CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 25
December 5, 2018

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

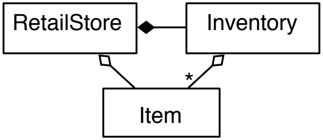
These are recognized as important fundamental design principles.

- 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.

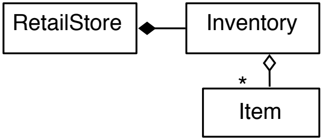
Low coupling

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.

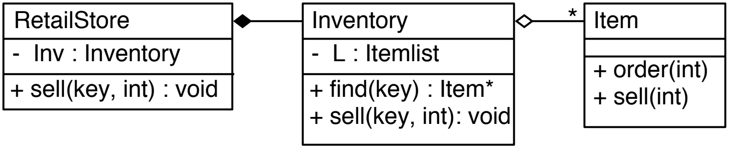
- The members of the class should be strongly related and focused on the purpose and responsibilities of the class.
- Don't let the classes "sprawl" by adding more and more specialized features.
- Maintain the separation of purposes: Define structural elements and semantic elements in separate classes.
- 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.

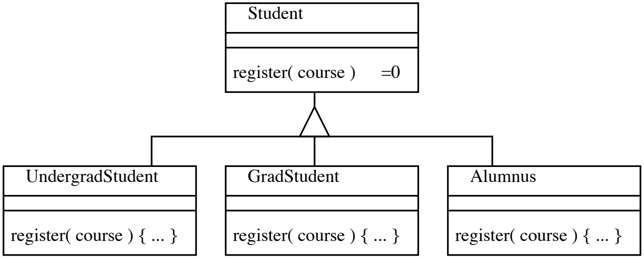
In the diagram, RetailStore should not be calling the `Item::sell` because there is no mention of the Item class in the RetailStore class definition. Instead, RetailStore should call a function in Inventory and let Inventory figure out how to handle its Items.



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 or moved. 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.

Function-Like Constructs

Named functions

C and C++ support named global functions. Declaration syntax is `return_type name( parameters...)  {function_body}`

C++ also permits named member functions. These are declared inside of a class using the same syntax as for named global functions, except an optional `const` keyword following the parameter list.

The function name is qualified by the class name. Thus, the name of `f` within class `MyClass` is `MyClass::f`

Function calls

Syntax for calling a named global function:
`name( arguments...)`

A function call is an **expression** whose type is the `return_type` of the function.

Syntax for calling a named member function of class `MyClass` is
`implicit_arg.name( explicit_arguments...  )`

The implicit argument must be an object of type `MyClass`. If called from within `MyClass`, the implicit argument defaults to `*this`.

Function types

The type of a named global function is its **signature**, that is, the return type and the list of parameter types.

For example, a function that takes a double and an integer parameter and returns a long integer has signature

```
long int (&) (double, int&)
```

To define such a function, we would add names for the function and parameters and a body to perform the computation.

```
long int (myfun) (double d, int& k) {
    return (k += 2.5*d);
}
```

The parens around `myfun` are optional and would normally be omitted.

`typedef` for function types

Function types can be named using `typedef`. The declaration

```
    typedef long int (myFunType) (double, int&);
```

defines the type name `myFunType`.

One can then declare a function to be of that type.

```
    myFunType myfun;
```

and use it to define an alias `myfun2` for `myfun`:

```
    myFunType& myfun2 = myfun;
```

Without the `typedef`, it would look like this:

```
    long int (&myfun2) (double, int&) = myfun;
```

(See demo `25-Functors/funtypes.cpp`.)

A new way to give names to types

C++ now provides another syntax for defining new type names (since c++11). Instead of

```
typedef long int (myFunType) (double, int&);
```

one can write

```
using myFunType = long int (&) (double, int&);
```

(See demo 25-Functors/using.cpp.)

Function pointers

One can have pointers to functions. Function points can be passed as the argument to functions that take function parameters such as the standard template function `sort()`. The third argument is the comparison function `comp` to be used when comparing elements. It returns `true` if its first argument is "smaller" than its second.

```
bool descendingOrder( int i, int j)
    { return (j<i); }
using Comp = bool (*) (int, int);
Comp cptr;
cptr = descendingOrder;
sort (myvector.begin(), myvector.end(), cptr);
```

(See demo `25-Functors/sort-funptr.cpp`.)

Anonymous functions (a.k.a. lambda functions)

C++ now has anonymous functions. The previous example could be rewritten as

```
    sort (myvector.begin(), myvector.end(),
          [] (int i, int j) { return (j<i);} );
```

(See demo 25-Functors/sort-lambda.cpp.)

Functors

A **functor** an instance of a class that defines `operator()`. It can be called using function syntax. Here's how it can be used as a sort comparator.

```
class ComparAscending {
public:
   bool operator()(int i, int j) { return (i<j); }
};
...
sort( myvector.begin(), myvector.end()-4,
      CompareAscending() );
```

(See demo `25-Functors/sort-functor.cpp`.)

Closures

Functors and lambda expressions gain their power through closures.

A **closure** is an expression in which some of the variables have been **bound** to values, whereas other remain as function parameters.

A closure can be used like any other function.

A functor with data members acts like a closure. It result may depend on the values of those data members. Different instantiations of the functor class may result in functors with different behaviors since the data members may have different values.

Lambda capture

A lambda expression has the (simplified) syntax:
`[capture_list] (parameters...)  -> result_type {body}`

When evaluated, the result is a closure with the variables on the capture list being "imported" into the body.

In fact, when a lambda expression is evaluated, the result is a functor of a new compiler-generated class.

The `function` template in header file `<functional>` can be used to turn a closure into an object of a function type.

Closure example

Demo `25-Functors/closure.cpp` uses `map`, `function`, and closures with value-capture to define a table of sales tax functions.

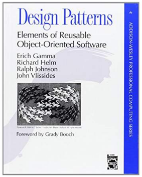
The map key is a state name (or abbreviation), stored as a `string`. The map value is a tax computation function that takes a sales amount as its argument and returns the sales tax. It uses the captured variable `rate` to computer the tax.

Each map pair has a tax computation function that was generated at run time to use a particular tax rate in its computation. The rate capture takes place in the line

```
taxFunType taxfun = [rate](double amt) { return amt*rate; };
```

Design Patterns

What is a design pattern?


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison-Wesley Professional Computing Series

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.

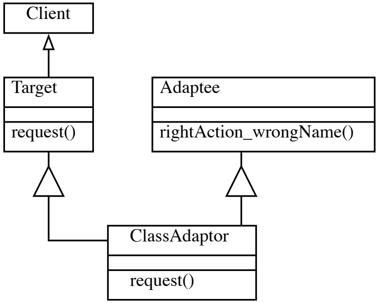
Adaptor pattern

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

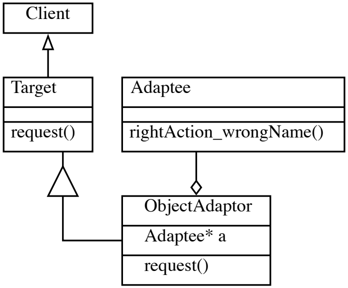
Solution: Define an adapter class that can add, subtract, or override functionality, where necessary.

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();
}
```

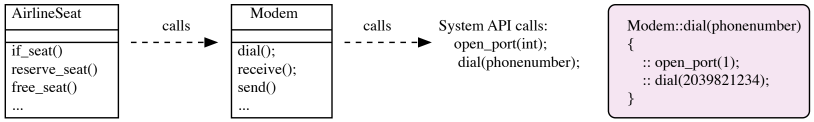


```
ObjectAdaptor::request() {
   a->rightAction_wrongName();
}
```

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);
}
```

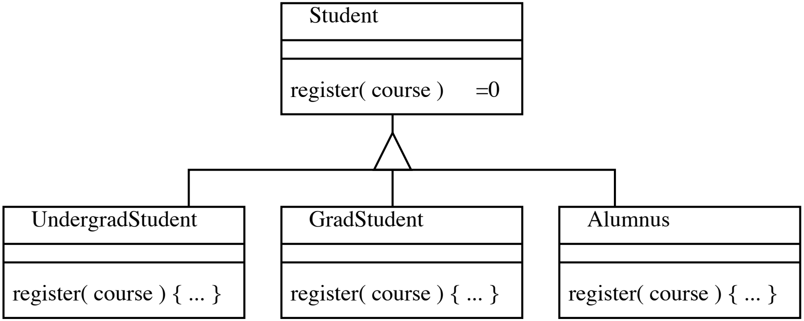
Polymorphism pattern

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) { ... }

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.

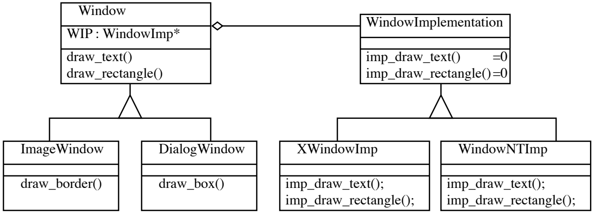
Bridge diagram

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

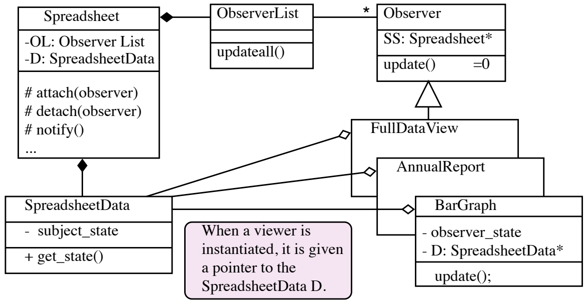
Call the SpreadsheetData class the **subject**; the views and graphs are the **observers**.

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.