Chapter 1
Programming by Intention

Everything old is new again. The folks who brought us the eXtreme Programming books[1] were, among other things, promoting a set of best practices in software development. One of them, which they termed "Programming by Intention", was not actually new, but was something that had been a very common coding technique in languages like COBOL and Smalltalk (usually called "top down" programming) years before. That's actually a good thing; time-tested practices are often the most credible ones, because they've proven their value over and over again under realistic circumstances. In this chapter we'll examine this practice, first by simply demonstrating it, and then by investigating the advantages we gain by following it. Finally, we'll discuss it as it relates to testing and testability, and design.

Programming by Intention: A Demonstration

You need to write some code. It's just a service that takes a business transaction and commits it. You've decided (rightly or wrongly) to simply create a single object, with a simple public method that does the work.

The requirements are:

- The transaction will begin as a standard ASCII string
- The string must be converted to an array of strings, which are tokens in the domain language being used for the transaction
- Each token must be normalized (first character uppercase, all others lowercase, spaces and non alpha-numeric characters removed)
- Transactions of more than 150 tokens should be committed differently (using a different algorithm) than smaller transactions, for efficiency
- The API method should return a true if the commit succeeds, a false if it fails

We're leaving out some details -- such as what the commitment algorithms actually are -- to focus narrowly on the practice we're interested in here.

In learning to write in a programming language, you train your mind to break problems down into functional steps. The more code you write, the better your mind gets at this sort of problem-solving. Let's capitalize on that.

As you think about the problem above, each bullet point represents one of these functional steps. In writing your code, you will have the *intention* of solving each one as you go. Programming by Intention says: rather than actually writing the code in each case, instead *pretend* that you already have an ideal method, local in scope to your current object, which does

[1] Among them are Kent Beck, Cynthia Andres, Martin Fowler, James Newkirk, Robert Martin, Ron Jeffries, Lisa Crispin, Tip House, Ann Anderson, and Chet Hendrickson

precisely what you want. Ask yourself "what parameters would such an ideal method take, and what would it return? And, what name would make the most sense to me, right now, as I imagine this method already exists?"

Now, since the method does not actually exist, you are not constrained by anything other than your intentions (hence, you are "programming by" them). You would tend to write something like this:

```
public class Transaction {
      public Boolean commit(String command) {
            Boolean result = true;
            String[] tokens = tokenize(command);
            normalizeTokens(tokens);
            if(isALargeTransaction(tokens)) {
                  result = processLargeTransaction(tokens);
            }
            else {
                  result = processSmallTransaction(tokens);
            }
            return result;
      }
}
```

The commit() method is the defined API of our object. It's public, of course, so that it can serve up this behavior to client objects. All of these other methods (tokenize(), isALargeTransaction(), processLargeTransaction(), processSmallTransaction()) are not part of the API of the object, but are simply the functional steps along the way. They are often called "helper methods" as a result. For now, we'll think of them as private methods although, as we'll see, that's not always literally true. The point is: their existence is part of the internal implementation of this service, not part of the way it is used from the outside.

And, they don't really exist, yet. If you try to compile your code, naturally the compiler will report that they do not exist (we like that, though… it's a sort of to-do list for the next steps). They have to be created for the code to compile, which is what we must do next[2].

In writing this way, we allow ourselves to focus purely on how we're breaking the problem down, and other issues of the overall context we're in. For instance, we have to consider if the String array being used is, in our implementation language, passed by reference or passed by copy (obviously we're imagining a language where they are passed by reference, otherwise we'd be taking tokens back as a return). What we don't think about, at this point, are implementation details of the individual steps.

So what's the point? There are actually many advantages gained here, which we'll investigate shortly, but before we do that let's acknowledge one important thing: this is not hard to do. This is not adding more work to our plate. The code we write is essentially the same that we would have written if we'd simply written all the code into one big method (like our "programs" back in the day; one big stream of code logic). We're simply writing things in a slightly different way, and in a slightly different order.

That's important. Good practices should, ideally, be things you can do all the time, and can promote across the team as something that should always be done. This is only possible if they are very low cost, essentially free to do.

[2] Or someone else will. Sometimes breaking a problem up like this allows you to hand out tasks to experts. Perhaps someone on your team is an expert in the domain language of the tokens; you might hand off the tokenize() and normalize() methods to them.

Advantages

So again, what's the point of programming in this way?

Something so terribly simple actually yields a surprising number of beneficial results while asking very little, almost nothing of you in return. We'll summarize these benefits in a list here, and then focus on each individually.

If you program by intention, your code will be:

- More cohesive (single-minded)
- More readable and expressive
- Easier to debug
- Easier to refactor/enhance, so you can design it minimally for now
- Easier to unit test

And, as a result of the others – easier to modify/extendIn addition:

- Certain patterns will be easier to "see" in your code
- You will tend to create methods that are easier to move from one class to another

Method Cohesion

One of the qualities of code that tends to make it easier to understand, scale, and modify is cohesion. Basically, we like software entities to have a single-minded quality, to have a single purpose or reason for existing.

Classes, for instance. A class should be defined by its responsibility, and there should be only one general responsibility per class. Within a class are methods, state, and relationships to other objects that allow the class to fulfill its responsibility. Class cohesion is strong when all the internal aspects of a class relate to each other within the context of the class' single responsibility.

You might argue in our example above, that some of the things we're doing are actually separate responsibilities and should actually be in other classes. Perhaps; this is a tricky thing to get right[3]. However, even if we don't always get that right, we can get another kind of cohesion right at least most of the time if we program by intention.

Method Cohesion is also an issue of singleness, but the focus is on function. We say a method is strongly cohesive if the method accomplishes one single functional aspect of an overall responsibility.

The human mind is pretty single-threaded. When people "multi-task", the truth is usually that they are actually switching quickly between tasks; we tend to think about one thing at a time. Programming by intention capitalizes on this fact, allowing the singleness of your train of thought to produce methods that have this same singleness to them.

This cohesion of methods is a big reason that we get many of the other benefits of programming by intention.

Readability and Expressiveness

Looking back at our initial code example, note how readable it is.

---

[3] …which is not to say we don't have some help for you on class cohesion. See our chapter called "Define Tests Up-Front" for more on this.

```
public class Transaction {
      public Boolean commit(String command) {
            Boolean result = true;
            String[] tokens = tokenize(command);
            normalizeTokens(tokens);
            if(isALargeTransaction(tokens)) {
                  result = processLargeTransaction(tokens);
            }
            else {
                  result = processSmallTransaction(tokens);
            }
            return result;
      }
}
```

The code essentially "says":

We are given a command to commit. We tokenize the command, normalize the tokens, and then depending on whether we have a large set of tokens or not, we either process them using the large transaction mechanism, or the small one. We then return the result.

Because we are not including the "how" in each case, only the "what", we can examine the process with a quick read of the method, and easily understand how this all works. Sometimes, that's all we want, to quickly understand how something works.

This is readability, but it is also expressive Note that we did not include any comments in this code, and yet it's still easy to "get". That's because those things that we would have included in comments are now instead the actual names of the methods.

Comments are expressive too, but the problem with them is that they are ignored by the compiler[4] and often by other programmers who don't trust them to be accurate. A comment that has been in the code for a long time is unreliable, because we know that the code may have been changed since it was written, and yet the comment may not have been updated. If we trust the comment, it may mislead us, and there is no way to know one way or the other. We are forced to investigate the code, and so the expressiveness evaporates. Hence, comments may be ignored by programmers as well, making them less than useful[5].

The central, organizing method in programming by intention contains all the steps, but very little or no actual implementation. In a sense, this is another form of cohesion; the process by which something is done is separated from the actual accomplishing of that thing.

Another thing that tends to make code both readable and expressive is that the names of the entities we create should express the intention we had in creating them. When methods are cohesive, it is easy to name them with a word or two that comprehensively describes what they do, without using lots of underscores and "ands" and "ors" in the names. Also, since we name the methods before they actually exist, we tend to pick names that express our thinking. We call names of this type "intention revealing names" because they reveal the intention of the name. We want to avoid picking names that make sense after you understand what the function does but can be easily misinterpreted before its intention is explained by someone who knows what is going on..

[4] Not all comments are to be avoided, however. If the comment exists to make the code more readable, change the code to make it more readable on its own. If the comment exists to explain the purpose of a class or method, or to express a business issue that drives the code, these can be very helpful to have.

[5] They are less than useful because sometimes someone believes them when they are wrong.

Comments as a Code Smell

While we're not claiming that you shouldn't write comments, certain comments are actually a type of code smell. For example, let's say you had written something like this:

```
public class Transaction {
      public Boolean commit(String command){
            Boolean result = true;
            Some code here
            Some more code here
            Even some more code here that sets tokens
            Some code here that normalizes Tokens
            Some more code here that normalizes Tokens
            Even more code here that normalizes Tokens
            Some code that determins if you have a large transaction
            Set lt= true if you do
            if (lt) {
                  Some code here to process large transactions
                  More code here to process large transactions
            }
            else {
                  Some code here to process small transactions
                  More code here to process small transactions
            }
            return result;
      }
}
```

You might look at this and say – "wow, I don't understand it, let's add some comments" and you'd create something like this:

```
public class Transaction {
      public Boolean commit(String command){
            Boolean result = true;

            // tokenize the string
            Some code here
            Some more code here
            Even some more code here that sets tokens

            // normalize the tokens
            Some code here that normalizes Tokens
            Some more code here that normalizes Tokens
            Even more code here that normalizes Tokens

            // see if you have a large transaction
            Some code that determines if you have a large transaction
            Set lt= true if you do
            if (lt) {
                  // process large transaction
                  Some code here to process large transactions
                  More code here to process large transactions
            }
            else {
                  // process small transaction
```

```
                Some code here to process small transactions
                More code here to process small transactions
            }
            return result;
        }
}
```

Note that you've inserted comments that describe what is going on after writing the code.. These comments would not be needed if we had programmed by intention. The methods we've used in place of the comments are more useful because they *have* to be up to date in order to compile.

Debugging

In most of the courses we teach at Net Objectives, somewhere along the way we'll ask people if they think they spend a lot of time fixing bugs. Unless they are already a student of ours, they'll tend to say yes, that's a significant part of what makes software development tricky[6].

We point out, however, that debugging really consists largely of *finding* the bugs in a system, whereas fixing them once they are located is usually less problematic. Most people agree to that almost immediately.

So, the real trick in making code that you will be able to debug in the future is to do whatever you can to make bugs easy to find. Naturally, you should try to be careful to avoid writing them in the first place, but you can only be so perfect, and you're probably not the only person who'll ever work on this code.

When you program by intention, the tendency is to produce methods that do one thing. So, if something in your system is not working you can:

1. Read the overall method to see how everything works, and
2. Examine the details of the helper method that does the part that's not working.

That's almost certainly going to get you to the bug more quickly than if you have to wade through a big blob of code, especially if it contains many different, unrelated aspects of the system.

Legacy systems, for example, are tough to debug, and there are many reasons for this. One big one, however, is that often they were written in a monolithic way, and so you end up printing out the code, breaking out the colored hi-lighters, and marking code blocks by what they do…. "I'll mark the database stuff in yellow, the business rules in blue, etc…" It's laborious, error-prone, boring, and not a good use of a developer's time.

Let the computer do the grunt work.

Refactoring and Enhancing

It's hard to know exactly how far to go in design, how much complexity to add to a system in your initial cut at creating it. Because complexity is one of the things that can make a system hard to change, we'd like to be able to design minimally, adding only what is really needed to make the system work.

However, if we do that we're likely to get it wrong from time to time, and fail to put in functionality that is actually needed. Or, even if we get it right, the requirements of our

[6] If they are students of ours, they've heard this question from us before – we're not claiming our students don't write bugs.

customers, stakeholders, or the marketplace can change the rules on us after the system is up and running.

Because of this we often have to:

- Refactor the system (changing its structure while preserving its behavior)
- Enhance the system (adding or changing the behavior to meet a new need)

Refactoring is usually thought of a "cleaning up" code that was poorly written in the first place. Sometimes it is code that has decayed due to sloppy maintenance, or changes made under the gun without enough regard to code quality. Refactoring can also be used to improve code once it is clear it should have been designed differently after more is known about the progam.

Martin Fowler wrote a wonderful book in 1999 called Refactoring[7] that codified the various ways in which these kinds of behavior-preserving change can be made, and gave each way a name (often called a "move").

One of the refactoring moves that most people learn first when studying this discipline is called "Extract Method"; it takes a piece of code out of the middle of a large method, and makes it into a method of its own, calling this new method from the location where the code used to be. Because temporary method variables also have to move, and so forth, there are a number of steps involved.

Many of the other refactoring moves in the book begin by essentially stating "before you can do this, you must do Extract Method over and over until all of your methods are cohesive". However, you'll find if you program by intention, you've already done this part. In his book Prefactoring[8], Ken Pugh examines extensively how simple, sensible things like programming by intention can help you.

If you know that your code has routinely been "prefactored" in this way, your expectation about how difficult it will be to refactor it in other ways will be ameliorated, because code that already has method cohesion is simply easier to refactor.

Similarly, programming by intention can make it easier to enhance your system as well. Let's go back to our transaction processing example

Imagine that six months after this code was put into production, a new requirement is added to the mix: due to the way some 3rd party applications interact with the transaction processing, we have to convert some tokens (there's a list of a dozen or so) from an older version to the one supported by our system. The notion of "updating" all the tokens is now something we must always perform in case the command string contains deprecated tokens from the domain language.

The change here would be reasonably trivial, and could be made with a high degree of confidence:

```
public class Transaction {
      public Boolean commit(String command){
            Boolean result = true;
            String[] tokens = tokenize(command);
            normalizeTokens(tokens);
            updateTokens(tokens);
            if(isALargeTransaction(tokens)){
                  result = processLargeTransaction(tokens);
            }
```

[7] Fowler, Martin et al. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999

[8] Pugh, Ken. Prefactoring. Cambridge, Ma: O'Reilly Media, Inc, 2005

```
            else {
                  result = processSmallTransaction(tokens);
            }
            return result;
      }
}
```

The next step would be to write the updateTokens() method, but in so doing we note the likelihood of doing any damage to the code in the rest of the system is extremely low. In fact, making changes to any of the helper methods can be done with a fair degree of confidence that we are changing only what we intend. Cohesion tends to lead to encapsulation[9] like this.

Unit Testing

In programming by intention, we're not trying to broaden the interface of the object in question, rather we're ensuring that we're defining the interface prior to implementing the code within it. In fact, we want to follow the general advice on design that patterns promote insofar as we'd like the clients that use our service to be designed purely to its interface, and not to any implementation details.

So, at least initially, we'd like to keep all these "helper methods" hidden away, as they are not part of the API of the service, and we don't want any other object, now or in the future, to become coupled to them (the way they work, or even that they exist at all). We'd like to be able to change our mind in the future about the exact way we're breaking up this problem, and not have to make changes elsewhere in the system where this object is used.

However, it would seem to work *against* testing this object, if we make all the helper methods private:

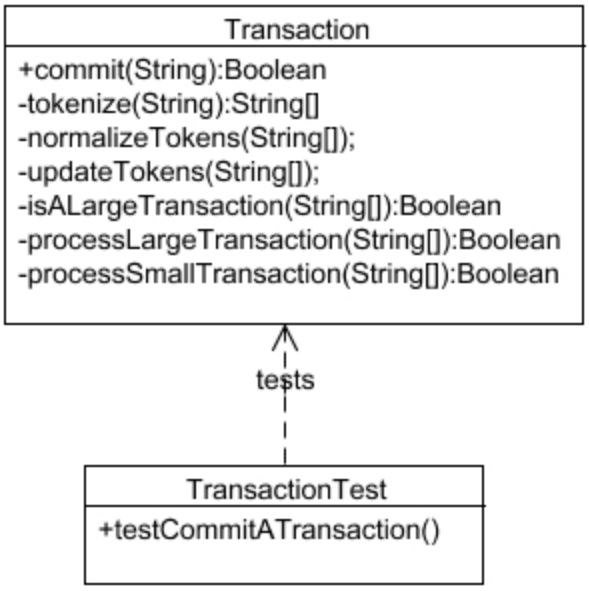


Figure 1.1
Programming by Intention, Private Methods

[9] See the chapter "Encapsulate That!" for more information on this.

Private methods cannot be called by the unit test either, and so the only test we can write is of the commit() method, which means we have to test the entire behavior in a single test. Such a test may be more complex than we want, and also we'll be writing a test that could fail for a number of different reasons, which is not what we'd prefer.[10]

If we can solve this conundrum, however, note that separating the different aspects of this overall behavior into individual methods makes them, at least in theory, individually testable, because they are not coupled to one another. Much as the client of this class is coupled only to its interface, the API method is coupled to the helper methods only though their interfaces.

So, how do we deal with the un-testability of private methods? There are three general possibilities:

1. We don't test them individually, but only through the commit() method. In unit testing, we want to test *behavior* not *implementation*, and so if these helper methods are really just steps in a single behavior, we don't want to test them. We want to be able to refactor them (even eliminate them) and have the same tests pass as before we refactored.
2. We need to test them individually, as a practical matter. Even though they are "just the steps", we know there are vulnerabilities that make them somewhat likely to fail at some point. For efficiency and security, we want to be able to test them separately from the way they are used. In this case, we need to get a little clever, and use testing tricks. These can be very language-dependent, but include: making the test a "friend" of the class (in C++), wrapping the private method in a delegate and handing the delegate to the test (also known as a Testing Proxy) (in .Net), making the methods protected and deriving the test from the class, and so on. Beware the overuse of these tricks, however; not all developers will readily understand what you have done, they often don't port well from one language/platform to another, and we don't want to couple our tests tightly to implementation unnecessarily.
3. Maybe these helper methods are not merely "steps along the way" but are, in fact, different behaviors that deserve their own tests. Maybe they are *used* in this class, but they are, in implementation, entirely separate responsibilities. Note that the desire to test this class is forcing us to look at this "are they really just steps?" issue, which is an important aspect of design. If this is the case, then we need to revisit our design… for example, let's say that we determine that the step that normalizes the tokens is really something that legitimately should be tested on its own. We could change the design just enough:

[10] See our chapter called "Define Tests Up-Front" for more information on this issue.

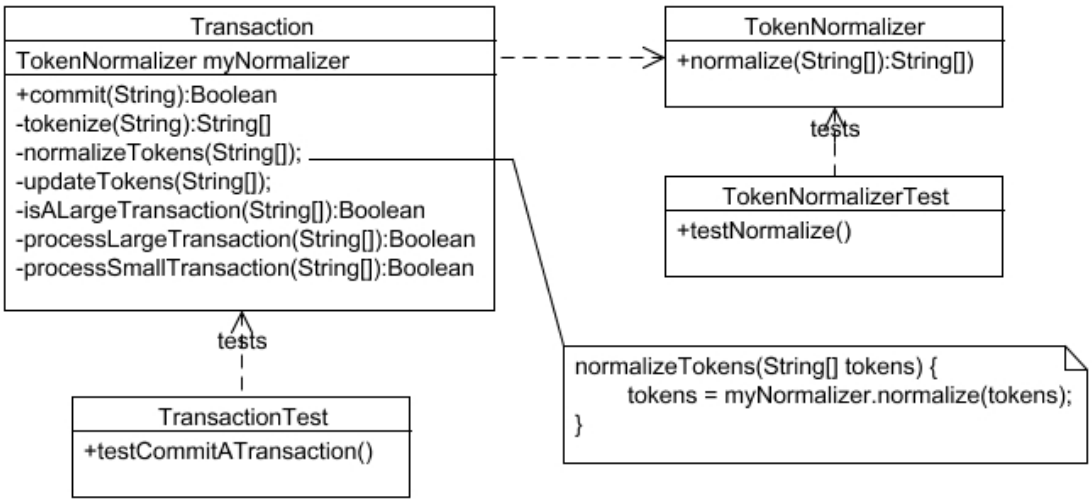


Figure 1.2
Design Change

You'll note that the *use* of the TokenNormalizer is private. Nothing outside Transaction is coupled to the fact that the TokenNormalizer is used there[11]. However the *implementation* of the TokenNormalizer is accessible through its own API, and is thus testable on its own.

Also, you'll note that the fact that the normalization code was in a method by itself in the first place makes it pretty easy to pull it out now into its own class. Extracting a class is usually fairly trivial if methods are cohesive from the get-go. We're making this decision just as we need it, based on the realization that the desire to test is providing us.

Another reason we might have seen this would be if another entity in the system also had to perform token normalization. If the algorithm is stuck in a private method in Transaction, it's not usable in another context, but if it is in a class by itself, it is. Here the desire to avoid code duplication[12] is leading us to this same just-in-time decision.

Easier to Modiy/Extend

Given the above improvements to the quality of the code as a result of programming by intention, it should be evident that modifying and extending your code should be easier to do. We know from experience that we will need to modify our code. It's also known from our experience that we don't know exactly what these modifications will be. Programming by intention gives us a way to set our code up for modification while paying virtually no price for doing so.

Seeing Patterns in Your Code

We do a lot of design patterns training at Net Objectives, and we routinely write and speak on them at conferences, etc… Invariably, once we are seen to be "pattern guys", someone will say "the patterns are cool, but how do you know which one to use in a given circumstance?"

---

[11] This does beg the question "how does Transaction obtain an instance of TokenNormalizer? See our chapter "Separate Use from Construction" for a discussion on this issue.

[12] See our chapter "Shalloway's Law" for a more thorough discussion on this concept.

The answer to this question leads to a very long and (we think) interesting conversation, but sometimes you can see the pattern in your implementing code if you program by intention.

Let's alter our example, slightly.

Let's say there are two completely different transaction types which go through the same steps (tokenize, normalize, update, process), but implement all of these steps differently, not just the processing step. If you programmed each of them by intention, while their implementing "helper" methods would all be different, the commit() method would look essentially the same. This would make the Template Method Pattern[13] sort of stand up and wave its hands at you:

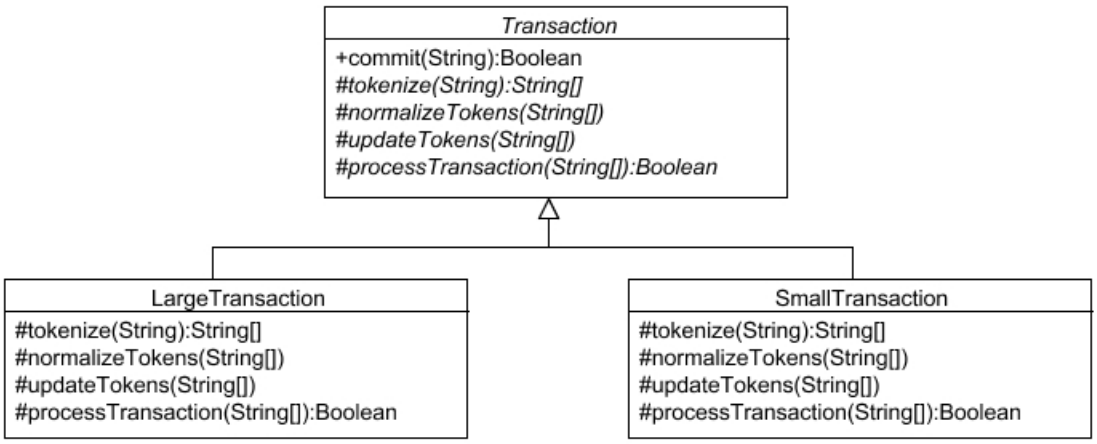


Figure 1.3
The Template Method Pattern

Taking it a bit farther, and going back to the testability issue, we might start pulling one or more of these behaviors out, to make them testable, and in so doing discover opportunities for the Strategy Pattern. As before, we'll pull out normalization:

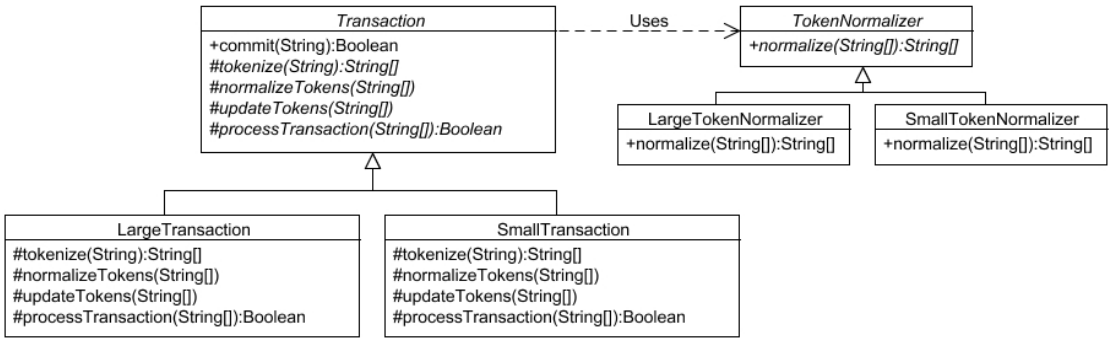


Figure 1.4
The Template Method and Strategy Patterns

[13] If you don't know the Template Method, pay a visit to http://www.netobjectivesrepository.com/TheTemplateMethodPattern

Movable Methods

As we pointed out before, the cohesion of a class is related to the notion that a class should, ideally, have a single responsibility. It may require many methods, member variables, and relationships to other objects to fulfill that responsibility, but there should only be one reason it exists, or would have to be changed[14].

Programming by intention helps you to create cohesive methods, through the simple expedient of creating those methods based on your own ability to do functional decomposition, but it does not, directly, do very much about class cohesion. Indeed, you could easily suggest that our Transaction class, as we initially coded it, was not very cohesive.

One way to improve class cohesion is to move methods and other members that it really should not have to other classes, perhaps new classes, and thus focus the class more narrowly. So, while programming by intention does not address class cohesion directly, it makes it easier for the developer to do so, through refactoring, as cohesion issues arise.

Why?

One reason is something we've already seen; programming by intention produces methods that have a single function. It's easier to move such a method because it avoids the problem of moving a method that contains some aspects that should not move. If a method does one thing, if part of it needs to move, all of it needs to move.

However, there is another reason as well. Sometimes a method is hard to move because it directly relates to the state members that exist on the class. In moving the method, we have to move the state as well, or find some way of making it available to the method it its new home, which can sometimes create odd and confusing coupling.

We have noted that, when programming by intention, we tend to pass the method whatever it is we want it to use, and take the result as a return, rather than having the method work directly on the object's state. These methods move more easily because they are not coupled to the object they are in.

Going back to our example, we could have done this:

```
public class Transaction {
      private String[] tokens;
      public Boolean commit(String command){
            Boolean result = true;
            tokenize(command);
            normalizeTokens();
            if(isALargeTransaction()){
                  result = processLargeTransaction();
            }
            else {
                  result = processSmallTransaction();
            }
            return result;
      }
}
```

We have made the token array a member of the class, and all the methods that used to take it as a parameter will now simply refer to it directly. There is nothing in programming by intention that *forces you* not to do this (any more than it forces you to refrain from naming your methods

[14] This is also often called the "Single Responsibility Principle."

m1() and go()), but it's a more natural fit to your mind to pass parameters and take returns, and we have noticed this strong tendency in developers who adopt the practice.

Summary

One of the critical aspects of bringing any complex human activity to maturity is the discovery of important practices that improve the success of those engaged in it. Such practices should ideally:

- Deliver a lot of value
- Be relatively easy to do
- Be relatively easy to promote, and teach others to do
- Low risk
- Be universal (you should not have to decide whether to follow them or not, in a given circumstance)

Programming by intention is just such a practice. Give it a try! For most people, it only takes a few hours to realize that it's easy, fun, and makes their code better without extra work.

Business-Driven Software Development™

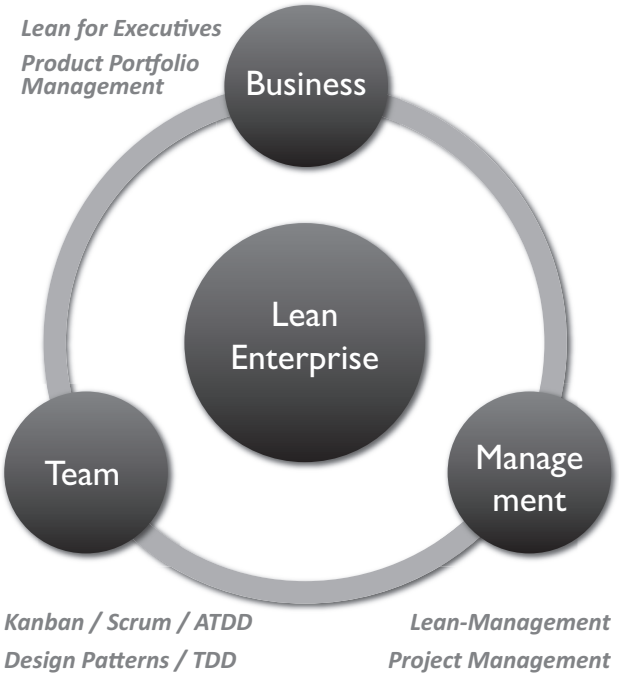
Assessments, Consulting, Training, and Coaching

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality



Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

Assessments

See where you are, where you want to go, and how to get there.

Business and Management Training

Lean Software Development

Product Portfolio Management

Enterprise Release Planning

Productive Lean-Agile Team Training

Team training in Kanban, Scrum

Technical Training in ATDD, TDD, Design Patterns

Roles Training

Lean-Agile Project Manager

Product Owner

For more information about Business Driven Software Development Services
Contact Mike Shalloway at Net Objectives: mike.shalloway@netobjectives.com / 404-593-8375
NetObjectives.com / 1-888-LEAN-244 (888-532-6244)