Evolutionary architecture and emergent design: Leveraging reusable code, Part 1

The relationship between code and design

Neal Ford

March 30, 2010

Once you identify idiomatic patterns in code, the next step is to harvest and use them. Understanding the relationship between design and code can facilitate the discovery of reusable code. This Evolutionary architecture and emergent design installment explores the code/design relationship, the importance using an expressive language, and the potential value of rethinking abstraction styles.

View more content in this series

As you know from the previous installments in this series, my contention is that every piece of software includes reusable chunks of code. For example, the way your company handles security is probably consistent throughout an application and across multiple applications. This is an example of what I refer to as an idiomatic pattern. These patterns represent common solutions to problems that you've encountered while building a particular piece of software. Idiomatic patterns exist in two styles:

1. Technical patterns - These encompass transactions, security, and other infrastructure elements.
2. Domain patterns - These include common solutions to business problems that span a single application or multiple applications.

In previous installments, I've focused most of my attention on how you can discover these patterns. However, once you discover them, you must be able to leverage them as reusable code. In this article, I'll investigate the relationship between design and code, and in particular how expressive code makes it easier to harvest patterns. And you'll see that you can sometimes solve seemingly intractable design problems — and simplify your code — by changing abstraction styles.
Way back in 1992, Jack Reeves wrote a perceptive essay entitled, "What is Software Design?" (see Related topics for an online copy). In it, he compares traditional engineering (such as hardware engineering and structural engineering) to software "engineering," with the goal of removing the quotation marks for software developers. The essay reaches some interesting conclusions.

Reeves' first observation is that the final deliverable for an engineering effort is "some type of documentation" (my italics). A structural engineer who designs a bridge doesn't deliver an actual bridge. The completed work is the design for a bridge. That design goes to a manufacturing team for construction. What is the analogous design document for software? Is it the napkin doodles, white board scribbles, UML diagrams, sequence diagrams, and other similar artifacts? These are all part of the design, but that collection isn't sufficient to hand over to a manufacturing team to make something real. In software, the manufacturing team is the compiler and deployment mechanism, which means that the complete design is the source code — the complete source code. Other artifacts aid in creating the code, but the final design deliverable is the code itself, suggesting that design in software cannot be abstracted away from code.

The next point Reeves makes is about the cost of manufacturing, which generally isn't considered part of the engineering effort but is part of the overall cost estimate for a engineered artifact. Building physical things is expensive, typically the most expensive part of the overall production process. In contrast, as Reeves says:

"...software is cheap to build. It does not qualify as inexpensive; it is so cheap it is almost free."

And remember, he was enduring C++ compile and link cycles, which are huge time sinks. Now, in the Java™ world, a team of elves springs to life and manufactures your design every time you stop typing! Software building is now so free that it is virtually invisible. We have a huge advantage over traditional engineers, who would probably love to be able to construct their designs freely and play what-if games. Can you imagine how incredibly elaborate bridges would be if bridge engineers could play with their designs in real time, for free?

Ease of manufacturing explains why we don't have much mathematical rigor in software development. Traditional engineers developed mathematical models and other sophisticated techniques for predictability so that they weren't forced to build things to determine their characteristics. Software developers don't need that level of analysis. It's easier to build our designs and test them than to build formal proofs of how they will behave. Testing is the
engineering rigor of software development. Which leads to the most interesting conclusion from Reeves' essay:

Given that software designs are relatively easy to turn out, and essentially free to build, an unsurprising revelation is that software designs tend to be incredibly large and complex.

In fact, I think that software design is one of the more complex things humans have ever tried, especially given the constantly escalating sophistication in what we're building. Considering that software development has been mainstream for only about 50 years, it is astounding how much complexity we've managed to build up in typical enterprise software.

Another conclusion from Reeves' essay is that design in software (that is, writing the entire source code) is by far the most expensive activity. That means that time wasted when designing is a waste of the most expensive resource. Which brings me back around to emergent design. If you spend a great deal of time trying to anticipate all the things you'll need before you've started writing code, you will always waste some time because you don't yet know what you don't know. In other words, you always run into unexpected time sinks when writing software because some requirements are more complex than you thought, or you didn't fully understand the problem at the beginning. The longer you can defer decisions, the greater your ability to make better decisions — because the context and knowledge you acquire increase with time, as shown in Figure 1:

**Figure 1. The longer you can defer decisions, the more contextualized they can be**

The lean movement has a great phrase: the last responsible moment — not the last moment, but the last responsible moment for decisions. The longer you can wait, the better chance you have for more suitable design.

**Expressiveness**

Yet another conclusion from Reeves' essay revolves around the importance of readable design, which translates to more readable code. Finding idiomatic patterns in code is hard enough, but if your language adds extra cruft, it becomes even harder. Finding an idiomatic pattern in an assembly language code base, for example, is very difficult because the language imposes so many opaque elements that you must be able to see around to "see" the design.

Because design is code, you should choose the most expressive language you can. Leveraging the language's expressiveness makes it easier to see idiomatic patterns emerge because the medium of design is clearer.
Here is an example. In an earlier installment of this series ("Composed method and SLAP"), I went through a refactoring exercise on some existing code, applying composed method and single level of abstraction (SLAP) principle. The top-level method I derived appears in Listing 1:

**Listing 1. Improved abstraction for the `addOrder()` method**

```java
public void addOrderFrom(ShoppingCart cart, String userName, Order order) throws SQLException {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (SQLException sqlx) {
        rollbackTransaction();
        throw sqlx;
    } finally {
        cleanUp();
    }
}
```

This looks like a good candidate for harvest as an idiomatic pattern. The first pass, shown in Listing 2, does this in the "native" language — the Java language:

**Listing 2. Refactoring an idiomatic "unit of work" pattern**

```java
public void wrapInTransaction(Command c) {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (RuntimeException ex) {
        rollbackTransaction();
        throw ex;
    } finally {
        cleanUp();
    }
}

public void addOrderFrom(final ShoppingCart cart, final String userName, final Order order) throws SQLException {
    wrapInTransaction(new Command() {
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

**Frameworks as pattern collections**

If you're familiar with Hibernate, you'll notice that the `wrapInTransaction()` method mimics Hibernate's `doInTransaction` helper. Most successful frameworks wrap a contextualized set of technical idiomatic patterns. The usefulness of a framework's patterns correlates pretty closely with how that framework came into existence. If the framework was extracted from working code, the patterns are more focused on real-world problems. Good frameworks (Hibernate, Spring, and Ruby on Rails; see Related topics) mostly come from the crucible of real-world use.
If, on the other hand, a framework was created in an ivory tower, many of the patterns sound cool but aren't that useful in real projects. My favorite example of speculative development in frameworks is the custom rendering pipeline "feature" of JavaServer Faces (JSF). It allows you to output any of a variety of output formats (for example, HTML, XHTML, and WML). I've never yet met a developer who needed this feature (although I'm sure there are some), but you pay a bit of cost for it in every JSF application you write. (It adds complexity to understanding the event model and pipeline.)

In this version, I have abstracted the boilerplate code into the `wrapInTransaction()` method, using the Gang of Four's Command design pattern (see Related topics). The `addOrderFrom()` method is now much more readable — the essence of the method (the two innermost lines) is more obvious. However, to get to that level of abstraction, the Java language forces a lot of technical cruft. You must understand how anonymous inner classes work (the inline declaration of the `Command` subclass) and understand the implications of the `execute()` method. For example, only final object references from the outer class are invokable within the body of the anonymous inner class.

What if I write this same code in a more expressive modern Java dialect? Listing 3 shows the same method, rewritten using Groovy:

**Listing 3. The addOrderFrom() method rewritten in Groovy**

```groovy
class OrderDbClosure {
    def wrapInTransaction(command) {
        setupDataInfrastructure()
        try {
            command()
            completeTransaction()
        } catch (RuntimeException ex) {
            rollbackTransaction()
            throw ex
        }
    } finally {
        cleanUp()
    }

    def addOrderFrom(cart, userName, order) {
        wrapInTransaction {
            add order, userKeyBasedOn(userName)
            addLineItemsFrom cart, order.getOrderKey()
        }
    }
}
```

This code (especially the `addOrderFrom()` method) is much more readable. The Groovy language includes the Command design pattern; any code in Groovy delimited with curly braces — `{ }` — is automatically a code block, executable via the syntactic sugar of putting open and close parentheses after the variable that holds the code-block reference. This built-in pattern allows the body of the `addOrderFrom()` method to be more expressive (by virtue of less obtuse code). Groovy also allows you to eliminate some parentheses around parameters, leading to fewer noise characters.

Listing 4 shows a similar translation, this time in Ruby (via JRuby):
Listing 4. The addOrderFrom() method translated to Ruby

```ruby
def wrap_in_transaction
  setup_data_infrastructure
  begin
    yield
  complete_transaction
  rescue
    rollback_transaction
  ensure
    cleanup
  end
end

def add_order_from
  wrap_in_transaction do
    add order, user_key_based_on(user_name)
    add_line_items_from cart, order.order_key
  end
end
```

This code is more similar to the Groovy code than the Java version is. The main difference between the Groovy code and the Ruby code is in the Command pattern characteristics. In Ruby, any method can take a code block, which is executed via the `yield` call within the method body. Thus, in Ruby, you don’t even need to specify a special type of infrastructure element — the capabilities exist within the language to handle this common usage.

**Abstraction styles**

Different languages handle abstractions in different ways. Everyone reading this article is familiar with a few pervasive abstraction styles — such as structured, modular, and object-orientation — which appear in numerous languages. When you work in a particular language for a long time, it becomes the golden hammer: every problem looks like a nail that can be driven by the abstractions in your language. This is particularly true in more or less purely object-oriented languages (such as the Java language) because the primary abstractions are hierarchy and mutable state.

The Java world is showing a lot of interest now in functional languages such as Scala and Clojure. When you code in a functional language, you think about solutions to problems differently. For example, the default in most functional languages creates immutable variables rather than mutable ones, which is exactly opposite of the Java approach. In Java code, data structures are mutable by default, and you must add more code to make them act immutably. This means that it is much easier to write multi-threading applications in functional languages because immutable data structures inherently interact cleanly with threads.

Abstractions aren’t purely the realm of language designers. An interesting paper presented at OOPSLA in 2006, titled “Collaborative Diffusion: Programming Antiobjects” (see Related topics), introduced the concept of an antiobject, which is an object that does the opposite of what we think it should do. This approach addresses a problem elucidated in the paper: *The metaphor of objects can go too far by making us try to create objects that are too much inspired by the real world.*
The point of the paper is that it is too easy to get caught up in a particular abstraction style, making the problem harder than it should be. By coding your solution as an antiobject, you can solve a simpler problem by changing your point of view.

The example cited in the paper illustrates this concept beautifully — the original Pac-Man video console game from the early 1980s (shown in Figure 2):

**Figure 2. The original Pac-Man video game**

The original Pac-Man game had less processor power and memory than some current-day wristwatches. The game designers had a serious problem given their limited resources: how do you calculate the distance between two moving objects in a maze? They didn't have nearly enough processor power for that, so they took an antiobject approach by building all the game intelligence into the maze itself.

The maze in Pac-Man is a state machine, where each cell runs rules for each iteration of the board. The designers invented the concept of Pac-Man smell. Whatever cell the Pac-Man character occupied had maximum Pac-Man smell, and the most recently vacated cell had maximum Pac-Man smell minus 1, and the smell decayed rapidly. The ghosts (who pursue Pac-Man and can move slightly faster) wander pseudo randomly until they encounter Pac-Man smell, at which time they go to the cell where it is stronger. Add some randomness to the ghosts movements, and you have Pac-Man. One side effect of this design is the inability for the ghosts to cut Pac-Man off: they can't see him coming, they can only tell where he's been.

This simple rethinking of the problem made the underlying code much simpler. By changing their abstraction to the background, the Pac-Man designers achieved their goal in a highly constrained environment. When confronted with a particularly nasty problem (especially when refactoring away from overly complex code), ask yourself if there is an antiobject approach that might make more sense.
Conclusion

In this installment, I've been looking at why expressiveness matters and manifestations of expressiveness in code. I agree with Jack Reeves' engineering comparison; I think that the complete source code is the design artifact in software. Once you understand that, it explains a lot about past failures (such as model-driven architecture, which tries to go directly from UML artifacts to code and fails because the diagramming language isn't expressive enough to capture the required nuances). This understanding has several side effects, including the realization that design (which is coding) is the most expensive activity you can perform. This doesn't mean that you shouldn't use preliminary tools (such as UML or something similar) to help you understand the design before you start coding, but the code becomes the real design once you move to that phase.

Readable design matters. The more expressive your design, the easier it is to modify it and eventually harvest idiomatic patterns from it via emergent design. In the next installment, I'll continue this line of thought and provide concrete ways to leverage design elements that you harvest from code.
Related topics

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book expands on a number of the topics in this series.
- *Design Patterns* (Erich Gamma et al., Addison-Wesley, 1995): The classic work on design patterns, including the Command pattern.
- "*Groovy: A DSL for Java programmers*" (Scott Davis, developerWorks, February 2009): Start reading the *Practically Groovy* series to learn how Groove's advanced syntax lets you write more readable (and less) code.
- "*Ruby off the rails*" (Andrew Glover, developerWorks, December 2005): Get to know Ruby from a Java developer's perspective.
- Hibernate: This popular open source object-relational mapping framework encapsulates many handy idiomatic patterns.
- Spring: The Spring framework is considered one of the most useful frameworks in all of Javadom.
- Ruby on Rails: Rails a sophisticated framework for creating Web applications in the Ruby (which includes JRuby) language.
- "*Collaborative Diffusion: Programming Antiobjects*" (Alexander Repenning, OOPSLA 2006): This paper describes the antiobject abstraction approach.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)