CS108, Stanford	Handout #20
Fall, 2008-09	Osvaldo Jiménez

OOP Design Conclusions and Variations

Thanks to Nick Parlante for much of this handout

Part 1 -- Mainstream OOP Design

- First, we have the standard, mainstream understanding of OOP, summarizing the most important ideas...

Encapsulation / Modularity

- Isolate detail inside a class -- the data structures and code that implement
- Clients should not need to know, or in fact be able to know, the details of implementation
- Fight the n^2 blowup of complexity in a project by isolating and hiding details inside each module
- As a rule of thumb, this leads to a class design where instance variables are all declared private, and then select methods are declared public to expose a limited view to the client.
- The ultimate working system looks like a community of largely independent objects, each offering clean interface services to the others while keeping implementation complexity hidden inside.

Tight vs. Loose Coupling

- Two classes are "tightly coupled" if they share details and assumptions such that changing one will likely require changing the other.
- Two classes are "loosely coupled" if they share the minimum information to work together, each hiding its own implementation details.
- If classes are loosely coupled, it should be possible to remove one and replace it with another implementation. If a class is loosely coupled to its environment (i.e. it has a general purpose interface), we can use it in other places, e.g. `ArrayList`.
- Modularity, in other words, is about trying to keep classes loosely coupled with each other as much as possible.

Private ivars and Public methods

- Suppose a `count` ivar is `public`
- A client can write code…
```
    - if (foo.count == 10) {…
```
- If the ivar is private, we can force the client to go through a getter…
```
    - if (foo.getCount() == 10) {…
```
- By forcing the client through the public method … we decrease coupling. We can change how the `count` is stored and updated, and so long as we keep `getCount()` in sync with our scheme, the client will never know. Ultimately, the client does not know if we store or compute `foo`, and in fact we have the freedom to change how it works over time -- that's loose coupling.
- For a large project and for team projects, we certainly want to keep coupling low where possible.

Client-Oriented API Design

- Expose only the minimal public interface (API) that the clients need -- minimize coupling between a class and its clients

- Public API should meet the client needs, what they want to accomplish (not **how** it is accomplished). It's a bad sign if the API is a 1-1 reflection of the implementation.
- Design the client API to meet their needs and common use cases -- make things easy for the client, using the client's natural vocabulary. It's a good sign if the API works at a high, problem-client-wants-solved level e.g. `HashMap.get(key)`, `TetrisBoard.clearRows();`
- Principle of least surprise
- Convenience methods for common client use cases

Receiver Relative Coding

- Move the computation to the class that owns most of the data for that computation.
- Don't pull data out of an object to do the computation. In that situation, push the computation to that object's class, so it can just work on the receiver.
- If an object is changed by the computation, that object is a great candidate to be the receiver.
- Sometimes there is not a clean solution
    - Select the most centrally used object as the receiver, and pass the others as arguments
    - Add helper methods to the other objects to try to maintain receiver-relative style

Inheritance

- There is a very strong "isa" constraint between a subclass and superclass. The subclass, must be a particular type of the general superclass.
- Inheritance allows us to have one copy of the code/data up in the superclass, shared by all the subclasses. Avoid duplicating superclass code in subclass -- Horse/Zebra story.

Message Resolution, pop-down

- A message end, such as `x.foo()`, uses the runtime class of `x` to "pop down" and use the correct code, aka "message resolution".
- Leverage message resolution to select the right bit of code, rather than manually constructing if-logic to choose bits of code based on the type of object.

Subclassing is Non Trivial

- To write a subclass, you must understand and obey the superclass design (e.g. `JBrainTetris` -- must first understand how `JTetris` works). This is harder than being an ordinary client of a class.
- In other words, it is much harder to obtain loose coupling between a superclass and its subclass. They are likely to share implementation ideas.
- Do not worry about supporting subclassing for every class you write. Supporting subclassing should be a deliberate choice where it makes sense, and in that case the design and docs will need to think about how to support subclassing.

Typical Inheritance Strategies x 2

- The main theme is that there's one copy of the code up in the superclass, used without duplication by all the subclasses. This plays out in two typical ways:
- 1. There is an existing superclass which contains common, factored-up code. The superclass may be written by someone else, e.g. `AbstractCollection`, `JComponent`. Introduce a new subclass off the superclass. The subclass overrides just a few key methods. Control pops down to the subclass at key moments, while most of the code is up in the superclass.
- 2. Start from scratch to build a set of classes to build. Organize them into an inheritance hierarchy, factoring the common behavior up into a purpose built "abstract superclass" which contains the common/default behavior shared by the subclasses.

Part 2 -- Variations and Exceptions

- Realistic, shade-of-gray examples that do not necessarily fit the standard OOP Design picture.

Encapsulation/Coupling Level -- It Depends

- We certainly prefer loose coupling -- keeping things as independent as possible
- Test of good encapsulation / loose coupling -- can you go in and change the implementation of a class around, and the other classes continue to just work?
- However, for a real project, the level of coupling between classes depends on the situation
- Some classes encapsulate really cleanly, totally separating their implementation from their clients
  - e.g. `Collection` and `HashMap` and `String` do a nice job of this -- the interfaces are minimal, and the clients do not depend (or have any idea really) about the implementation details.
  - This is especially important, since those classes are out in use by millions of other pieces of code. Those interfaces need to be clean, and cannot change.
  - This is great when it's possible.

Loose Coupling All The Time -- NO

- Loose coupling is an important goal, however it is not always possible at reasonable cost.
- Sometimes two classes have such as close relationship that abstracting them from each other imposes too high a cost.
- There is a sweet spot -- how much detail can be hidden while keeping the API reasonably simple?
- e.g. Tetris `Piece` and `Board` succeed in hiding most of their implementation details form each other (e.g. how rotation and row clearing work), but they share basic a basic conceptual framework about how blocks are numbered.
  - e.g. Tetris `Piece` and `Board` share a basic implementation paradigm -- how the blocks are indexed, etc.
  - The hide much of their implementation, but not 100%
- Do not insist on making every class in your program totally separate from every other class. Sometimes, little clusters of classes share implementation ideas/assumptions.
- Questions that suggest the benefit of loose coupling: are these two classes revised at the same time? (tight coupling may be tolerable), or are they controlled by separate parties on separate release schedules? (tight coupling very bad)

Variant: Struct

- Sometimes an object is just serving as storage, and the real work is in some other class. We might tolerate public ivars on a struct class (purists might still object though).
- e.g. `BinaryTree` class has interesting binary tree implementation code. It uses a nested `Node` class that is just a struct to store the (data, left, right) pointers. The tree is built from `Node` objects, but the `BinaryTree` contains the code.
- Q: Why have a `BinaryTree` class? Why not put the methods in `Node`?
- A: there turns out to be a distinction between the tree as a whole (Binar©yTree) vs. an individual node (`Node`). You want to address messages like `insert()` to the tree as a whole, not to an individual node. Also, the empty tree case is a problem for the Node, since you can't send a message to a null pointer.
- The "struct" class could have some methods or a ctor to obey receiver-relative style, but most of the code is somewhere else.

Variant: Inner/Nested

- The relationship between the outer class (`ChunkList`) and its inner class (`Chunk`, `ChunkIterator`) is not one of pure encapsulation. The classes share all sorts of implementation assumptions, and it can be appropriate for them to access each other in a way that would not be appropriate for two unrelated classes.

Variant: Detail/Knowledge Isolation

- It can be argued that modularity and receiver-relative are just techniques -- detail/knowledge isolation is the true goal. With good detail isolation, the code is structured so that the knowledge and details for each aspect of the solution are isolated in one area of the code (as opposed to be spread all over the code).
- Such detail isolation might be at odds with receiver-relative style (see below)
- When changing one aspect of the program -- how it deals with files, or how it stores some objects -- ideally the change should occur in one area of the sources, not spread all over. Other parts of the program are insulated from the detail, since they go through `loadData()` or `findObject()` messages.

Tetris Drawing Example

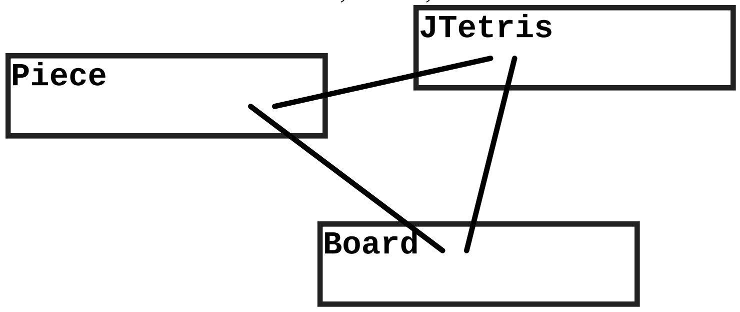
- What about drawing in Tetris?
- Not straight receiver relative OOP -- `Piece` and `Board` do not draw themselves.
- OOP variation: Isolate the knowledge of some aspect in one place.
- e.g. rather than both `JTetris` and `Board` and `Piece` know about pixels (and need to all be consistent), make a design decision that only `JTetris` will know about pixels and drawing. Code the `Piece` and `Board` to work in (simpler) terms of abstract "blocks" and are totally ignorant about pixels and drawing.

Layered Design / 1 Smart + N Dumb Classes

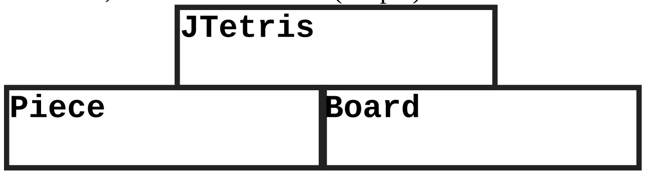
- Pick some class to be smart (dependent) about some aspect of the program, and make all the other classes deliberately dumb about that aspect (independent). That is, concentrate the detail of that aspect in one class.
- A common form of this is a "layered" stack of classes -- low-level basic classes used by higher-level classes that solve the overall problem.
- "low-level" classes -- they solve some basic problem, but do not know anything about larger context. Not knowing about an aspect keeps them and the classes and their APIs simple.
- "high-level" class can use the services of the low-level classes, and centralize the handling of some aspect of the problem.
- The point: the knowledge of the aspect must exist somewhere in the code. Avoid **everyone** needing to know about it. Isolate the complexity in a small number of classes.
- This is related to the "bottleneck" design idea for methods -- a problem is solved in one place (a single method), and everyone calls through that one method.

e.g. Tetris Drawing Designs

- Could have a world where `Piece`, `Board`, and `JTetris` all knew about drawing -- they are peers

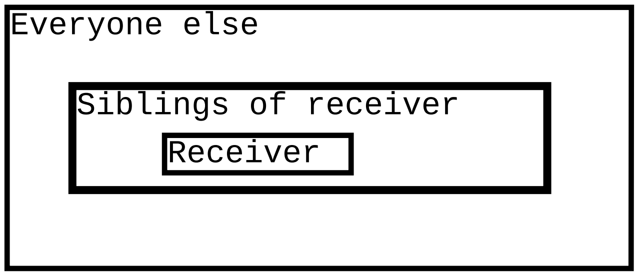


- Better to select `Piece` and `Board` to **not know** about drawing or the larger game context -- they can just implement basic piece and board logic. Isolate the complexity of drawing and game play just in `JTetris`, which builds on the (simple) facilities of the lower level classes.



Sibling Access

Most of the time, we think of the world divided into two: the receiver object and its clients. However, in reality there is a third category in between: other objects that are of the **same class** as the receiver, which I will call "sibling" objects of the receiver.



Ideally the receiver object operates on its own state. That is the best case. We also know that having some random object operate on the state of some other object is not good OOP style. In all OOP languages, it is also allowed that an object can operate on the state of another object **in the same class**, even state declared private...

```
// Suppose we are in the Binky class
// with a private "count" instance variable
public class Binky {

    private int count;

    // Increment the other objects count by our count
    public void foo(Binky other) {
        other.count += count;
        ...
```

Such sibling access is worse than having the object operate on its own state, but it is not terrible, and in some cases it cannot be avoided. In this case, the code would be cleaner adding an upCount() method so the object can change its own state, thus keeping the code in the receiver-relative style...

```
    // Increase the count by the given delta
    private void upCount(int delta) [
        count += delta;
    }

    // Increment the other objects count by our count
    public void foo(Binky other) {
        other.upCount(count);
        ...
```

However, sometimes sibling access is ok. For example, comparison operations like equals() inevitably need to access the private state of both objects.

```
    // Compare us to the other
    public boolean equals(Object other) {
        //  <equals() boilerplate code>
        ...
        // cast other to a Binky
        Binky sibling = (Binky) other;

        // Test our count vs. theirs (this works -- sibling access)
        return (count == sibling.count);

        // could write as (count == sibling.getCount())
    }
```

For read-only data access, sibling access is fine. Avoid sibling access for changes to the receiver -- those should be implemented as methods on the receiver to maintain the receiver relative style.