Object Calisthenics
9 steps to better software design today, by Jeff Bay
http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf
http://www.pragprog.com/titles/twa/thoughtworks-anthology

We've all seen poorly written code that's hard to understand, test, and maintain. Object-oriented programming promised to save us from our old procedural code, allowing us to write software incrementally, reusing as we go along. But sometimes it seems like we're just chasing down the same old complex, coupled designs in Java that we had in C.

Good object-oriented design is hard to learn. Transitioning from procedural development to object-oriented design requires a major shift in thinking that is more difficult than it seems. Many developers assume they're doing a good job with OO design, when in reality they're unconsciously stuck in old habits that are hard to break. It doesn't help that many examples and best practices (even Sun's code in the JDK) encourage poor OO design in the name of performance or simple weight of history.

The core concepts behind good design are well understood. Alan Shalloway has suggested that seven code qualities matter: cohesion, loose coupling, no redundancy, encapsulation, testability, readability, and focus. Yet it's hard to put those concepts into practice. It's one thing to understand that encapsulation means hiding data, implementation, type, design, or construction. It's another thing altogether to design code that implements encapsulation well. So here's an exercise that can help you to internalize principles of good object-oriented design and actually use them in real life.

The Challenge
Do a simple project using far stricter coding standards than you've ever used in your life. Below, you'll find 9 "rules of thumb" that will help push your code into good object-oriented shape.

By suspending disbelief, and rigidly applying these rules on a small, 1000 line project, you'll start to see a significantly different approach to designing software. Once you've written 1000 lines of code, the exercise is done, and you can relax and go back to using these 9 rules as guidelines.

This is a hard exercise, especially because many of these rules are not universally applicable. The fact is, sometimes classes are a little more than 50 lines. But there's great value in thinking about what would have to happen to move those responsibilities into real, first-class-objects of their own. It's developing this type of thinking that's the real value of the exercise. So stretch the limits of what you imagine is possible, and see whether you start thinking about your code in a new way.

The Rules
1. One level of indentation per method
2. Don't use the ELSE keyword
3. Wrap all primitives and Strings

4. First class collections
5. One dot per line
6. Don't abbreviate
7. Keep all entities small
8. No classes with more than two instance variables
9. No getters/setters/properties

Rule 1: One level of indentation per method

Ever stare at a big old method wondering where to start? A giant method lacks cohesiveness. One guideline is to limit method length to 5 lines, but that kind of transition can be daunting if your code is littered with 500-line monsters. Instead, try to ensure that each method does exactly one thing – one control structure, or one block of statements, per method. If you've got nested control structures in a method, you're working at multiple levels of abstraction, and that means you're doing more than one thing.

As you work with methods that do *exactly* one thing, expressed within classes doing exactly one thing, your code begins to change. As each unit in your application becomes smaller, your level of re-use will start to rise exponentially. It can be difficult to spot opportunities for reuse within a method that has five responsibilities and is implemented in 100 lines. A three-line method that manages the state of a single object in a given context is usable in many different contexts.

Use the *Extract Method* feature of your IDE to pull out behaviors until your methods only have one level of indentation, like this:

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                for (int i = 0; i < 10; i++) {
                        for (int j = 0; j < 10; j++)
                                buf.append(data[i][j]);
                        buf.append("\n");
                }
                return buf.toString();
        }
}


class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                collectRows(buf);
                return buf.toString();
        }
```

```
        void collectRows(StringBuffer buf) {
                for (int i = 0; i < 10; i++)
                        collectRow(buf, i);
        }

        void collectRow(StringBuffer buf, int row) {
                for (int i = 0; i < 10; i++)
                        Buf.append(data[row][i]);
                buf.append("\n");
        }
}
```

Notice that another effect has occurred with this refactoring. Each individual method has become virtually trivial to match its implementation to its name. Determining the existence of bugs in these much smaller snippets is frequently much easier.

Here at the end of the first rule, we should also point out that the more you practice applying the rules, the more the advantages come to fruition. Your first attempts to decompose problems in the style presented here will feel awkward and likely lead to little gain you can perceive. There is a skill to the application of the rules – this is the art of the programmer raised to another level.

Rule 2: Don't use the ELSE keyword

Every programmer understands the if/else construct. It's built into nearly every programming language, and simple conditional logic is easy for anyone to understand. Nearly every programmer has seen a nasty nested conditional that's impossible to follow, or a case statement that goes on for pages. Even worse, it is all too easy to simply add another branch to an existing conditional rather than factoring to a better solution. Conditionals are also a frequent source of duplication. Status flags and state of residence are two examples which frequently lead to this kind of trouble:

```
if (status == DONE) {
        doSomething();
} else {
        …
```

Object-oriented languages give us a powerful tool, polymorphism, for handling complex conditional cases. Designs that use polymorphism can be easier to read and maintain, and express their intent more clearly. But it's not always easy to make the transition, especially when we have ELSE in our back pocket. So as part of this exercise, you're not allowed to use ELSE. Try the Null Object pattern; it may help in some situations. There are other tools that can help you rid yourself of the else as well. See how many alternatives you can come up with.

Rule 3: Wrap all primitives and Strings

In the Java language, int is a primitive, not a real object, so it obeys different rules than objects. It is used with a syntax that isn't object-oriented. More importantly, an int on its own is just a scalar, so it has no meaning. When a method takes an int as a parameter, the method name

needs to do all of the work of expressing the intent. If the same method takes an Hour as a parameter, it's much easier to see what's going on. Small objects like this can make programs more maintainable, since it isn't possible to pass a Year to a method that takes an Hour parameter. With a primitive variable the compiler can't help you write semantically correct programs. With an object, even a small one, you are giving both the compiler and the programmer additional info about what the value is and why it is being used.

Small objects like Hour or Money also give us an obvious place to put behavior that would otherwise have been littered around other classes. This becomes especially true when you apply the Rule 9, and *only* the small object can access the value.

Rule 4: First class collections
Application of this rule is simple: any class that contains a collection should contain no other member variables. Each collection gets wrapped in its own class, so now behaviors related to the collection have a home. You may find that filters become a part of this new class. Also, your new class can handle activities like joining two groups together or applying a rule to each element of the group.

Rule 5: One dot per line
Sometimes it's hard to know which object should take responsibility for an activity. If you start looking for lines of code with multiple dots, you'll start to find many misplaced responsibilities. If you've got more than one dot on any given line of code, the activity is happening in the wrong place. Maybe your object is dealing with two other objects at once. If this is the case, your object is a middleman; it knows too much about too many people. Consider moving the activity into one of the other objects.

If all those dots are connected, your object is digging deeply into another object. These multiple dots indicate that you're violating encapsulation. Try asking that object to do something for you, rather than poking around its insides. A major part of encapsulation is not reaching across class boundaries into types that you shouldn't know about.

The Law of Demeter ("Only talk to your friends") is a good place to start, but think about it this way: You can play with your toys, toys that you make, and toys that someone gives you. You don't ever, *ever* play with your toy's toys.

```
class Board {
   ...

   class Piece {
      ...
      String representation;
   }
   class Location {
      ...
      Piece current;
   }

   String boardRepresentation() {
      StringBuffer buf = new StringBuffer();
      for (Location l : squares())
         buf.append(l.current.representation.substring(0, 1));
      return buf.toString();
   }
}
```

```
class Board {
   ...

   class Piece {
      ...
      private String representation;

      String character() {
         return representation.substring(0, 1);
      }

      void addTo(StringBuffer buf) {
         buf.append(character());
      }
   }
   class Location {
      ...
      private Piece current;

      void addTo(StringBuffer buf) {
            current.addTo(buf);
      }
   }

   String boardRepresentation() {
      StringBuffer buf = new StringBuffer();
      for (Location l : squares())
         l.addTo(buf);
      return buf.toString();
   }
}
```

Note that in this example the algorithm's implementation details are more diffuse, which can make it a little harder to understand at a glance as a whole, however, you have created a named method for the piece's transformation into a character. This is a method with a strong cohesive name and job, and is quite likely to be reused - the odds of "representation.substring(0, 1)" being repeated in other parts of the program has now been reduced dramatically.

Rule 6: Don't abbreviate

It's often tempting to abbreviate in the names of classes, methods, or variables. Resist the temptation – abbreviations can be confusing, and they tend to hide larger problems.

Think about why you want to abbreviate. Is it because you're typing the same word over and over again? If that's the case, perhaps your method is used too heavily and you are missing opportunities to remove duplication. Is it because your method names are getting long? This might be a sign of a misplaced responsibility, or a missing class.

Try to keep class and method names to 1-2 words, and avoid names that duplicate the context. If the class is an Order, the method doesn't need to be called shipOrder(). Simply name the method ship() so that clients call order.ship() – a simple and clear representation of what's going on.

Rule 7: Keep all entities small
This means no class over 50 lines and no package over 10 files.

Classes over 50 lines usually do more than one thing, which makes them harder to understand and harder to reuse. 50-line classes have the added benefit of being visible on one screen without scrolling, which makes them easier to grasp quickly.

What's challenging about creating such small classes is that there are often groups of behaviors that make logical sense together. This is where we need to leverage packages. As your classes become smaller and have fewer responsibilities, and as you limit package size, you'll start to see that packages represent clusters of related classes that work together to achieve a goal. Packages, like classes, should be cohesive and have a purpose. Keeping those packages small forces them to have a real identity.

Rule 8: No classes with more than two instance variables
Most classes should simply be responsible for handling a single state variable, but there are a few that will require two. Adding a new instance variable to a class immediately decreases the cohesion of that class. In general, while programming under these rules, you'll find that there are two kinds of classes, those that maintain the state of a single instance variable, and those that coordinate two separate variables. In general, don't mix the two kinds of responsibilities.

The discerning reader might have noticed that rules 3 and 4 can be considered to be isomorphic. In a more general sense, there are few cases where a cohesive single job description can be created for a class with many instance variables. As an example of the kind of dissection we are asking you to engage in:

```
class Name {
   String first;
   String middle;
   String last;
}
```

would be decomposed into two classes thus:

```
class Name {
   Surname family;
   GivenNames given;
}

class Surname {
   String family;
}

class GivenNames {
   List<String> names;
}
```

Note that in thinking about how to do the decomposition, the opportunity to separate the concerns of a family name (used for many legal entity restrictions) could be separated from an essentially different kind of name. The GivenName object here contains a list of names, allowing the new model to absorb people with first, middle, and other given names. Frequently, decomposition of instance variables leads to an understanding of commonality of several related instance variables. Sometimes several related instance variables actually have a related life in a first class collection.

Indeed, it is the authors' experience that decomposing objects from a set of attributes into a hierarchy of collaborating objects, leads much more directly to an effective object model. Prior to understanding this rule, we spent many hours trying to follow data flows through large objects. It was possible to tweeze out an object model, but it was a painstaking process to understand the related groups of behavior and see the result. In contrast, the recursive application of this rule has lead to very quick decomposition of complex large objects into much simpler models. Behavior naturally follows the instance variables into the appropriate place.

Rule 9: No getters/setters/properties

The last sentence of the previous rule leads almost directly to this rule. If your objects are now encapsulating the appropriate set of instance variables but the design is still awkward, it is time to examine some more direct violations of encapsulation. The behavior will not follow the instance variable if it can simply ask for the value in its current location. The idea behind strong encapsulation boundaries is to force programmers working on the code after you leave it to *look for* and *place* behavior into a single place in the object model. This has many beneficial downstream effects, such as a dramatic reduction in duplication errors and a better localization of changes to implement new features.

Another way this rule is commonly stated is “Tell, don’t ask”.

Conclusion

7 of these 9 rules are simply ways to visualize and implement the holy grail of object oriented programming – encapsulation of data. In addition, another drives the appropriate use of polymorphism [not using else and minimizing all conditional logic], and another is a naming strategy that encourages concise and straightforward naming standards – without inconsistently applied and hard to pronounce abbreviations. The entire thrust is to craft code that has no duplication in code or idea. Code which concisely expresses simple and elegant abstractions for the incidental complexity we deal with all day long.

In the long run, you will inevitably find that these rules contradict each other in some situations, or the application of the rules leads to degenerate results. For the purpose of the exercise, however, spend 20 hours and 1000 lines writing code that conforms 100% to these rules. You will find yourself having to break yourself of old habits and change rules that you may have lived with for your whole programming life. Each of the rules has been chosen such that if you follow it you will encounter situations that would typically have an easy answer that is not available to you.

Following these rules with discipline will force you to come up with the harder answers that lead to a much richer understanding of object oriented programming. If you write a thousand lines that follow all these rules you will find that you have created something completely different than you expected. Follow the rules, and see where you end up. If it isn't comfortable, back off and see what you can leverage that is comfortable. You might find that if you keep working at it, you see code you are writing conform to these rules without any conscious effort on your part.