software development
The lifecycle starts here.

Getting Out of the Software Mud Pit

December 2000

If your software is quicksand and you're ready to add features and fix defects, you'll want to restore modularity through time-tested design patterns.

by Steve Adolph

A few years ago, a friend of mine took me four-wheeling in his Jeep. We had a great time, driving over old logging roads and occasionally getting stuck in muddy patches. With a bit of quick work on the gear box to rock the Jeep, we were able to quickly continue on our way. As we were driving along, we saw a large boggy section in the road that looked large enough to swallow a logging truck, never mind a little Jeep. My friend was confident that we could get through it easily, but no sooner had we driven into the mud pit than we sank down to the axles. Rocking and spinning only settled us deeper, with the mud oozing up through the gear shifter seal on the floor. We were stuck and we were stuck good.

My friend, speaking in the calm tone of an airline pilot explaining to passengers that everything is all right despite the flames shooting from the engine, assured me that we were OK. We weren't going to sink out of sight like some poor actor caught in B-horror-movie quicksand. If we planned our escape right, we might only get our shoes dirty. Otherwise, we would probably be scraping mud off of ourselves until the cows came home.

Ongoing software development (or what used to be called maintenance) is often a lot like trying to get out of a mud pit (or tar pit, as described by Fredrick Brooks in *The Mythical Man-Month*, Addison-Wesley, 1995). Small defects are easy to remove; a quick fix here and there keeps things running. But there is code that is so badly scarred by generations of quick fixes, so incohesive and so tightly coupled that it swallows all who fall into it. When you get stuck with one of these monsters, the situation you're in is much like what my friend and I experienced. Do it right and you may spend only a few late nights patching code. Do it wrong, and you'll be stuck in a cubicle picking out defects for an eternity.

In a previous article, I described how easily good software can go bad ("Four Wheel Drive, Garbage Barges and Objects," June 2000). Even when object-oriented programming techniques are apparently applied, software can become rigid, inflexible and incohesive. Rehabilitation requires the software to be reworked, but rework is often seen as a risky activity that adds no tangible benefit to the software.

Refactoring is a systematic approach to reworking the software to restore the software's modularity. We will explore how to exploit refactoring to help prevent your software from becoming a mud pit. Or, once you're in that mud pit, how you can use a systematic approach to get out and minimize the mud that gets splattered on you.

The Big Ball of Mud

Textbooks and articles describe the characteristics of high-quality software architecture. Many authors have captured good architectural principles as patterns with names like Layered Architecture, Master-Slave, Blackboard and Pipeline. Despite such a wealth of information demonstrating the characteristics of good software architecture, Brian Foote and Joseph Yoder, of the Department of Computer Science at the University of Illinois at Urbana-Champaign, suggest that the most prevalent software architecture is The Big Ball of Mud (*Pattern Languages of Program Design 4*, Addison-Wesley, 2000; the article is available at www.laputan.org/mud/mud.html).

"A Big Ball of Mud is haphazardly structured, sprawling, sloppy, duct-tape-and-bailing-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important

information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems."

It has been my experience that not only are such software architectures unpleasant to work with, they are economically unsustainable. With each haphazard change, we sink deeper into the quagmire. Subsequent modifications become more onerous. Eventually the software becomes too expensive and too unpredictable to maintain, and it must be rewritten-an extremely expensive and risky proposition.

Few developers start off the day saying, "Today, I think I'll write some crappy, tightly coupled, incohesive code." Messy software results from the repeated application of expedient changes that are made without regard to their impact on the overall modularity of the software.

Messy software is rehabilitated by reworking it to restore its modularity. Unfortunately, "rework" is a dangerous term in our industry. Many programmers or project managers spout one or more of the following justifications for avoiding rework to restore software modularity:

- *We have four-wheel drive, so we can't possibly get stuck!* Developers don't perceive a problem because they believe object technology hides their ugly secrets behind tidy interfaces; therefore, they don't have to worry about the consequences changes will have on the system's overall modularity.
- *We're stuck! Maybe if we spin the wheels something will happen.* Developers perceive a problem but don't know what action is required to improve the code's modularity.
- *We're not stuck. Drive on!* Managers can't see the benefit of the effort required to improve code modularity and believe the change is just gold-plating.
- *We're stuck, but spinning our wheels will only make things worse.* Managers can see the benefit of improving modularity, but the associated risks and the open-ended nature of the rework go beyond the manager's comfort level.

In my opinion, a powerful benefit of the recent rise of interest in refactoring is that it can remove many of the road blocks to reworking code by giving us a set of principles and a language for describing rework.

Refactoring

Refactoring is a both a noun and a verb. As a noun, refactoring is a behavior-preserving transformation aimed at improving code quality. "Behavior-preserving" means that no new features are added to the software by refactoring. Test results obtained after the refactoring is applied are the same as the results prior to the refactoring. As a verb, refactoring is the systematic approach to applying the transformation rules.

The origin of the term is lost deep in software antiquity, possibly coined in the mid-1980s by Smalltalk developers. In 1992, William Opdyke's University of Illinois at Urbana-Champaign doctoral thesis, "Refactoring Object-Oriented Frameworks," provided a theoretical basis for refactoring (see sidebar for an example). It is interesting to note that William Opdyke's supervisor was Ralph Johnson, one of the authors of what has become part of the software development canon, *Design Patterns* (Addison-Wesley, 1995). It is also interesting to note that design patterns are offered as refactorings. Recently, refactoring has been raised as a core practice of the increasingly popular Extreme Programming (XP) methodology. Refactoring was further enhanced by the recent publication of Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Addison Wesley, 1999).

Refactoring is not a new concept. Most of us have refactored code to some extent, although we may not have explicitly called it such. Most of us have referred to software rehabilitation as rework. So, what differentiates refactoring from rework?

Refactoring is a systematic approach to rework. There are limits on what you can do and when you should do it. First, refactoring keeps its objectives simple and doesn't add new features. Applied in its pure form, it's done solely to improve the modularity of software. Second, work done by Opdyke and others has provided a theoretical foundation for refactoring and established the rules for correctly applying refactorings. Next, rigorous and, preferably, automated testing ensures that refactoring is correctly applied. Combine rigorous testing with small well-defined steps and it's hard to get into trouble. Finally, from a management perspective, refactoring provides us with a vocabulary for discussing code problems and planning a course of corrective action.

Escaping the Mud

So how does all this help us get out of the mud? Before going any further, I want to emphasize the first and most important rule that you must apply to any project: Let sleeping dogs lie. If a piece of ugly, incohesive software is making you lots of money and customers are happy, why fix it? Refactorings are used to restore modularity because it makes subsequent changes to software easier and less error-prone. Refactorings are intended to make the software easier to work with and more "forward-compatible." Do not refactor a piece of software unless you are planning to add a new feature or to correct a defect. You have better things to spend your time and money on.

So now that you're planning to add a feature or fix a defect, when do you refactor? The XP philosophy drains the small mud puddles before they become huge, developer-swallowing mud pits. Each time a code fragment is changed, the developer reevaluates it. A simple rule of thumb for deciding when to apply refactorings is "three strikes and you refactor."

However, most of us aren't working in XP-friendly environments. Usually, we walk onto a project and inherit a deep, muddy swamp that has swallowed many of those who came before us. While there has been a great deal of emphasis on refactoring from a programmer's perspective, I would also like to emphasize the importance of refactoring from a manager's perspective.

A benefit of having a published set of refactorings is that it allows us to characterize what is wrong with a piece of software, what must be done to repair the software and what the benefit will be. Convincing a team lead or manager to let you rework a piece of software is at best difficult. Rework is a dangerous, open-ended word that a lot of software development managers don't like to hear. In my career, I have sat on both sides of the desk, as a software developer and as a software development manager. To a manager, rework means spending time tearing out working code and replacing it with new, unproven code. When a developer says to a manager, "I need to spend a few days cleaning this up," what the manager hears is "I'm ashamed to show this code to my fellow developers and I want to fix it so that my colleagues admire my skill. I have no idea how long this is going to take, or what the true scope of the change is. There is no immediate economic value in this change other than reaffirmation of my sense of self-worth." Naturally, the manager, who usually has a hot blow torch to her back from her boss to get the software out, will tell the developer "no."

Let's imagine a different conversation between the developer and manager. The developer comes up to the manager and says, "Before I add feature X to this system, I want to rework these classes using the extract hierarchy refactoring. Given the work that must be done for this refactoring, I estimate it will take three days to apply it. It will then take two days to add feature X. Without the refactoring it will take four days to add feature X. The refactoring will improve the modularity of the code and make it easier to add features to this part of the system in the future. You can have confidence that this will work because there is a large body of knowledge demonstrating how these refactorings improve software modularity. This is also the third time we've changed this piece of code, so it's likely to change again."

Now the manager hears what the developer is really saying and can make an intelligent, informed decision. The work is not open-ended; rather, the developer has clearly stated what must be done, why it must be done, what it will cost and what the benefits are. Furthermore, the refactoring framework of small behavior-preserving steps and rigorous unit testing helps reassure the software manager that after the refactoring the software will continue to do what it was supposed to do.

You Will Get Dirty

There are a number of ways to apply refactorings. In XP, you consider refactoring every time you make a change to your code. In an iterative software development process, like the Unified Process, you would plan your refactorings for the next iteration at the end of the current iteration. This kind of approach is like the quick rocking of a Jeep to get it out of small mud pits.

Legacy code that has been passed down through the generations, scarred by numerous expedient patches, is the mud pit in which you're going to sink up to your axles. There is no way to get around it. Improving the code quality is going to take days—perhaps weeks—of dirty, hard work. Refactoring offers you a viable approach to planning and dealing with the problem intelligently.

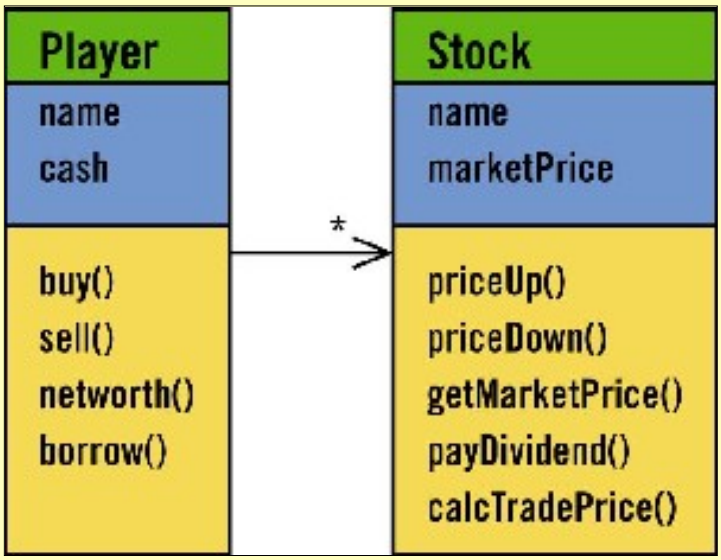
Fixing Feature Envy: An Example of a Minor Refactoring

When one class's object monopolizes the methods of another, it may never access its own member variables.

William Opdyke's 1992 doctoral thesis at the University of Illinois at Urbana-Champaign, "Refactoring Object-Oriented Frameworks," proposed some 26 minor and three major refactorings. Most of these were fairly simple in scope: create an empty class, create a method from a code fragment, move a method between classes, rename a method or class, and remove an unreferenced method or class. Opdyke referred to these simple transformations as minor refactorings. A major refactoring, on the other hand, is one that systematically applies a set of minor refactorings. An example of a minor refactoring follows.

"Feature envy" describes a situation in which an object of one class makes excessive use of the methods of another, usually the accessors and mutators (or getters and setters). Figure 1 shows a UML class diagram for a method of a Player and Stock class used in a stock-trading game. (The code fragment is listed below.) When a player buys stock, he specifies the name and number of units to purchase. The Player object needs to know the purchase price for the stock based on the current market price of the stock.

Figure 1.




```
          class Player
{
:
:
public void buy(String stockName, long units)
{
        Stock s = (Stock) m_stockList.get( stockName);
        // this calculation really wants to be in
        // the Stock class.
        long marketPrice = s.getMarketPrice();
        long purchasePrice = units * marketPrice;
        if (purchasePrice < m_cash)
// continue with purchase
```

The Player uses the Stock class's getMarketPrice() method to retrieve a stock's current market price and then performs the calculation. While this is a trivial example, it demonstrates a small mud pit where Player is more tightly coupled to Stock than it need be. This code fragment in the Player buy() method seems more interested in the members of Stock class than it is in its own members. In fact, the highlighted code fragment never makes use of the Player's member variables. In other words, this code fragment is envious of the Stock class's features.

The first refactoring that might be applied is what Opdyke calls Replace_Code_Segment_With_Function_Call, or what Martin Fowler calls Extract Method(110). The code fragment has no dependencies on the method's local

variables, therefore we can use this refactoring to extract the code fragment in question and place it into its own method.

The first refactoring that might be applied is what Opdyke calls Replace_Code_Segment_With_Function_Call, or what Martin Fowler calls Extract Method(110). The code fragment has no dependencies on the method's local variables, therefore we can use this refactoring to extract the code fragment in question and place it into its own method.

```
                class Player
{
:
:
long calcTradePrice(Stock s, long units)
{
        long marketPrice = s.getMarketPrice();
        return units * marketPrice;
}
public void buy(String stockName, long units)
{
        Stock s = (Stock) m_stockList.get( stockName);
        long tradePrice = calcTradePrice(s, units);

        if (tradePrice < m_cash)
// continue with purchase
```

Opdyke's thesis offers a nice proof of why this is behavior-preserving, but it should be clear just on inspection that running a test for a buy transaction before and after the refactoring will yield the same results. This was also a good move because the sell() method probably uses the same code fragment, so it helps reduce redundant code as well.

While we may have eliminated some duplicate code, we still haven't solved the feature-envy problem yet. For this, we apply another refactoring: Move_Member_Function or Fowler's Move Method (142), to move our newly created calcTradePrice() over to the Stock class.

```
class Player
{
:
:
public void buy(String stockName, long units)
{
        Stock s = (Stock) m_stockList.get( stockName);
        long tradePrice = s.calcTradePrice(s, units);
if (tradePrice < m_cash)
// continue with purchase
```

Again, Opdyke offers a proof of why this is behavior-preserving, and on inspection you can be pretty certain that the buy transaction test results will be the same.

One observation: The reason the method calcTradePriced() moved so easily from the Player to the Stock class is that it had no dependency on Player's member variables, a precondition for the refactoring. These preconditions are clearly stated as part of the refactoring and are to ensure the safety of the refactoring-that is, to ensure that the refactoring preserves behavior and does not introduce any interesting side effects. Had this not been the case, then either we couldn't apply the refactoring, or we would have to apply other refactorings to shuffle member variables around before applying Move_Member_Function.

—Steve Adolph