CPSC 427: Object-Oriented Programming

Michael J. Fischer

(slides by Alice E. Fischer)

Lecture 24
December 5, 2016

General OO Principles

General OO principles

1. Encapsulation
2. Expert
3. Delegation
4. Narrow Interface
5. Creator
6. Low Coupling
7. High Cohesion
8. Don't Talk to Strangers
9. Polymorphism
10. Chain of Responsibility

Basic Principles for OO Design

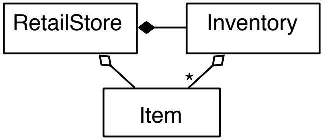
These are recognized as important fundamental design principles.

- Encapsulation: data members should be private. Accessors should be defined only when necessary.
- Expert: Each class should do for itself all actions that involve its data members.
- Delegation: Delegate all actions to the class that is the expert on the data.
- Narrow interface: Keep the set of public functions as simple as possible. Functions that are not needed by client classes should be private.
- Creator: Allocate and initialize an object in the class that composes, aggregates, or contains it.

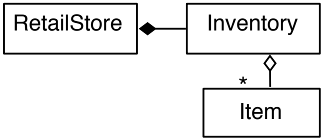
Low coupling

A UML diagram contains links between classes. The number of links should be minimized.

When assigning responsibility for a task to a class, assign it so that the placement does not increase coupling.



Bad: Unnecessary coupling



Good: Minimal coupling

High cohesion

A class should have a single, narrow purpose.

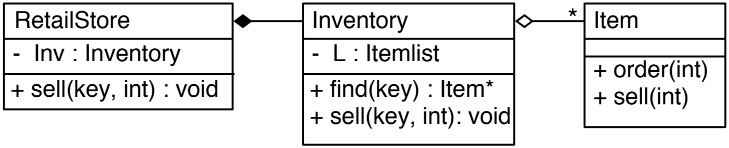
- The members of the class should be strongly related and focused on the purpose and responsibilities of the class.
- Don't let the classes "sprawl" by adding more and more specialized features.
- Maintain the separation of purposes: Define structural elements and semantic elements in separate classes.
- Example: If you want a linked list of books, define a book class and define a pair classes, List and Cell. Don't define the members of a book in the Cell class.

Don't talk to strangers!

Principle: The class A should only call functions in class B if you can see the class name B when looking at the header file of A.

Reason: Program maintenance is difficult when there are hidden dependencies.

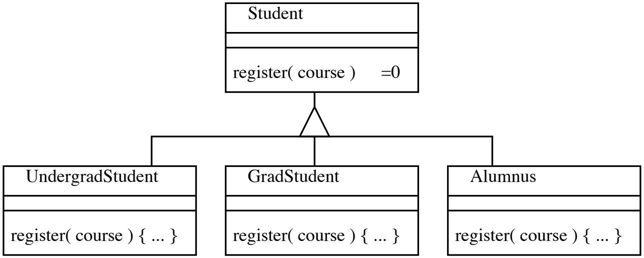
In the diagram, RetailStore should not be calling the `Item::sell` because there is no mention of the Item class in the RetailStore class definition. Instead, RetailStore should call a function in Inventory and let Inventory figure out how to handle its Items.



Polymorphism

Principle: Use polymorphism (derivation + virtual functions) to implement a set of related but not identical classes.

Reason: This lets the programmer create a common stable interface for dealing with all variations, and also avoids duplicating blocks of code. Coding, debugging, and program maintenance all become easier.


Student
register(course) =0
UndergradStudent
register(course) { ... }
GradStudent
register(course) { ... }
Alumnus
register(course) { ... }

Chain of Responsibility

Objects created by declarations are stored on the run-time stack.

- These objects are deleted automatically when control leaves the declaring block.
- When an object is put into an array or an STL container (such as vector), it is copied. The array or vector becomes the "owner" and will manage the storage.

Objects created using new are the programmer's responsibility.

- There must be a clearly defined "owner" of every dynamically created object.
- When an object pointer is put into a container (an array or vector), the "owner" is the class that declares the container.
- The owner is responsible for deleting the object at the end of its useful lifetime.

Design Patterns

What is a design pattern?

A pattern has four essential elements.[1]

1. A *pattern name*.
2. The *problem*, which describes when to apply the pattern.
3. The *solution*, which describes the elements, relations, and responsibilities.
4. The *consequences*, which are the results and tradeoffs.

[1] Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995.)

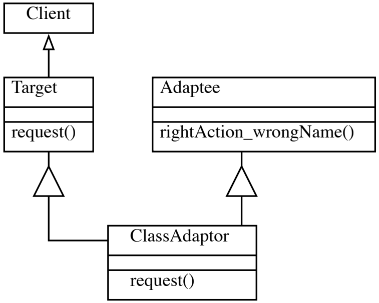
Adaptor pattern

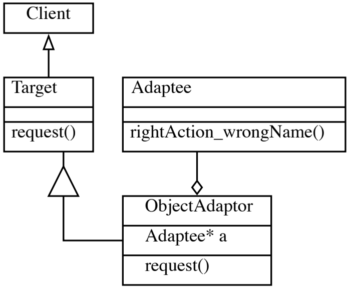
Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires.

Solution: Define an adapter class that can add, subtract, or override functionality, where necessary.

Adaptor diagram

There are two ways to do this; on the left is a class adapter, on the right an object adapter.





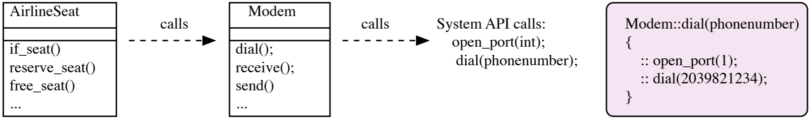
```
ClassAdaptor::request() {
   rightAction_wrongName();
}
```

```
ObjectAdaptor::request() {
   a->rightAction_wrongName();
}
```

Indirection

This pattern is used to decouple the application from the implementation, where an implementation depends on the interface of some low-level device.

Goal is to make the application stable, even if the device changes.



```
Modem::dial(phonenumber)
{
   :: open_port(1);
   :: dial(2039821234);
}
```

Proxy pattern

This pattern is like Indirection, and is used when direct access to a component is not desired or possible.

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.

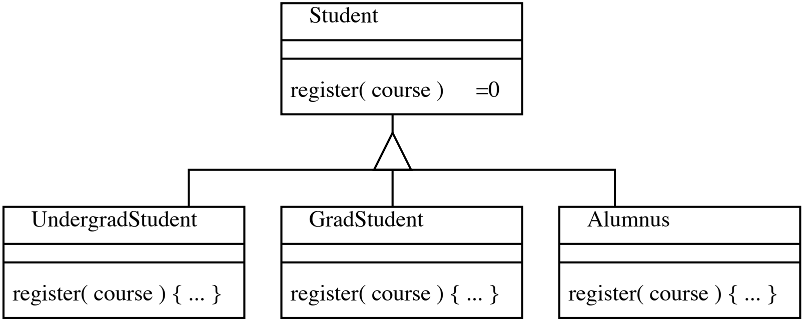
Polymorphism pattern

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.

This decouples the implementation from the abstraction and allows multiple implementations to be introduced, as needed.

Polymorphism diagram


Student
register(course) =0
UndergradStudent
register(course) { ... }
GradStudent
register(course) { ... }
Alumnus
register(course) { ... }

Controller

A controller class takes responsibility for handling a system event.

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.

Three kinds of controllers

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.

Choose among these according to the number of events to be handled, cohesion and coupling, and to decide how many controllers there should be.

Bridge pattern

Bridge generalizes the Indirection pattern.

It is used when both the application class and the implementation class are (or might be) polymorphic.

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.

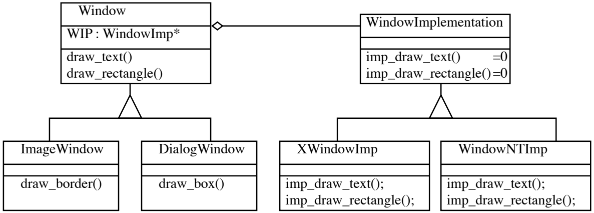
Bridge diagram

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.

```
Window::draw_text() {
   WIP->imp_draw_text();
}
```

```
ImageWindow::draw_border() {
   draw_rectangle();
}
```

```
DialogWindow::draw_box() {
   draw_rectangle();
   draw_text();
}
```



Subject-Observer or Publish-Subscribe: problem

Problem: 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.

Subject-Observer or Publish-Subscribe: pattern

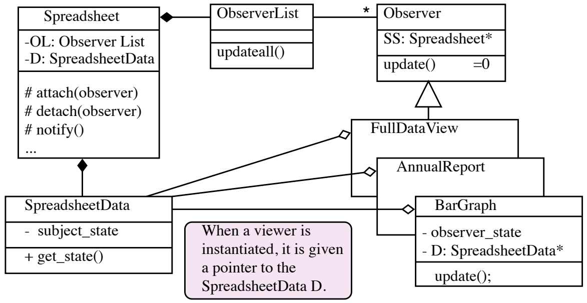
Call the SpreadsheetData class the **subject**; the views and graphs are the **observers**.

The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects.

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.

Subject-Observer or Publish-Subscribe: diagram



```
Spreadsheet::notify() {
    OL.updateall()
}
```

```
ObserverList::updateall() {
    for all x in the list,
        x->update()
}
```

```
Observer::update() {
    observer_state =
        D->get_state();
}
```

```
Spreadsheet::attach( ob ) {
    OL.add (ob );
}
```

See textbook for more details.

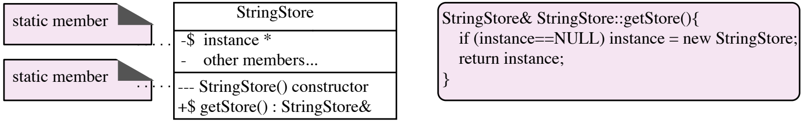
Singleton pattern

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.

StringStore example



```
StringStore& StringStore::getStore(){
   if (instance==NULL) instance = new StringStore;
   return instance;
}
```

Example: Suppose several parts of a program need to use a StringStore. We might define StringStore as a singleton class.

The `StringStore::put()` function is made `static` and becomes a global access point to the class, while maintaining full protection for the class members.