Chapter 18: Design Patterns

Design patterns are elegant, adaptable, and reusable solutions to everyday software development problems. Each pattern includes a description of a commonly occuring type of problem, a design for a set of classes and class relationships that solve that problem, and reasons why the given solution is wise.

18.1 Definitions and General OO Principles

18.1.1 Definitions

1. Subclass: X is a subclass of Y if X is derived from Y directly or indirectly.
2. Collaboration: two or more objects that participate in a client/server relationship in order to provide a service.
3. Coupling: A dependency between program elements (such as classes) typically resulting from collaboration between them to provide a service. Classes X and Y are coupled if...

   - X has a function with parameter or local variable of class Y.
   - X has a data member that points at something of class Y.
   - X is a subclass of Y.
   - X implements an interface for class Y (Y gives friendship to X).

   [Example:] If the Key class calculates the hash-table index, it must know the length of the hash table. But this couples two classes that would not otherwise be coupled.
4. Cohesion: This is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion is a "specialist" with narrow power.
5. System event: A high-level event generated by an external actor; an external input event. For each system event, there is a corresponding operation. For example, when a word-processor user hits the "spell check" button, he is generating a system event indication "perform spell check".
6. Use case: The sequence of events and actions that occur when a user participates in a dialog with a system during a meaningful process.

18.2 General OO Principles

1. Encapsulation. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another.
2. Narrow interface. Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. [Why] This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.
3. Delegation: a class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert. [Example:] HashTable::find selects one list and delegates the searching task to List::find.

18.3 Patterns

A pattern is a design issue or communication problem... with a solution based on class structure... and guidance on how to apply the solution in a variety of contexts.

18.3.1 GRASP: General Responsibility Assignment Software Patterns

- High cohesion is desirable. [Why?] It makes a class easier to comprehend, easier to maintain, and easier to reuse. The class will also be less affected by change in other classes. [Example:] Cohesion is low if a HashTable class contains code to extract fields from a data record. Cohesion is low if the data class computes a hash index. Cohesion is high if each class does part of the process.
- Low coupling is desirable. Which class should be given responsibility for a task? [A] Assign a responsibility so that its placement does not increase coupling. [Why?] High coupling makes a class harder to understand, harder to reuse, and harder to maintain because changes in related classes force local changes.
- Expert. Who should do what? [A] Each class should do for itself actions that involve its data members. Each class should "take care of" itself and handle its own emergencies. [Why] This minimizes coupling.
- Creator. Who should create (allocate) an object? [A] The class that composes, aggregates or contains it. [Why] This minimizes coupling.

  Who should delete (deallocate) an object? [A] The class that created it. [Why] To minimize confusion, and because, often, nothing else is possible.
- Don't Talk to Strangers. That is, don't "send messages" to objects that are not close to you. Non-strangers are:

  – your own data members.
  – elements of a collection which is one of your own data members.
  – a parameter of the current function.
  – a locally-created object.
  – `this` (but using `this` is rarely the right thing to do).

  Delegate the operation [Why] This is a generalization of the old rule, don't use globals. It maximizes the locality of every reference and avoids unnecessary coupling between classes.

  [Example] Don't deal directly with a component of one of your own members. Suppose a hardware store class composes an object of class Inventory, and the Inventory is a flex-array of Item pointers, as shown below. The retail store would be talking to a stranger if its sell function called the sell function in the Item class directly, like this: `Inv.find(currentKey)->sell(5);` The preferred design is to work through the intermediate class. That is, Inventory should provide a function such as `sell(key, int)` that can be called by RetailStore, and `Inventory::sell(key, int)` should call `Inventory::find(key)` followed by `Item::sell(int)`. Evaluation: In the first design, a change in the Item class might force a change in both RetailStore and Inventory. Using the second design, only Inventory is affected.

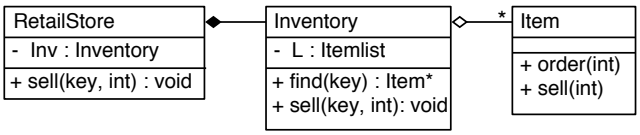

Figure 18.1: Don't talk to strangers.

18.4 More Complex Design Patterns

- Adapter. Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires. [Solution:] Define an adapter class that can add, subtract, or override functionality, where necessary. There are two ways to do this; on the left is a class adapter, on the right an object adapter.

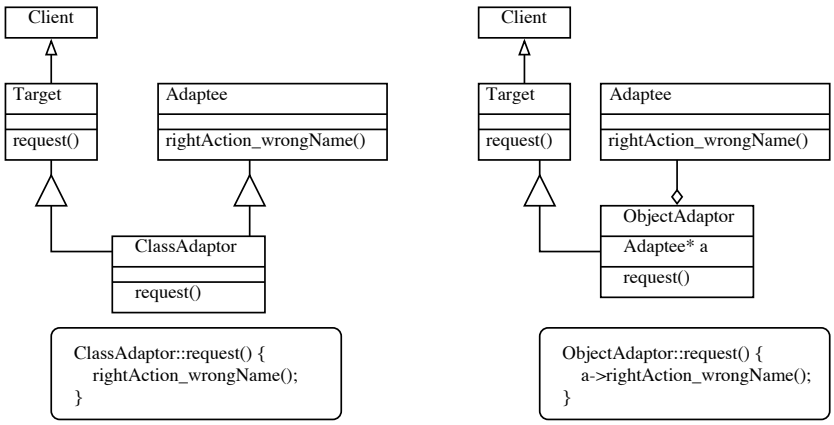

Figure 18.2: Two kinds of adapters.

- Indirection. This pattern is used to decouple the application from the implementation where an implementation depends on the interface of some low-level device. [Why] To make the application stable, even if the device changes.

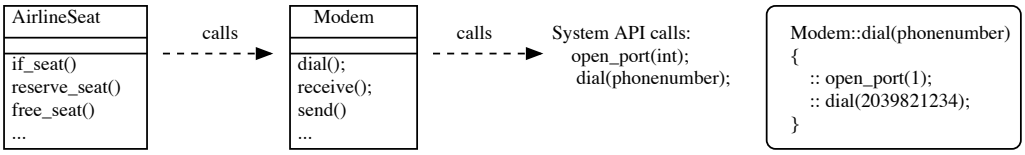

Figure 18.3: Indirect coupling to an unstable interface.

- Proxy. This pattern is like Indirection, and is used when direct access to a component is not desired or possible. What to do? [Solution:] Provide a placeholder that represents the inaccessible component to control access to it and interact with it. The placeholder is a local software class. Give it responsibility for communicating with the real component. [Special cases:] Device proxy, remote proxy. In Remote Proxy, the system must communicate with an object in another address space.

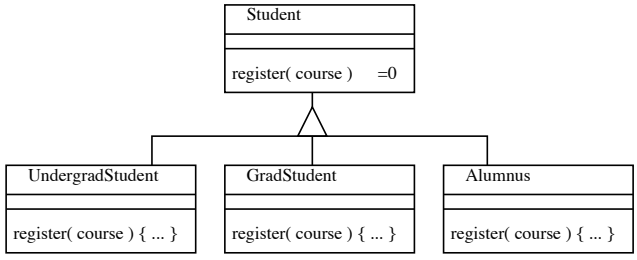

Figure 18.4: Polymorphic implementation of an abstraction.

- Polymorphism: In an application where the abstraction has more than one implementation, define an abstract base class and one or more subclasses. Let the subclasses implement the abstract operations. [Why] to decouple the implementation from the abstraction and allow multiple implementations to be introduced, as needed.
- Controller: Who should be responsible for handling a system event? [A] A controller class. The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

  Factors such as the number of events to be handled, cohesion and coupling should be used to decide among the three kinds of controllers described below and to decide how many controllers there should be. A controller class represents one of the following choices:

  – The overall application, business, or organization (facade controller).

  - Something in the real world that is active that might be involved in the task (role controller). [Example:] a menu handler.
  - An artificial handler of all system events involved in a given use case (use-case controller). [Example:] A retail system might have separate controllers for BuyItem and ReturnItem.

- Bridge: This pattern is a generalization of the Indirection pattern, used when both the application class and the implementation class are (or might be) polymorphic. The bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments. In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.

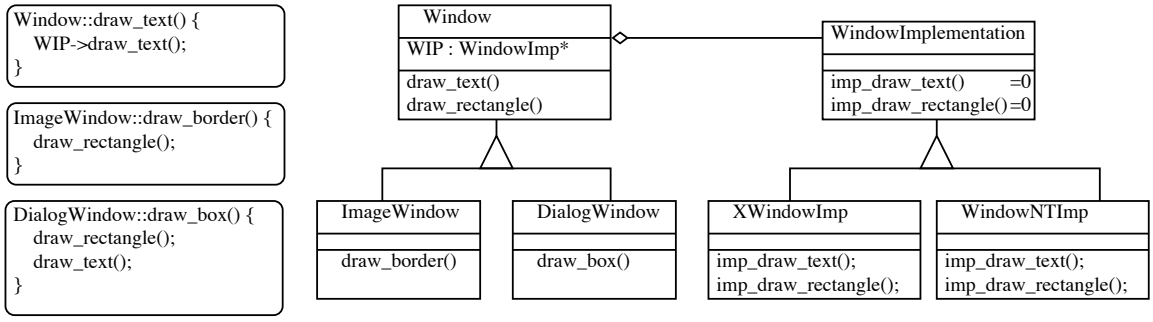

Figure 18.5: Bridging the implementation gap.

- Subject-Observer or Publish-Subscribe: Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

  [Example:] An OO spreadsheet application contains a data object, several presentation "views" of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

  [Solution:] In the following discussion, the SpreadsheetData class is the subject, the views and graphs are the observers. The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects from its list. Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes. We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a `get_state()` function, that returns a copy of its state.

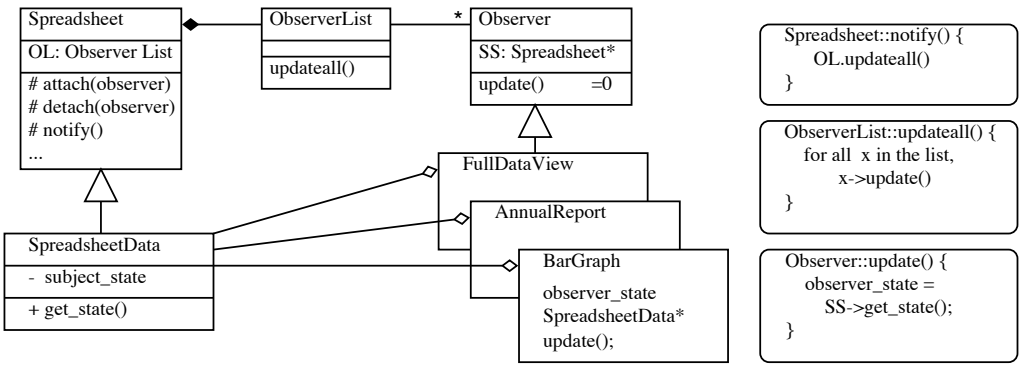

Figure 18.6: One subject with three observers

The ObserverList class defines an updating interface for objects that should be notified of changes in a subject. The Observer class provides an abstract public function called update() which will be called by

ObserverList whenever updateall() is called. This abstract function must be implemented in each concrete observer class.

When the state of the SpreadsheetData subject changes, it executes its inherited notify() function, which calls `ObserverList::updateall()`, which notifies all of the observers. Each one, in turn, executes its update function, which calls the subject's get_state function. Changes can then be made locally that reflect the change in the subject's state.

- Singleton: Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance. [Solution:] A single object may be made available to all objects of class C by making the singleton a static member of class C. A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.

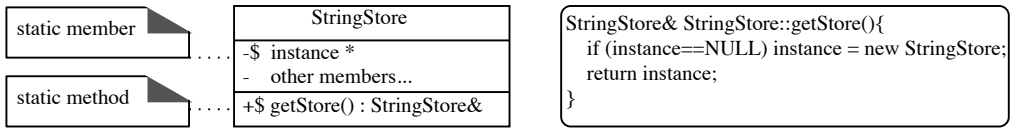

Figure 18.7: How to create a singleton.

[Example] Suppose there were several parts of a program that could use a StringStore. We might define StringStore as a singleton class. The `StringStore::put` function would be made static and would become a global access point to the class, while maintaining full protection for the class members.

More patterns? The seven patterns presented here are some of the earliest and most useful that have been developed. But many, many more design patterns have been identified and published. A professional working in either C++ or Java would do well to study some of the available literature.