OOP Design

Handout written by Nick Parlante

OOP Design

- OOP Design spans a few main themes...
  - Encapsulation and modularity
  - API/Client interface design
  - Inheritance hierarchy and interfaces
- Here we concentrate on OOP encapsulation and API/interface design. Note that this is more than one lecture's worth of material.

Software Engineering Crisis -- Systems -- Modularity

- Problem -- building large systems made up of complex parts, hit "n squared" limit
- Picture of typical project -- many complex parts
- Code of one part can interfere with code of another
- Solution -- modularity. Keep components as separate and independent from each other as possible. This idea works in many coding styles, but is embodied in OOP especially well.
- Self taught people can fall into the trap of building the system as one large thing. Experience with very large projects shows the value of deliberate modularity -- building the large thing out of consciously modular parts.

OOP Design -- Encapsulation

- Divide the code for a project up around its nouns.
- Create a class for each type of noun -- storing the data for that type of noun and the operations that work on it.
- "Encapsulation" is the idea of housing data in an object which stores and manages that data, exposing a clean interface for use by clients while keeping implementation details hidden.
- Each class exposes public methods for use by other "client" classes.
  - Expose operations in a way which is most convenient for the clients
  - The data and implementation details are kept private inside the object as much as possible.
- e.g. An Address object is made of a String street address, a String country, ... gather those components together to form the single coherent "Address" object.

Interface vs. Implementation

- **Implementation** -- private, internal
  - How is the object implemented -- how does it organizes its data into instance variables.
  - Methods -- what data structures and code to do the actual implementation.
  - The word "detail" suggests some feature of issue of the implementation, but which the client does not need to know. It can be kept hidden inside the object -- great! When someone says "that's a detail", they mean it's not something the client cares about.
- **Interface** -- public API, external
  - How does the object "expose" its abilities as public methods for use by client classes.
  - The interface must expose just the issues that are needed and relevant to the computation -- keeping the implementation details hidden as much as possible.
  - The public interface/API should be organized for the convenience and needs of the clients. A great client interface may look quite different from the underlying implementation. The clients need never know this -- they never see or depend on the implementation details.

- **Asymmetry** -- there is a basic asymmetry between the interface and implementation. With a good design, the interface is simple and minimal compared to the (complex, detailed) implementation that it summarizes. If the interface is as complex as the implementation ... that's a bad sign.

OOP Program Goal Picture

- With OOP design of a large system, we get...
- Modularity -- the large system is conceptually divided into modular classes
- Each class encapsulates some area of the problem, exposing clean services for use by the other parts of the program.

Advantages

- Avoid n-squared problem -- most details are hidden inside each class, incapable of interfering with details in other classes.
- Code re-use, team programming -- classes are organized for client convenience, so we have a lot of code we can use easily without knowing or depending on its implementation..
  - ArrayList, Jpeg, etc. classes in the Java library
  - Foo class written by our teammate

"Standard Model" OOP Class Design

First we'll look at the standard pattern for a class.

Easy for the Client -- Client Driven Design

- Much of the design is oriented towards making things convenient and reliable for the client code
- It's great if the client can be clumsy and not read the docs, and the API and compiler errors messages will guide them to do the right thing any way.

Data Private

- Declare data stored in the object -- instance variables -- private
- The reduces the ability of the client to screw things up, since they are forced to go through public constructors and methods
- This allows the implementation to change later on, without changing any client code

Constructors

- Have one or more constructors to set the object up for common cases.
- Constructors are, in some ways, easier for the client than calling setters.
  - You can forget to call a setter or call it at the wrong time -- that's just not possible with a constructor.
  - With the constructor, the object is set up at the start, so it avoids bugs where some aspect of the object has not been set.
- In your constructors, be sure to initialize all the instance variables
- One constructor can call another as shown here. Can have multiple constructors --- makes it convenient for the client to call the constructor that fits their situation.

```
public class Foo {
  // ctor 1
  public Foo(int val) {
    ...
  }

  // ctor 2
  public Foo() {
    this(6);  // "this" syntax calls ctor 1
  }
}
```

Public Getters

- Provide getter methods to provide data to the client, using the receiver data
  - public int getFoo() { …. }
- Notice that the client cannot tell if Foo is an ivar, is computed on the fly, is gotten out of the database -- how it's stored is a detail, and we can change the implementation later and the client will not know.
- Typically, getters do not change the receiver state.
- It's not **required** to have getters for all the data in the object-- they're just for parts of the data which it makes sense to expose to the client as part of the overall API plan.

Public Setters, Mutators

- Provide setFoo() "setter" methods to change data features that make sense for the client, operating on the receiver state. Methods that change the receiver state are also known as "mutators".
- Note that "immutable" classes don't have any mutators -- the object does not change state, which is limiting but also keeps things very simple.
- Changing object state is more complex than just looking at it (getters), so only add public setters/mutators if it's a real client need.
- The use of the words "get" and "set" is simple convention that makes Java code a little easier to read.
- We have these two widely understood method types -- getters, mutators -- so design a method to be one or the other. Don't create a method, like getBalance() that looks like a getter but actually also changes the object in some way (see "principle of least surprise" below).

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 mutators work like that.
- Each mutator 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.

Valid States vs. Modes

- In a more complex design, the object may have "modes" or special states where only some methods work. Sometimes such modes are necessary, but they are certainly not as clean, e.g. for an Iterator, next() and remove() can only be called at certain times.
- Immutable objects avoid the whole problem, since they don't change.

OOP Robust Object Lifecycle

- Client creates object -- must go through implementation constructor
- Client can call methods in public interface to change object -- uses implementation code for each method. Methods move the object from one valid state to another.
- The client says what they want, but the implementation code is the only one to touch the data -- the implementation can maintain basic correctness and consistency of the object.
- Notice that the client has a very limited ability to screw up the object. The client can give the object bad input, but the internal correctness of the object can be maintained by its own methods.
- If the implementation is correct, the ability of the client to mess things up is limited

Private Utilities

- You may decompose out a method for your own use within your class -- declare it **private** unless there's a good reason to expose it for client use.

- For production code, once a method is public, it must be supported with later releases of the code, since client code starts depending on it -- you cannot remove it or change its parameters.
- If a method is private, you can change whatever you want, since nobody else is using it.

Push the Code to the Data

- Suppose we are creating an online video site.
- Gather the data into logical clumps -- a User object for the data for one user, a Movie object for one movie..
- Push code to the class that contains the data it manipulates. So the code that, say, compares two users, push to the User class. So when compareUsers() runs, it runs against a User object with the data it needs right there.
- Or rather, it's a bad sign if you find yourself pulling lots of data out of an object to do a computation.

Encapsulation Examples

OOP Encapsulation Examples

- Each object encapsulates some possibly complex data
- Exposes some operations on that data to clients
- Operations compute or modify the data in the receiver object
  - The operations work on the stored data for the client, inevitably shielding the client from the details of both how the data is stored and the code for the computation. The client may never need to see the "raw" data. The object takes care of it for the client.

1. BufferedImage Example

- Image class -- represents an image, such as read from a JPeg
- Stores the pixel data … somehow
- Operations
  - get size, getScaledIntance(width, height) returns a new Image which is a size scaled version of the receiver

2. ShoppingCart Example

- Suppose your .com has some checkout/cart process, implemented in part by a Cart class..
- Cart stores (has) a collection of Items, each Item has a price and weight
- Cart stores a shipping choice
- Operations
  - Add/remove Item
  - Set shipping choice
  - Get sum of item prices
  - Get shipping price (function of shipping choice and the items)
  - Get total price
- Notice that the operations on the cart object work on the whole cart state (all the items + shipping method) and return nice summary answers to the client. Don't make the client go digging through all the items -- the Cart deal with it all for the convenience of the client.
- Notice that we do not depend on how the Cart stores the data -- an ArrayList, a HashMap, in a database on the local network, … it's an implementation detail inside the Cart.
- The Item is a separate class -- exposing methods like getPrice(), getTitle(), getWeight(), isInStock() ... for use by classes like ShoppingCart.
- There is probably some design coupling between the Cart and Item -- they work closely together, so that's ok. Try to keep the coupling to the minimal, relevant details they need to agree on.

4. Digital Camera CardInterface

- CardInterface class -- deals with image storage in a digital camera. Appears to have a collection of Images, in reality reads and writes images to compact flash card.
- addImage(Image) -- adds to file systems, gives it a name/serial number (called by record mode, when you take a shot)
- getSize(), Image getImage(n), deleteImage(n) -- called by play mode to see how many there are, allow you to look back and forth through them, see them, delete them
- Not exposed: how compact flash card works, how the file system on it is organized, the wacky "img0234.jpg" filename convention used by that layer.

Variant: Immutable Style -- popular for simple cases

- The object state is set when it is created, and the object never **changes** after that. Pointer sharing of immutable objects never causes problems, since the objects never change -- eliminates a whole category of bugs.
- Has getters to expose state or computation, but no mutators.
- In a way, a very simple model to expose to the client. e.g. as seen with the classes String, Integer, Color
- If the client wants an object with a different state, a method returns a **new** object with that state in it (e.g. String.substring())
- The immutable style is simple, but has limited capability. Its attractiveness is its simplicity -- easier for the client to understand and use, easier to implement.
- The simple immutable style is an attractive design if it is capable of expressing the needed computation.

Variant: Default Constructor/Bean Style

- New object is created with the default, zero-argument constructor
- The new object is in an "empty", no-data state, which may not be valid
- The client calls setters to put data into the object for each field, getting it to a valid state
- Essentially -- we use setters instead of constructor arguments
- Advantage: simpler in a way, we do not have both constructor args and setters to set things up. Can be used in an automated way by code-gen tools that know how to call setters. You can add more data fields over time without adding more fields to the constructor. The number of fields can be large.
- Disadvantage: blurs when an object is valid, and the client needs to remember to call the right setters. The beauty of constructors is that if the client forgets, it won't even compile!

Loose Coupling vs. Tight Coupling

- It's best if classes are "loosely coupled" -- with minimal dependencies on other classes. In this way, they can be used in many contexts.
- Loose coupling is a result of a minimal interface that does not bring in any extraneous detail or dependency.
- e.g. ArrayList -- its interface includes just the clean minimum needed to talk about any type of collection of elements. Therefore, ArrayList is general purpose and can be used in many contexts.
- Loose coupling is not always possible. Sometimes a pair of classes cooperate closely, and the most reasonable design acknowledges that dependency (aka "tightly coupled" classes). Not all classes are general purpose.

Encapsulation in a Nutshell

- Design the public interface for the convenience and needs of the clients -- the design is fundamentally client driven
- Keep implementation detail hidden inside the class as much as possible -- avoid dependencies between classes as much as possible, as that's what kills large systems

OOP Interface/API Design

Interface/API Design

- Encapsulation is the 1st principle of OOP.
- Good, "client oriented" interface design is the 2nd.
- Interface design is also known by older term "API design" -- Application Programmer Interface
- What abstraction should an object expose for client classes to use?
- The guiding principle is that the interface should be "client oriented" -- meeting the needs of client classes. Solve the problem the client wants solved, using the client's natural vocabulary and data format. Hide the non-relevant details of the implementation as much as possible.

Start With Client Viewpoint

- The first step in thinking about interface design is taking the client viewpoint, not the implementation.
- e.g. designing String class don't think about arrays of chars, think about what operations clients are likely to want from a String class. Work from those needs to the implementation.

Easy To Be a Client

- Being a client should be easy. If the implementation is complex -- fine.
- Imagine that the code is implemented once, but used by many clients.
- The constructors and methods should expose the features the client wants and guide the client to do the right thing. The class could throw an exception for common client errors -- alerting them when they go astray.
- Ideally, the client should not be able to "reach in" to the object and mess things up (recall Robust Object Lifecycle).
- Ideally, the exposed interface does not expose issues/pitfalls that are outside the client's expertise. The interface can guide the client on the correct path, even if they don't read the docs.

Interface / Implementation Asymmetry -- High Level

- The interface can be much simpler than the implementation -- there's a basic asymmetry in their complexity
- If the interface is as complex as the implementation, OOP is little use
- The interface should not just be a 1-1 translation of the implementation
- Thinking in terms of the implementation can be misleading for interface design
- The operations should be at a higher solve-problem level compared to the components of the implementation.
- Examples. Exposed interface idea -- what it accomplishes -- is much simpler than underlying implementation
  - String.indexOf()
  - Tetris Board.clearRows()
  - Tetris Piece.computeNextRotation()

Documentation Test

- If the docs describing the interface are short, intuitive, 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 a Restatement of the Implementation -- High Level

- The most common API design 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 ill suited to thinking of the interface -- their mind is already biased towards the implementation world view.
- Good API methods should work at a higher level -- solving a client problem, not down at the implementation level.

Operate On Whole Object

- Suppose we have an Address object that represents a mailing address.
- Wrong: the client calls pull out getStreet() and getCounter() getZipCode() to get the Strings out to make a mailing label.
- Right: the Address has a getMailingLabel() that knows how to make a mailing label using all the data in the address.
- Point: the object operates as a functional composite of its data. The client can think of "Address" as a functional unit, and it takes care of making the component parts work as a coherent whole.

Invented Interface For the Client -- Lying!

- Expose abstractions and vocabulary that make sense to the client. The abstraction should be optimized to be comprehensible -- expressing things the client cares about while hiding the details they don't care about.
- 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. (Actually, different JVMs can implement String differently (the client can't tell!), but some use the implementation described above.)
- e.g. ArrayList
  - Invent the Iterator abstraction -- hasNext() and next() methods -- as a made-up abstraction for the client to use to see all the elements.
- e.g. HI design -- the File System Browser
  - Inside, the file system is made of inodes, different devices, different filesystems, ...
  - The file browser presents an invented, graphical representation that includes the relevant details and supports relevant operations. It is an internally consistent world.

Receiver Relative -- Move The Code to the Data

- Ideally, most of the data used by a method should come from its receiver object.
- In other words, place the method in the class that contains the data needed by that method. Move the code to the data.
- Other objects may be passed in as pointers, but ideally they are peripheral, and the operation mostly uses the state of the receiver object.
- If a method **changes** an object, then that object is a good candidate to be the receiver (as opposed to changing an object passed in as an argument to the method).

Code Goes Where? Which Object is the Receiver?

- Where to place a method if the operation requires data from 2 or more objects -- which object should be the receiver?
- Choose as the receiver, the object who's state is used the most
- Choose as the receiver, the object who's state is changed

Cart Example

- Suppose we have a shopping Cart object that encapsulates a collection of Items. We want to check if an item is in the cart.
- Question -- which interface design?
  - boolean cart.hasItem(item) -or-
  - boolean item.inCart(cart)
- In this case, I prefer cart.hasItem(item). The cart encapsulates the problem of storing all the items. It "has" all the items, and that's the major data structure for this problem. Move the code to the data.

Recipe Example

- Suppose
  - we have a Recipe object that encapsulates a list of ingredients and amounts
  - we have a Party object, that knows who is coming to the party, and how much they each eat
  - we want to scale the ingredient amounts to match out party guest list
- Question -- which interface design?
  - recipe.scaleFromParty(party) -or-
  - party.scaleRecipe(recipe)
- In this case, recipe.scaletoParty(party) is better, since the object being **changed** is the receiver. Also, scaling ingredients is operates on recipe data (using party input), not the other way around. The computation should be in the class that holds the data of the computation.
- Not all examples have a tidy solution, but understanding the receiver-relative style is an important goal to keep in mind.

Public/Private Not The Focus

- Great designs are not made by getting public and private exactly right.
- Programmers can get caught up in the details of public/private too much, just because they are so visible. The more interesting design issues are more subtle.
- 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.

OOP Interface Principles

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. Most of the uses of a class are common and obvious. Weird uses of a class are more rare.
- Have convenience methods that do exactly the common cases. Emphasize these in the docs and sample code.
- e.g. Collection.add()

  - add(obj) is really a special case of insert(index, obj). Some libraries have forced the client to add to the end of a collection using insert like this:
    coll.insert(coll.size()-1, obj);
  - Technically, insert(index, obj) 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().
- e.g. Generate random int in range 0...n-1 -- random.nextInt(n) vs. doing it manually: (int)(rand()*n)
- e.g. Print panel in your OS has an "all" button -- super common use case.
- General vs. Specific
  - General case tools, like insert(index, obj), are more powerful. Mathematically, we like general solutions. However specific tools, like add(obj), are easier to use and understand. General is not necessarily better for the goal of ease-of-use.

Implementation Should Call Client Methods ("circle back")

- When writing code in the implementation, you can still call your own high level public interface methods.
- e.g. in the Cart implementation, you can take advantage of "high level" Cart methods like hasItem() and addItem().
- The public methods tend to be useful, high-level methods, and they make changing the implementation easier.
- e.g. optimize hasItem() to cache results or something ... the optimization just works automatically if the implementation code itself calls hasItem() where appropriate.

Path of Least Resistance vs. Incorrect Code

- 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 provides reasonably correct behavior. In other words, if the client does the totally obvious, minimal work thing, they should get reasonable behavior.
- e.g. bad design: C malloc() -- unreliable design, the client is supposed to NULL check the returned pointer, but they often omit the checking code. Therefore, the C program just crashes in random ways if memory is getting tight when the client takes the easy path.
- e.g. good design: Java new -- by default throws an exception on out of memory. If the client calls new and does not think about or do anything extra, the exception will flag the out of memory case automatically and terminate the program in a well defined way. That's actually a big improvement over the C behavior.
- If the client wants standard memory behavior, they don't have to do anything (the default behavior is reasonable). If they want some custom error-handling, they have to understand the issues and make the extra effort to in the exception handling code.
- **Calling the code in the obvious way should yield reasonable default behavior.**

Easy Things Should be Easy, Hard Things Should be Possible

- Old design saying that hits these same ideas
- Common, obvious cases should be easy to call and have reasonable defaults
- If the client wants to do something weird, it should be possible, and we don't mind if they have to write more code in that case.

Bad Design: strncpy()

- Worst API design ever
- 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, leaving dest as a C string (i.e. a '\0' terminated string). Truncate the dest string if it is too long.
- It's not very obvious how to get that effect with strncpy(). The apparent post condition is that dest is left as a C-string, but in fact sometimes it is and sometimes it is not. 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 or longer than the dest, since in that case '\0' is not put in, and so dest is no longer left as a valid C string. Essentially, the post condition is hard to characterize -- sometimes it's a valid C string and sometimes it's not. It's an especially bad design that appears to work when run on small cases, but changes its behavior for large cases -- makes testing especially difficult.
- It's ridiculous that trying to do the most obvious, common case requires the client to think deeply about weird cases. Calling some code in the obvious way should not require the client to get out a little piece of paper and make drawing to decipher the cases.
- Modern C implementations have functions srlcpy() strlcat() that interpret "n" in a reasonable way, and have the postcondition that dest is always a valid C string.