CS193j, Stanford
Winter, 2002-03

Handout #6
Nick Parlante

OOP Design

OOP Design #1 -- Encapsulation

The most basic idea in OOP is that each object encapsulates some data and code. The object takes requests from other client objects, but does not expose its the details of its data or code to them. The object alone is responsible for its own state, exposing public messages for clients, and declaring private the ivars and methods that make up its implementation. The client depends on the (hopefully) simple public interface, and does not know about or depend on the details of the implementation.

For example, a HashTable object will take get() and set() requests from other objects, but does not expose its internal hash table data structures or the code strategies that it uses. The theme here is one of separation -- trying to keep the complexity inside one object from interfering with other objects. Or put another way, trying to keep the various objects as independent from each other as possible.

Each object provides some service or "interface" for the other objects. Ideally, the service is exposed in a way that is simple for the other objects to understand. The complexity of the implementation still exists, but it is isolated inside the one class. This works because for most problems there are all sorts of details of the implementation that the clients don't really care about -- how the hash buckets are arranged for example. Inevitably, the interface that captures just the issues relevant to the client is much simpler than the full implementation.

Looking at a whole program, we have many objects, each exposing a simple interface to the other objects and keeping the details of its own implementation hidden. With all the objects following this strategy, we get a divide-and-conquer solution to the whole program. Instead of a 1000 line program, we have a bunch of 200 line objects with minimal dependencies on each other. In this way, we escape the n^2 trap of writing and debugging large programs.

Public interface Design

In its simplest form, encapsulation is expressed as "don't expose internal data structures," but there's more to it than that. For good OOP design, an object must think of an interface that most succinctly meets the needs of its clients. What do the clients want to accomplish? What problem do they want solved, and what is the minimum set of details necessary to express the problem and its solution? The client of the BinaryTree class doesn't want getLeft() and insertRight() messages, they want add() and find().

Suppose an object has ivars x, y, and z. It's not very impressive OOP design if the exposed interface is getX(), setX(), getY(), setY(), and so on. An OOP interface is not just a 1-1 translation of the implementation. A good OOP design invents an interface that meets the needs of the client in terms they understand.

There's the old story of the drill bit salesperson who was much more successful once they realized that their clients didn't want to talk about drill bits, the clients wanted to talk about holes. By the same token, if you find yourself making accessors for getNumElements() and getElement(), you have to think of the underlying problem the client wants solved— probably something more like findElement() or writeElements(). Think about what the client wants to **accomplish**, not the details and mechanism of doing the computation.

3 Examples

Often, the first rule that people learn for encapsulation is that instance variables should be declared private. Client objects should not "reach in" to access the data inside an object. Here are three examples that start with that simple sense of encapsulation and enlarge it to express the larger goals of OOP.

For this example, suppose we are writing client code to a Binky object that contains some integers, and we want to know the sum of those integers.

Example 1 - Reach In

This first example is bad code, since the client reaches into the Binky object to access the data. This violates the simplest notion of encapsulation. Typically an OOP design prevents this by keeping instance variables private.

```
// client side code
private int computeSum(Binky binky) {
   int sum = 0;
   for (int i=0; i<binky.length; i++) {   // NO -- reaching in
      sum += binky.data[i];               // NO -- reaching in
   }
   return sum;
}
```

Example 2 - Letter But Not the Spirit

This example is also bad. The code follows the letter of the law of encapsulation, but not its spirit. Accessors getLength() and getData() have been added to the Binky interface, so the client does not technically access the data directly. However, this does not look like good OOP design. The client is pulling all the data out of the object to do an operation with it. Ideally, an operation using an object's data should be performed by the object on itself.

```
// client side code
private int computeSum(Binky binky) {
   int sum = 0;
   for (int i=0; i<binky.getLength(); i++) { // NO -- external entity doing
      sum += binky.getData(i);            // too much work on object's data
   }
   return sum;
}
```

Example 3 - Right

If you find yourself wanting to do some foo() operation that uses a lot of data form some object, then consider adding foo() as a method of the object and **it can do the operation on itself**. In this case, we move the computeSum() code to be a method of the Binky class. Notice how easy it is to write the code for computeSum() once it's been moved to the right class. No parameter is necessary since the Binky is the receiver and the required ivars are right at hand. Move the operation to the data.

```
// Give Binky the capability
// (this is a method in the Binky class)
public int computeSum() {
   int sum = 0;
   for (int i=0; i<length; i++) {
      sum += data[i];
   }
   return sum;
}



// Now on the client side we just ask the object to perform the operation
// on itself which is the way it should be!
...
int sum = binky.computeSum();
```

Reality

In reality not all cases have a tidy solution, and strict encapsulation may not be worth the trouble, but it's a good goal to keep in mind. Sometimes an operation requires significant access to data from two or more objects. In that case, you end up making it a behavior of one object and pass in the other object as a parameter. Comparison operations always have this problem. Many times though, you can add some helper behaviors to one of the objects so the other can interact with it while still maintaining the spirit encapsulation.

Summary

- Separate abstraction from implementation -- in OOP, expressed as messages (interface) vs. methods (implementation).

- "Expose" an interface that makes sense to the clients. Ideally, the interface is simple and useful to the client, and the implementation complexity is hidden inside the object.

- Objects are responsible for their own state -- move the code to the data it operates on.

OOP Design Ideas

Robust Object Lifecycle

Ctor sets object to a valid state

Methods -- take object from one valid state to another valid state

Client can send messages to invoke methods, but the state of the object is ultimately controlled by its own methods.

Notice that the client has a very limited ability to screw up the object. The client can give the object bad input, the but internal correctness of the object maintained by its own methods.

Methods as Transactions

In databases, a "transaction" is a change to the database that either happens completely, or the database "rolls back" so the transaction has had no effect at all. In this way, the database is always kept in a valid state.

It's nice if methods work like that.

Each method changes the object from one valid state to another. When the method is done, the object is in a new, correct state, and ready to respond to any message. Ideally, a method never leaves the object in a "half baked" state where it cannot respond to some of the messages it is supposed to support.

Modal States

- Sometimes the object has unavoidably different states -- some messages only work in some states.
- e.g. Iterator -- can't send remove() without preceding next()
- These are a little harder for the client -- a less client friendly design. Harder to document as well.

Clueless Client Test

Is it easy to be a client?

Even if the client did not read the docs carefully and is not really paying attention -- depends on having a simple, intuitive interface, and having the compiler flag common errors.

For the classic lifecycle, the constructor forces the client to put the object in a valid state. Then they can send messages, but they always leave the object in a valid state.

Forcing the client to always go through messages allows us to design reasonable "clueless client" performance. It would not be possible if the client could reach in and change ivars.

Advantages of Classic Encapsulation...

1. Clean Code

At a syntactic level, writing code as an OOP method of the receiver is easy since all the private ivar data is right at hand -- no parameters etc. are required. Similarly, the client code is nice and simple -- the client just sends a message, nicely shielded from the complexity of the implementation.

2. Modularity -- Escape the N^2 Trap

With good OOP encapsulation, the program looks like five 200 line classes, instead of a single 1000 program. This helps us escape the n^2 complexity curve of writing and debugging long programs.

When coding and debugging each class, we are shielded from dealing with the details of the other classes.

If a change needs to made, that change and its new debugging cycle is ideally isolated inside a single module.

3. Separate testing

Easier to test smaller modules individually, rather than test the whole thing at once. Also, testing can start earlier, rather than waiting for the whole thing to be done. In Java, this is frequenly done by writing a little main() in each class with some simple test code that exercises objects of that class.

4. Code Re-use...Libraries

By minimizing inter-module dependency, modules are ready to be re-used in throughout the program. The natural extension of that theme is adding the class to an official library for use by others. Many of the classes in the Sun libraries started out as little

5. Team Programming

Modular style works well with a team -- each module can be owned, tested, etc. by a person.

Client Oriented Design

Encapsulation is the 1st principle of OOP. Client Oriented Design is the 2nd.

What abstraction should an object expose?

The guiding principle should be to meet the needs of the client classes -- solve the problem they want solved. Hide the non-relevant details of the implementation as much as possible.

Great Designs

Great designs are not made by getting public and private exactly right.

IMHO, programmers can get caught up in the details of public/private too much, just because they are so visible. The real issue is much more subtle -- constructing a sensible abstraction. There is no simple rule to getting this right.

Great designs depend on thinking of a set of messages that expose a simple, sensible abstraction to the client, while hiding implementation complexity as much as possible.

Documentation test

If the docs are short and easy to express, it's a good sign for the client oriented design.

Or put the other way, if the docs seem to need to explain aspects of the implementation, or use phrases like "unless" or "but first, you must always" .. that's a bad sign.

Not Just the Implementation

The common error is to simply expose each element of the implementation.

If it's a binary tree..

- Wrong: provide getLeft() and getRight() methods.
- Right: provide a findElement() method -- what problems does the client actually want solved?

There's a theory that the person who does the implementation is always ill suited to thinking of the abstraction -- their mind is already biased towards the implementation world view.

Choose Abstraction For the Client

Set up abstractions and vocabulary that make sense to the client. The abstraction should be optimized to be comprehensible, while expressing the details the client cares about while hiding the details they don't care about.

e.g. ChunkList

- Invent the Iterator abstraction -- hasNext() and next() methods -- as a made-up abstraction for the client to use to interact with iterating.

e.g. String

- String exposes an abstraction that its chars are numbered 0..len-1 in its charAt() and subString() methods.
- This is an easy to understand abstraction to expose to the clients -- but it is a huge lie!
- In reality, the Strings uses a section of chars with a particular offset and length inside of a char[] array that could be shared with other Strings. String presents the simple, consistent view to its clients, shielding them from the details of the implementation.

e.g. HI design -- the File System Browser

- Inside, the file system is made of inodes, different devices, different filesystems, ...

The Finder presents an invented, graphical representation that includes the relevant details and supports relevant operations. It is an internally consistent world.

Principle of Least Surprise

If an object responds to a message like add() or length(), the resulting behavior should be what the client would guess if they did not read the documentation, because in fact, they are not going to read the documentation.

If a message is going to do something weird or unusual, it should not have an innocent little name.

Client "Use Case" Analysis

If you are designing a class, think about the most common client "use cases" to drive what abstraction to expose.

What is the mindset of the typical client, their knowledge, their vocabulary...

What problems do the clients need solved? Which problem scenarios are most common?

Which details will be relevant and which can be hidden?

Common Case Convenience Methods

Usually, there are some obvious, common use cases.

Have convenience methods that do exactly the common cases. Emphasize these in the docs and sample code.

e.g. Collection.add()

- add() is really a special case of insert(). Some libraries have forced the client to add to the end of a collection using insert like this: coll.insert(obj, coll.length()-1);
- Technically, insert() exposes the needed functionality the client needs, but it's a pity to make the client go through several steps for such a common use case..
- It's better to support the common add-at-the-end case with a special purpose coll.add(obj) method, even if behind the scenes it just calls insert().

General vs. Specific

- General case tools, like insert(obj, pos), are more powerful. However specific tools, like add(obj), are easier for clients to understand, especially at first. This is a cognitive truism -- concrete cases are more comprehensible.

Path of Least Resistance

If there is an easy way and a hard way for the client to call your code, they will always choose the easy way.

Therefore, make sure the easy way also is the "right" way for the client to proceed

e.g. malloc() -- unreliable design, the client is supposed to check the error return code, but they never do.

e.g. new -- throws an exception on out of memory. If the client calls new and does nothing else extra, the exception will handle the out of memory case automatically..

Easy things should be easy, hard things should be possible.

If the client wants standard memory behavior, they don't have to do anything. If they want some custom error-handling, they have to understand the issues and put in the exception handling code.

Bad Design: strncpy()

strncpy(dest, source, n) -- "copy at most n chars from source to dest. Pad with '\0' chars if source has fewer than n chars."

The common client use case: copy source to dest, keeping it '\0' terminated. Truncate the string if it is too long.

It's not very obvious how to get that effect with strncpy(). In fact, there is no simple way to call strncpy() that will solve the common use case. It is a truly a terrible design.

Calling strncpy in the obvious way stncpy(dest, source, dest_len) -- appears to work for small strings, but will fail randomly if the source is as long as the dest, since in that case the '\0' is not put in.

It's ridiculous that trying to do the obvious thing requires the client to think hard about the various cases.

(HW1) Time Example

Suppose you have a Time class that represents a time, such as "10:53 am". There's a simple Time.java among the starter files for hw1) See the java Date class.

Operations

What operations might the Time class expose?

Theme: expose things for the convenience of the client. Hide implementation details.

- -getHours()/getMinutes()/isAM() -- standard accessors
- -setMinutes()/setHours()/setAM() -- these may need to "renormalize" the data from the client, e.g. minutes to the range 0..59, to maintain the internal correctness of the time object and its assumptions. This is an advantage of making the client go through setter methods -- the object can control its state.
- -isBefore(Time) -- compare to another time: is the receiver before or equal to the given time
- -shift(int hours, int minutes) -- shift the receiver by the given hours and minutes. (internally handles messy wrap-around logic for hours and minutes, midnight)

Implementation vs. Interface

Could implement as: int hours, int minutes, boolean am/pm

Could implement as: int minutesSinceMidnight

Give the illusion to the client of hours/minutes, but do internal logic just in terms of minutes

The abstraction should be optimized to be convenient and understandable to the client. The implementation should be optimized for easy implementation.

The hours/minutes representation may be most convenient for the client, but it is not an easy represenation for computations.