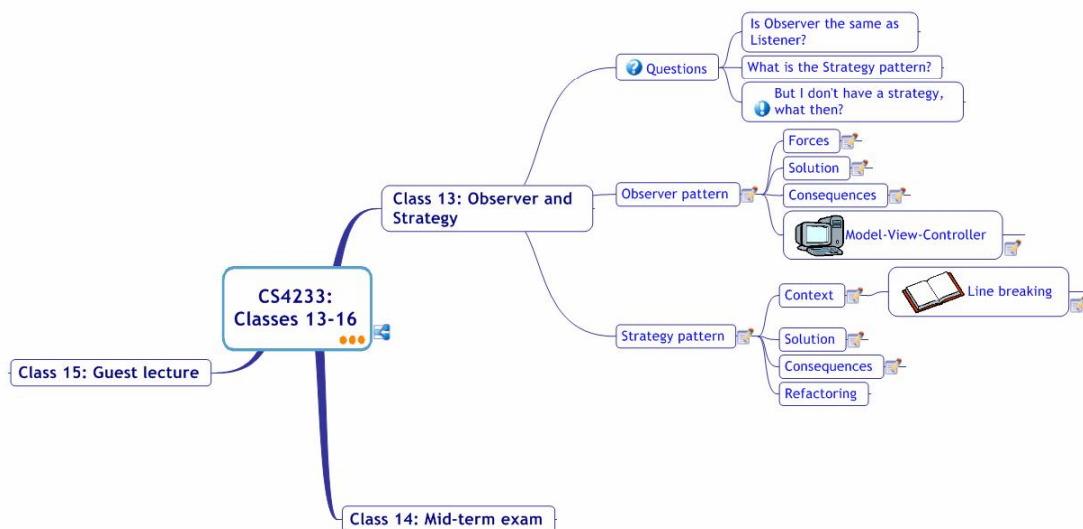
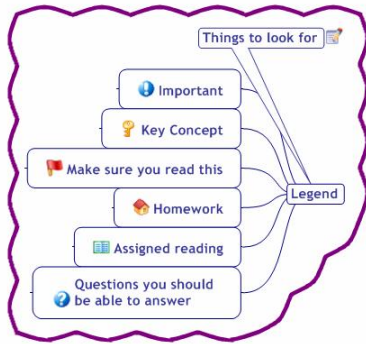


CS4233: Classes 13-16



See document: [CS4233 Object-Oriented Analysis and Design.mmap](#)

1 Class 13: Observer and Strategy

1.1 Questions

1.1.1 Is Observer the same as Listener?

1.1.2 What is the Strategy pattern?

1.1.3 But I don't have a strategy, what then?

1.2 Observer pattern

You have seen the Observer pattern several times during this course so far. The listeners that are used in the project are applications of the Observer pattern. The Java API calls classes that observe other classes, *listeners*.

The main purpose of the Observer pattern is to allow an object to tell other objects about events of interest.

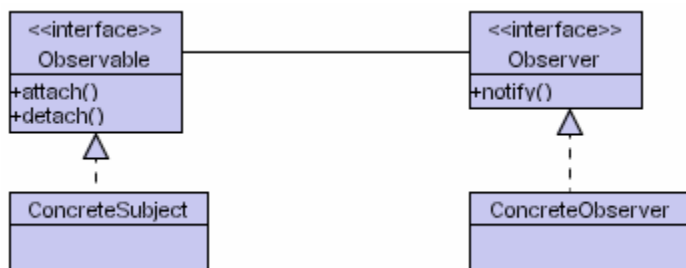
1.2.1 Forces

The context of the Observer pattern, and its synopsis should be obvious by now. The forces are, however, interesting and not so obvious. The two forces for that would guide you to choosing the Observer are:

- You have two independent objects that are not specifically intended to work with each other. In fact, loose coupling, and high cohesion dictate that they should not know about each other. However, one needs to notify the other of events and state changes.
- You have a one-to-many dependency relationship that may require an object to notify multiple objects of changes or events.

1.2.2 Solution

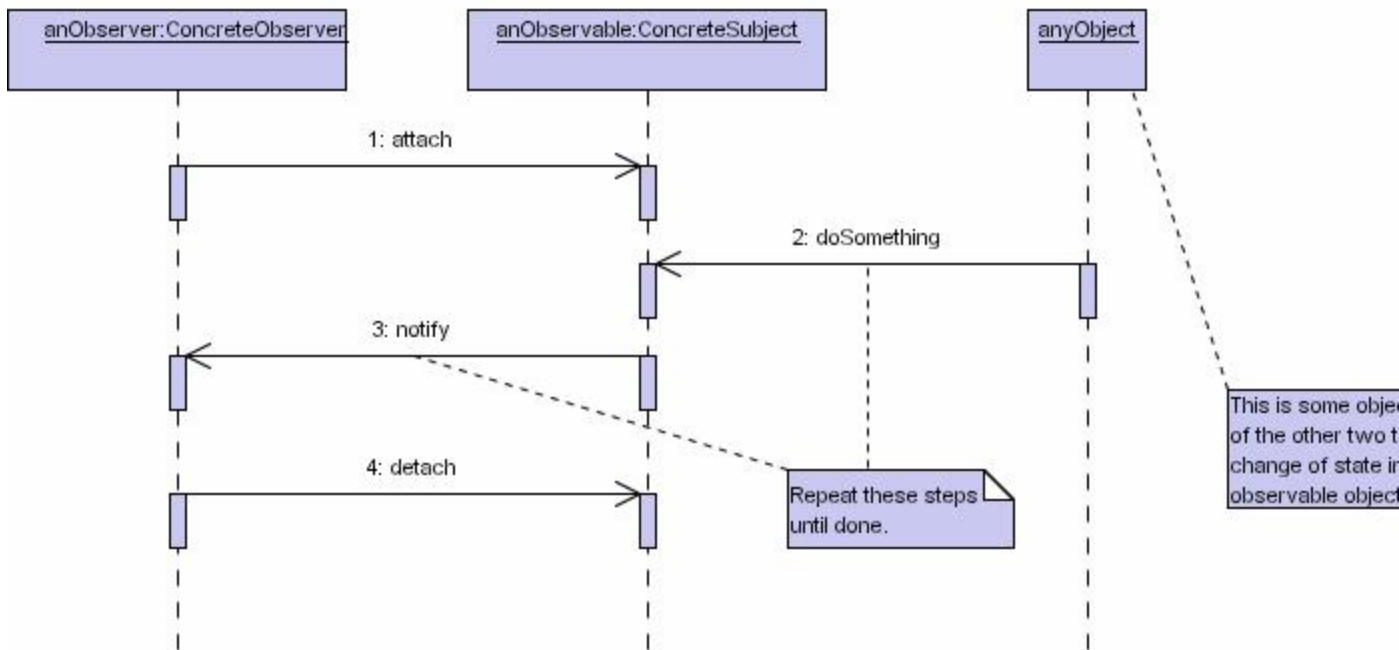
The UML class diagram for the Observable pattern looks like this, although there are several variations.



The ConcreteSubject knows its observers. When an event occurs that warrants notifying the observers, the ConcreteSubject calls the notify() method on each ConcreteObserver.

The ConcreteObserver must be attached and detached to the ConcreteSubject. It is not necessarily the responsibility of the ConcreteObserver to perform these operations, but some object must do it.

The basic sequence of how the observer and observed objects interact is shown in the following sequence diagram:



1.2.3 Consequences

The benefits of using the Observer are:

- There is an abstract coupling between the objects. All the observable object knows is that there is an observer and how to notify it. It doesn't know the type of the observer or anything else about it. *The two objects can belong to two different layers of abstraction* in the system.
- Observers can be used to support broadcasting messages to a series of objects.

Possible disadvantage of the Observer pattern are:

- There are unexpected updates. The cost of changing the state of the observable object can be quite a lot if there are many observers that each performs a lot of computation.
- Finding errors and debugging when the Observer pattern is applied can be difficult. One particularly egregious error is when there are cyclic notification loops and an event notification is propagated endlessly.

1.2.4 Model-View-Controller

There is an architectural style known as Model- View-Controller. It is one of the best known examples of the Observer pattern. In the MVC, the model contains the state of the system. When it is changed, the view objects are notified by the controller. The model is the observable part and the view is the observer part of this style.

1.3 Strategy pattern

The Strategy pattern is one that is used often, especially in applications like AI programs, games, and so on, where different algorithms can be used to have different behaviors present. The purpose of the pattern is to decouple the strategy from the implementation so that you can "plug-and-play" different algorithms.

Strategy is also referred to as *policy*. There are many times where teams will talk about implementing a policy, like a licensing policy, and so on. This can be done quite effectively with the Strategy pattern.

1.3.1 Context

The textbook uses an example of layout managers to lay out different organizations of a telephone keypad. This is one UI-based example, and not necessarily the most illustrative one.

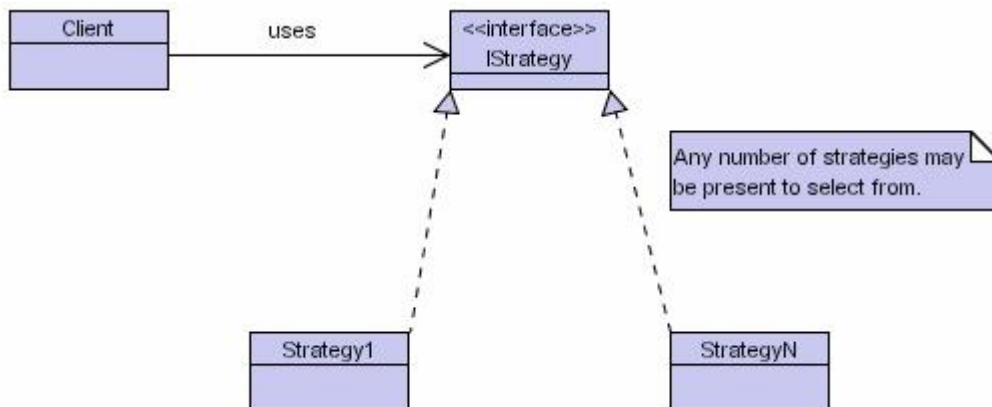
A better example is given in the Gang of Four book. This is described in the following section.

Line breaking

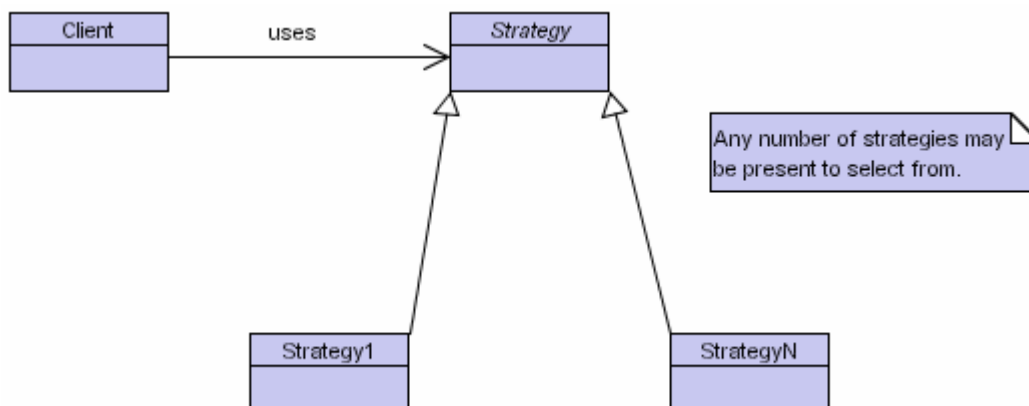
There are many algorithms for dividing a text stream into lines. If you were building an editor, you would want to allow your users to be able to decide which line-breaking strategy (policy) to implement. This is done by using the Strategy pattern. In programs that provide many options to a user, the Strategy pattern will be used throughout to provide different user options.

1.3.2 Solution

There are two basic solutions to implementing the Strategy pattern. The most general is:



Often, you will have methods that have common implementations among the strategies, in which case the second implementation is used as shown:



1.3.3 Consequences

Applying the Strategy pattern provides a clean design. It enables you to eliminate conditional statements that might be spread about your code (one branch for each strategy, at each decision point).

Strategy also gives you the ability to provide, or select a choice of implementation without committing to it too soon.

There are a couple of drawbacks to the Strategy pattern.

- Clients may have to be aware of different strategies.
- There is an additional communication overhead, but often it is not much more than will be used with simple delegation.
- There may be an increased number of dependent objects.

1.3.4 Refactoring

2 Class 14: Mid-term exam

3 Class 15: Guest lecture