Code Qualities and Coding Practices

Practices to Achieve Quality

Scott L. Bain and the Net Objectives Agile Practice

13 December 2007

Contents

Overview

Why do we think about Code Quality? Because we need to do less of the wrong stuff, do more of the right stuff when we code, do well the stuff we end up doing, and doing it in a managed way that is consistent with lean and agile practices. This paper briefly overviews thirteen practices that, taken together, vastly improve the quality of code that Agile teams produce.

Figure 1, below, lists these practices and puts them into relationship.

The technical practices track of the Net Objectives Lean-Agile Software Development curriculum goes into detail about each of these practices and describes how teams, individuals, and organizations can put them to use to improve their code quality.

One of the fundamental requirements of achieving superior code quality is understanding what is meant by code quality in the first place. The principles of Lean and Agile help organizations to understand this. Applying these principles to product development efforts, organizations can have the necessary conversations about quality in both code and process so that they can provide the quality products that their customers and their business require.

Net Objectives stands ready to help with this effort with training and coaching services in Lean and Agile Software Development.

For more information on courses in this series, please visit www.netobjectives.com/services, especially the Test-Driven Development and Design Patterns services.

You will also find helpful reading materials and recommended readings at www.netobjectives.com/resources.

If you would like to talk with someone, send a note to mike.shalloway@netobjectives.com or call **1-888-LEAN-AGILE**. We are happy to help however we can.

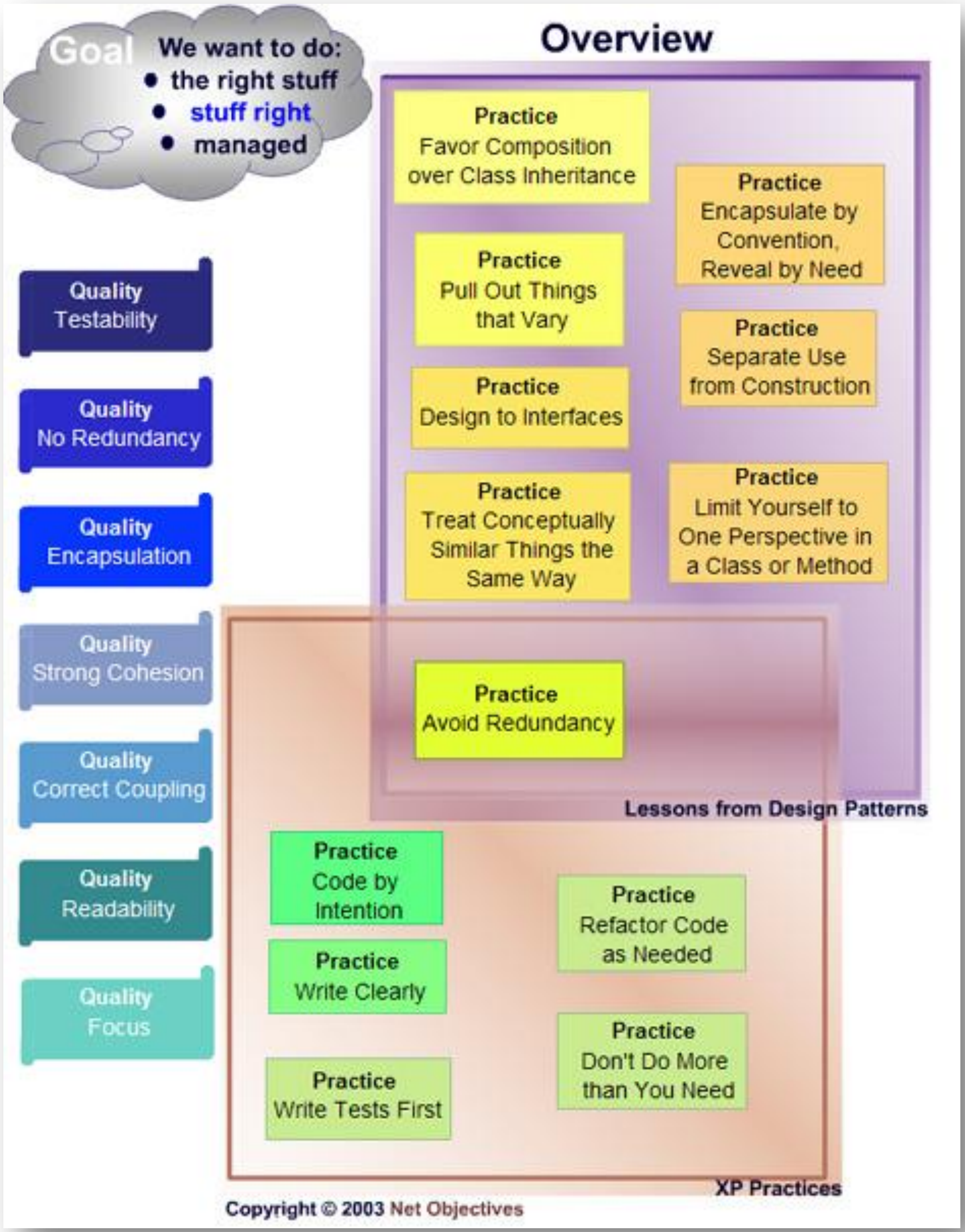
Goal
We want to do:
the right stuff
stuff right
managed
Overview
Quality Testability
Quality No Redundancy
Quality Encapsulation
Quality Strong Cohesion
Quality Correct Coupling
Quality Readability
Quality Focus
Practice Favor Composition over Class Inheritance
Practice Encapsulate by Convention, Reveal by Need
Practice Pull Out Things that Vary
Practice Separate Use from Construction
Practice Design to Interfaces
Practice Treat Conceptually Similar Things the Same Way
Practice Limit Yourself to One Perspective in a Class or Method
Practice Avoid Redundancy
Lessons from Design Patterns
Practice Code by Intention
Practice Refactor Code as Needed
Practice Write Clearly
Practice Don't Do More than You Need
Practice Write Tests First
XP Practices
Copyright © 2003 Net Objectives

Figure 1: The Thirteen Code Quality Practices

The Code Quality Practices

The following practices are involved in code quality. The first five are very easy to learn and apply in your organization.

Write Tests First

Writing tests first is sage advice. However, it is not the same as what many developers normally think about testing. Testing is normally thought about as a way of verifying that software works the way it is supposed to. This is the standard approach: write the software, get it working, and then verify that it does. This approach limits testing to pure verification.

Other things happen when tests are written before (or concurrently with) the code. Since the software being "tested" does not exist yet, developers cannot think in terms of how the code is implemented (it has not even been designed yet). Instead, one must think in terms of what the software is supposed to do and how a client object would communicate with it. This type of thinking is called "designing to interfaces" and is one of the fundamental practices (and not coincidentally one of the main design philosophies of the Gang of Four).

This is one of the ways that writing tests either before or concurrently with the writing of the code incorporates design. It is also consistent with Bertrand Meyers "design by contract" philosophy. This is because the tests you are writing are actually the contracts that you have decided that the objects must implement.

It should not be surprising that code that is more testable is also more maintainable and easier to understand. Let's look at this correlation to get a deeper understanding of why following the mandate to "write tests first" improves code. Consider a program that has to get the status of something, format it into a string, encrypt this information and then transmit it somehow. Consider how to test this software. You will need tests for:

- Getting the status
- Formatting it
- Encrypting it
- Transmitting it

Tests will be much easier to write if you can test these functions individually. This means you must at least have them in separate methods (strong method cohesion). It probably also means having the encryption and transmission functions be in different classes than the class getting the status. This enables you to test these functions without the "status" class being instantiated – making generating our tests easier. At least in this case, considering how to write tests suggests that you will have strong cohesion at both the method level and class level.

Of course, you could just consider the testability of objects and then write the code to pass these tests. Unfortunately, just thinking about testing is not really enough. Writing the tests serves two other purposes. It verifies your thinking, and ensures that you know what the class being tested is supposed to do under various circumstances. For example, maybe you are using a particular provided collection object which you "know" keeps things sorted. Until you write and run your test (or a customer sees a non-sorted collection) you will not be validating your understanding. Secondly, if you write your tests in an automatic framework, you can run them automatically whenever you change your code. This maintains code quality and allows you to refactor your code with confidence. Otherwise you are playing a kind of Russian roulette when you refactor your code. How will you know if you've broken anything if you don't rerun all of your tests?

Code by Intention

Coding by intention is the practice of pretending a piece of function you need already exists and is in the form you want it to be in. In other words, if you find that you have to write some function, instead of coding it in the place you need it run, put in a call to method you pretend exists. In the example in the "Write Tests First" section, this would mean writing the method as follows:

```
s= getStatus();
s= formatStatus(s);
s= encryptString(s);
transmit(s);
```

This is much better than writing one long method that is all implementation. This gives an easy to read method that describes what is happening. Note that each of these methods has strong cohesion and should be correctly coupled because no knowledge of how it is being implemented is being taken advantage. Since each method has a well-defined interface with a clear contract, each of these should also be relatively easy to test.

Coding by intention assists:

- **Testability** because a well-defined intention is easier to test.
- **Cohesion** because the intention should be about one thing, resulting in method cohesion
- **Encapsulation** because it encapsulates the implementation of the method being defined
- **Correct coupling** because the calling routine is written only to the interface
- **Readability** because functions are contained in well-named methods.

It may not be quite as clear that it also helps eliminate redundancy, but in practice it does. This is because coding by intention results in stronger cohesion which makes it easier to identify when a redundancy has been introduced.

Write Clearly

Writing clearly means to consider how your code will be read. Key aspects to writing clearly are:

- Use intention revealing names
- Follow coding standards

Using intention-revealing names helps ensure that people understand what the method/class does. Ward Cunningham once said that people will assume the meaning of a method from its name until they find out otherwise. You have to think about this for a second to realize that they will find out when the code doesn't work as expected - an expensive lesson.

Following coding standards makes code easier to read because people get used to a consistent standard across the project.

Encapsulate by Convention, Reveal by Need

It is crucial to encapsulate when possible. Encapsulation occurs at several different levels. The most commonly known one is "data-hiding". But this is just one type of encapsulation. The major types of encapsulation are:

- Data
- Implementation
- Type
- Construction
- Design

“Data encapsulation” (most commonly referred to as “data-hiding”) means to not let other classes access the data members of a class. This is most easily accomplished by using private data accessors. This also requires get and set methods to access the information. However, it also allows for changing how the state of the object is kept without affecting other code.

“Implementation encapsulation” means to not let a client object know how a method is implemented. This requires client objects to just refer to the interface of the method being used.

“Type encapsulation” means to hide the type of object actually being used. This is accomplished through the use of interfaces and abstract classes. Done properly, the client object doesn’t know (nor care) what type of object it is actually using. It refers to the object it has through the interface the object implements. This is the type of encapsulation referred to when they say to “encapsulate the concept that varies”.

“Construction encapsulation” means to hide how an object is constructed. A common practice to instantiate objects is through the keyword “new”. However, if the way an object is constructed changes, everywhere new is used to instantiate one must be changed. A simple, virtually no-cost, solution, is to use protected constructors with public static methods that call the constructor. The code segment below shows an example of this:

```
class ClassName {
      protected ClassName () { . . . }
      static public ClassName getInstanceOfClassName () {
            return new ClassName();
      }
}
```

While this does not have any advantage at this stage, notice how the constructor of `ClassName` can change without having to change any of the objects that need new instances of `ClassName`. Several design patterns take advantage of this approach (e.g., Singleton and Object Pooling).

"Design encapsulation" means that when one object uses another, the used object may actually be hiding a complete design behind it. A simple example of this is shown in Figure 2, below. In that figure, in the case shown on the right, the `Client` object has no idea that it is actually taking advantage of several objects. The "design" (the `Chip` object using one of several `Encrypt` objects) is hidden from it.

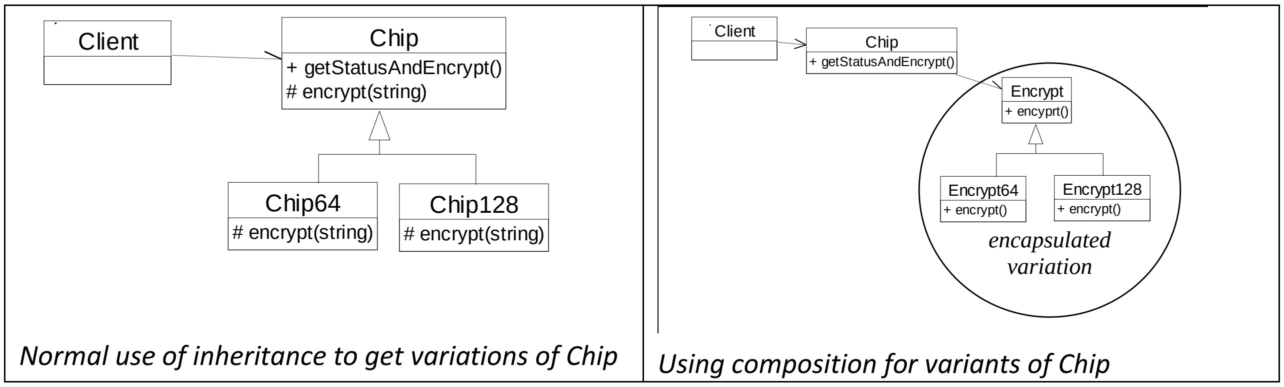

Figure 2: Composition and Encapsulation

Avoid Redundancy

"Avoid Redundancy" is also known as the "once and only once" rule. Developers know that duplicating code is not a good idea. However, all too often, they will only eliminate duplication if it takes virtually no effort to do so. A common way that code gets duplicated is when a new requirement comes up that can be satisfied by taking an existing module and making a relatively small change. Many developers cut and paste the current code, make the small change necessary and leave it at that. Unfortunately, the resulting code is essentially a duplicate of the first. An approach to consider would be to make the existing code work in either situation. This may require information being passed into it and/or an if/switch and is therefore not always done. However, when handled this way, no duplication of code exists.

Do Not Do More Than You Need

"Do not do more than you need" is a version of Extreme Programming's slogan, "Do the simplest thing possible". Stated this way makes it very clear: not doing more than you need springs from the idea of not having too many projects at one time. What happens when you do too much?

- Features are implemented that are never used
- Bells and whistles are added because developers were "sure" they were useful
- Frameworks handle more than is ever required of them

Certainly architecture is important and you should think ahead. The difficulty is that developers often think much *too far ahead* (or not ahead at all). Doing something you need now is a good idea. Doing something you *may* need in the future is often a waste of time. What is worse is that it often causes the design problem to greatly increase in magnitude. Over-designed software can be just as problematic as under- designed software.

Following the "do not do more than you need" mandate improves code quality because:

- You might not actually need it (YAGNI), thereby saving time to work on things you do need.
- If you do need it in the future, you may be smarter at that point when you have to implement it.
- Not doing what is not required *now* helps you focus on those things that *must* be done now.
- If you do need it later, more of the system will then exist so writing the function will be easier.

Pull Out Things That Vary

When a class begins to have more than one way to do something it is usually a good idea to hide this variation. An easy way to do that is to localize it in a separate class. The original class then refers to this new object. The result is strong cohesion and (possibly, if done correctly) proper coupling.

Treat Conceptually Similar Things the Same Way

This is part of the Gang of Four's rule, "find what varies and encapsulate it". This means if a client object uses several different objects that do conceptually the same thing, you should hide their variations from the client object so it can use them all in the same manner. For example, suppose you have to implement three different types of encryption. If you treat them all in the same manner, a client object can use them without worrying about which one is actually present. This, of course, implies a common interface to these different implementations. If the different implementations don't start out identical, the suggestion would be to make them identical. This may mean having a "fatter" interface than is really needed. That is, an interface with more parameters than all of the implementations actually need. However, by taking the extra work to accomplish this, the client object can be simpler and should also be able to take advantage of new server objects that implement this interface.

Following this practice forces you to look at the conceptual nature of these separate implementations (variations). It simplifies things because it splits things up into:

- The variations
- How to deal with all of the variations in a consistent way

This conceptual view often results in more modifiable (robust) code because new variations can be implemented without having to modify the client object.

Favor Composition over Class Inheritance

"Favor Composition over Class Inheritance" is perhaps the defining aspect of the Gang of Four's approach to design. It says that if you need to vary behavior, do not do so by deriving a special case from your current class. Instead, take the behavior that already exists in the class and put it in its own class. Contain this new class. This is called "composition."

Now in this practice, it is very common that this contained behavior needs to change and therefore you use inheritance to set up a common interface that all of these variations can derive from. This enables the using (containing) class to deal with all of the variations the same way. While inheritance is used by the contained class, it is being used a different way than using inheritance to get new variations.

From this vantage point, inheritance is not used to specialize a concrete class. Instead, it is used to categorize a set of similar classes as the same general type. Composition is used to vary a behavior by providing one class with a pointer to another, the second of which is abstract. The resulting behavior is determined by whichever of the concrete classes is *actually* pointed to.

Design to Interfaces

Design to interfaces means to define the interfaces of objects without regards to how to implement them. This effectively hides the implementation of the class. This supports proper coupling because implementations aren't known. Coding by intention is a systematized way of designing to interfaces.

Separate Use from Construction

If you encapsulate type many of our objects won't really know what types they are dealing with. However, somewhere in the system, this needs to be known.

When designing a class, two issues must be considered:

- How instances of the class will be used (the public interface of the class
- How instances will be built

Throughout the design process, these issues should be treated as separate responsibilities, handed by separate entities, and construction should always be considered last. This typically results in the use of object factories that are separate from the clients that used the instances once they are built.

Fortunately, it is easy to separate construction of objects and use them.

Refactor Code as Needed

"Refactoring" is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written."[1] Assuming you know when you mean by "quality code", refactoring gives a way to get there, even if you are not there already.

Refactoring is used in Test-Driven-Development as the way to correct your designs after implementing some function and then realizing how it should have been written. It should also be used in design-up-front methodologies to keep you code clean as the implementation evolves.

All developers should know at least the following types of refactoring:

- Extract Method
- Move Method
- Rename Method
- Convert Switch to Polymorphism

Limit Yourself to One Perspective in a Class or Method

There are two ranges of perspective: construction/use and conceptual/implementation. Limiting yourself to one perspective in a class or method means to have a class or method operate either at the conceptual level or implementation level (all classes/methods have specifications). This helps cohesion and understandability. It also supports many of the other approaches because it reinforces commonality / variability analysis (organizing your entities by concepts and their variations). It also means to be dealing with either construction or with use, but not both. This is mirrored in the practice of separating use from construction.

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley. 1999