.

- *View*: Displays data, interacts with the user, and pulls data from the model if necessary.
- *Controller*: Detects UI events such as mouse clicks and button pushes, and takes follow up action. Updates/changes the model/view when necessary.
- *Model*: Stores and maintains data. Updates the view 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 the data of a 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`. The `ref` frame indicates that the interactions within that frame have been extracted out to another separate sequence diagram.

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

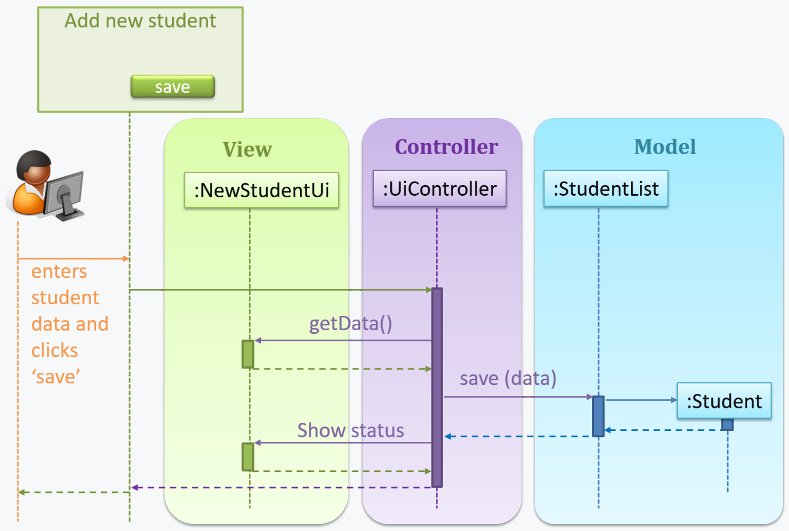
What

★★★☆ Can explain the Observer design pattern

Context

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

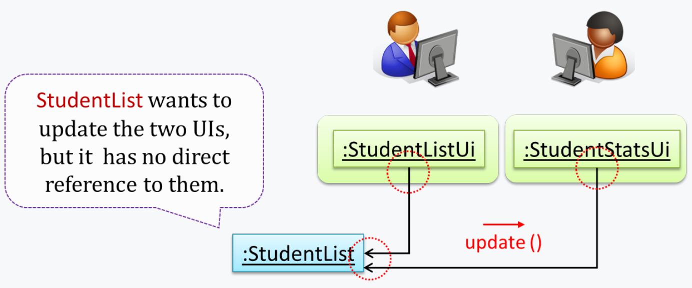
Consider this scenario from a student management system where the user is adding a new student to the system.



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

- `StudentListUi`: that accesses a list of students and
- `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.


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

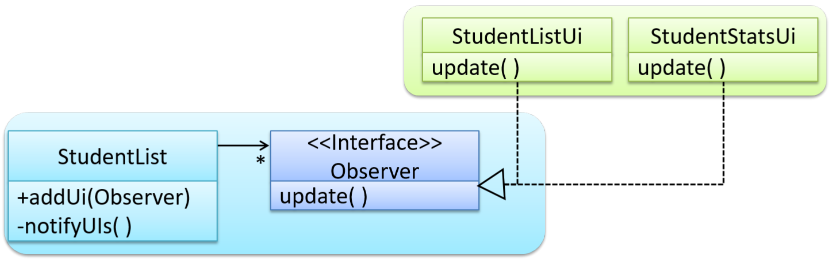
However, the `StudentList` object has no knowledge about `StudentListUi` and `StudentStatsUi` (note the direction of the navigability) and has no way to inform those objects. This is an example of the type of problem addressed by the Observer pattern.

Problem

The 'observed' object does not want to be coupled to objects that are 'observing' it.

Solution

Force the communication through an interface known to both parties.



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

During the initialization of the system,

1. First, create the relevant objects.

```java
StudentList studentList = new StudentList();
StudentListUi listUi = new StudentListUi();
StudentStatsUi statsUi = 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'.

```java
studentList.addUi(listUi);
studentList.addUi(statsUi);
```

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

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

Now, whenever the data in `StudentList` changes (e.g. when a new student is added to the `StudentList`),

1. All interested observers are updated by calling the `notifyUIs` operation.

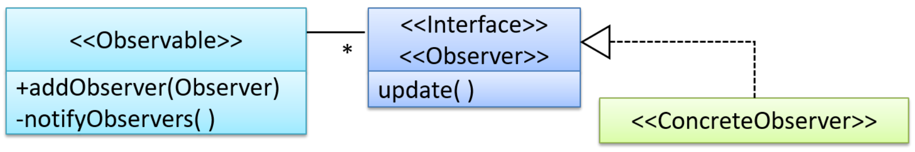
```java
// StudentList class
public void notifyUIs() {
    // for each observer in the list
    for (Observer o: observerList) {
        o.update();
    }
}
```

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

```java
// 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 have inbuilt support for the Observer pattern.

Exercises

More

Combining Design Patterns

★★★★ Can combine multiple patterns to fit a context

Design patterns are usually embedded in a larger design and sometimes applied in combination with other design patterns.

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.

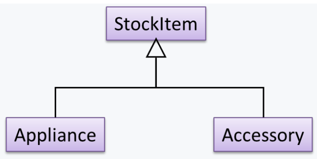
Use Cases:

- 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

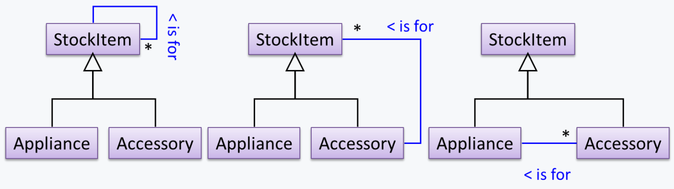
SIS can be accessed using multiple terminals. Shop assistants use their own terminals to access SIS, while the shop manager's terminal continuously displays a list of all items in the store. In the future, it is expected that suppliers of items use their own applications to connect to SIS to get real-time information about the 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.

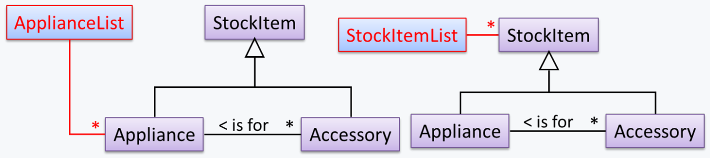
A `StockItem` can be an Appliance or an Accessory.



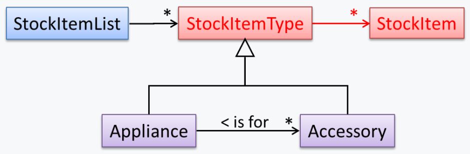
To track that each `Accessory` is associated with the correct `Appliance`, consider the following alternative class structures.



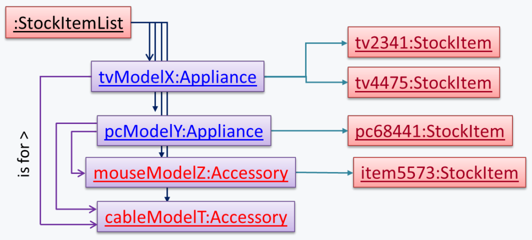
The third one seems more appropriate (the second one is suitable if accessories can have accessories). Next, consider between keeping a list of `Appliances`, and a list of `StockItems`. Which is more appropriate?



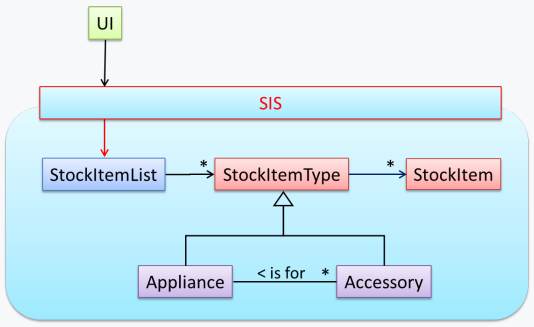
The latter seems more suitable because it can handle both appliances and accessories the same way. Next, an abstraction occurrence pattern is applied to keep track of `StockItems`.



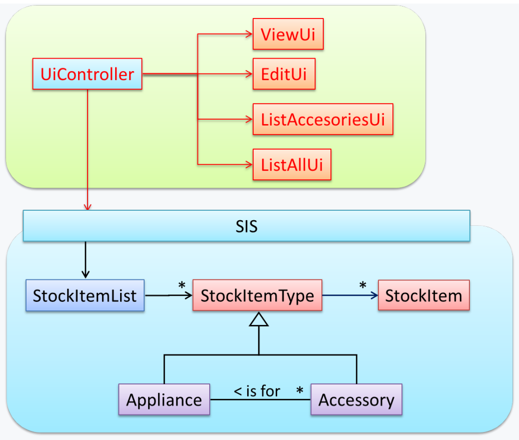
Note the inclusion of navigabilities. Here's a sample object diagram based on the class model created thus far.



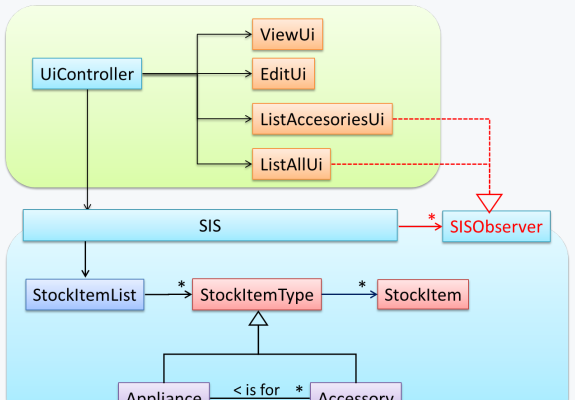
Next, apply the façade pattern to shield the SIS internals from the UI.



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



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



Other Design Patterns

★★★★ 🏆 Can recognize some of the GoF design patterns

The most famous source of design patterns is the "Gang of Four" (GoF) book which contains 23 design patterns divided into three categories:

- **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

Using Design Patterns

★★★★ 🏆 Can explain pros and cons of design patterns

Design patterns provide a high-level vocabulary to talk about design.

📦 Someone can say 'apply Observer pattern here' instead of having to describe the mechanics of the solution in detail.

Knowing more patterns is a way to become more 'experienced'. Aim to learn at least the context and the problem of patterns so that when you encounter those problems you know where to look for a solution.

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.

Be careful not to overuse patterns. 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).

Other Types of Patterns

★★★★ Can explain how patterns exist beyond the domain of software design

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 *The 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 that needs to be solved frequently 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.

Exercises

Design Patterns vs Design Principles

★★★★ Can differentiate between design patterns and principles

Design *principles* have varying degrees of formality – rules, opinions, rules of thumb, observations, and axioms. Compared to design patterns, principles are **more general**, have **wider applicability**, with correspondingly **greater overlap** among them.