Evolutionary architecture and emergent design: Composed method and SLAP

Skill Level: Intermediate

Neal Ford (nford@thoughtworks.com)
Software Architect / Meme Wrangler
ThoughtWorks

21 Apr 2009

How do you find hidden design in aging code bases? This article discusses two important patterns for code structure: *composed method* and *single level of abstraction*. Applying these principles to your code allows you to find reusable assets that remained hidden before, with the further benefit of letting you abstract existing code into harvested frameworks.

In the previous two installments of this series, I discussed how using test-driven development (TDD) helps you incrementally discover design. That works great if you start with a green-field project. But what about the more common case in which you have lots of code that isn't the best in the world? How do you find the reusable assets and designs hiding in an aging code base?

About this series

This series aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid grounding in the agile practices of *evolutionary architecture* and *emergent design*. By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects.

This article talks about two decades-old patterns that help you refactor your code to find reusable assets: *composed method* and the *single level of abstraction* (SLAP) principle. The elements of good design already appear in your code; you just need tools to help you expose the hidden assets you've already created.

Composed method

One of the unfortunate side effects of the pace of technological change is that as developers, we frequently ignore software *lore*. We tend to think that anything more than a few years old must be hopelessly outdated. That is, of course, untrue: lots of books form important lore for developers. One of those classics (now mostly ignored) is *Smalltalk Best Practice Patterns* by Kent Beck (see Resources). As a Java developer, you may ask yourself, "How could a 13-year-old book about Smalltalk have anything to do with me?" It turns out that the Smalltalkers were the first developers to program in an object-oriented language, and they developed lots of good ideas. One of them is *composed method*.

The composed method pattern defines three key statements:

- Divide your programs into methods that perform one identifiable task.
- Keep all the operations in a method at the same level of abstraction.
- This will naturally result in programs with many small methods, each a few lines long.

I discussed composed method in "Test-driven design, Part 1," in the context of writing unit tests before you write actual code. Rigorous adherence to TDD naturally creates methods that adhere to composed method. But what about preexisting code? Now it's time to investigate using composed method to reveal hidden design.

Idiomatic patterns

You're probably familiar with the formal Design Patterns movement, popularized by the seminal Gang of Four book *Design Patterns* (see Resources). It describes generic patterns that apply across all projects. However, every solution contains *idiomatic patterns*, which are not formal enough to emblazon in a book but are pervasive nonetheless. Idiomatic patterns represent common design idioms in your code. The real trick in emergent design is discovering these patterns. They range from purely technical patterns (for example, the way transactions are handled in this project) to problem-domain patterns (such as "always check the customer's credit before proceeding with shipping").

Refactoring to composed method

Consider the simple method in Listing 1, designed to use low-level JDBC to connect to a database, gather up `Part` objects, and put them in a `List`:

Listing 1. Simple method for harvesting Parts

```
public void populate() throws Exception  {
    Connection c = null;
    try {
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL, USER, PASSWORD);
        Statement stmt = c.createStatement();
        ResultSet rs = stmt.executeQuery(SQL_SELECT_PARTS);
        while (rs.next()) {
            Part p = new Part();
            p.setName(rs.getString("name"));
            p.setBrand(rs.getString("brand"));
            p.setRetailPrice(rs.getDouble("retail_price"));
            partList.add(p);
        }
    } finally {
        c.close();
    }
}
```

Listing 1 doesn't include anything particularly complex. Nor does it obviously contain reusable code. It's also pretty short, but it should be refactored. Composed method says that each method should do one and only one thing, and this method violates that rule. I have a heuristic for Java projects that any method longer than about 10 lines of code asks for refactoring because it probably does more than one thing. So I'll refactor this method, with composed method in mind, to see if I can isolate the atomic parts. The refactored version appears in Listing 2:

Olio methods

Olio is defined as "a miscellaneous collection of things," and it's used colloquially as another word for "leftovers." (It shows up in crossword puzzles a lot.) *Olio methods* are giant methods with a large collection of miscellaneous stuff in them, jumping all over the problem domain. The methods in your code base that hit the 300-line mark are by definition olio methods. How can such a method possibly be cohesive if it's that large? Olio is one of major inhibitors for refactoring, testing, and emergent design.

Listing 2. Refactored populate() method

```
public void populate() throws Exception {
    Connection c = null;
    try {
        c = getDatabaseConnection();
        ResultSet rs = createResultSet(c);
        while (rs.next())
            addPartToListFromResultSet(rs);
    } finally {
        c.close();
    }
}

private ResultSet createResultSet(Connection c)
        throws SQLException {
    return c.createStatement().
            executeQuery(SQL_SELECT_PARTS);
}
```

```
private Connection getDatabaseConnection()
        throws ClassNotFoundException, SQLException {
    Connection c;
    Class.forName(DRIVER_CLASS);
    c = DriverManager.getConnection(DB_URL,
            "webuser", "webpass");
    return c;
}

private void addPartToListFromResultSet(ResultSet rs)
        throws SQLException {
    Part p = new Part();
    p.setName(rs.getString("name"));
    p.setBrand(rs.getString("brand"));
    p.setRetailPrice(rs.getDouble("retail_price"));
    partList.add(p);
}
```

The `populate()` method is much shorter now, and it reads like an outline of the tasks it needs to perform, with the task implementation residing in private methods. Once I've pulled out all the atomic parts, I can look at what assets I actually have. Notice that the `getDatabaseConnection()` method has nothing to do with *parts* — it is generic functionality to connect to a database. That suggests that this method shouldn't be in this class, so I'm going to refactor it upward to a `BoundaryBase` class, which acts as the parent of the `PartDb` class.

Are there any more methods in Listing 2 generic enough to be generalized in a parent class? The `createResultSet()` method sounds pretty generic, but it does have a link to *parts*, namely the `SQL_SELECT_PARTS` constant. If I can figure out a way to force the child class (`PartDb`) to tell the parent class the value of this SQL string, I can pull this method up as well. This is exactly the purpose of an abstract method. Thus I pull `createResultSet()` into the `BoundaryBase` class along with a companion abstract method named `getSqlForEntity()` method, as shown in Listing 3:

Listing 3. The BoundaryBase class so far

```
abstract public class BoundaryBase {
    private static final String DRIVER_CLASS =
            "com.mysql.jdbc.Driver";
    private static final String DB_URL =
            "jdbc:mysql://localhost/orderentry";

    protected Connection getDatabaseConnection() throws ClassNotFoundException,
            SQLException {
        Connection c;
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL, "webuser", "webpass");
        return c;
    }

    abstract protected String getSqlForEntity();

    protected ResultSet createResultSet(Connection c) throws SQLException {
        Statement stmt = c.createStatement();
        return stmt.executeQuery(getSqlForEntity());
    }
```

That was fun. Can I pull more methods from the child into the generic parent class? If you look at Listing 2's `populate()` method itself, its ties to the `PartDb` class are the `getDatabaseConnection()`, `createResultSet()`, and `addPartToListFromResultSet()` methods. The first two methods have already moved to the parent class. If I abstract the `addPartToListFromResultSet()` method (along with an appropriate more generic rename), I can pull the entire `populate()` method into the parent, which I have done in Listing 4:

Listing 4. The BoundaryBase class

```
abstract public class BoundaryBase {
    private static final String DRIVER_CLASS =
            "com.mysql.jdbc.Driver";
    private static final String DB_URL =
            "jdbc:mysql://localhost/orderentry";

    protected Connection getDatabaseConnection() throws ClassNotFoundException,
            SQLException {
        Connection c;
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL, "webuser", "webpass");
        return c;
    }

    abstract protected String getSqlForEntity();

    protected ResultSet createResultSet(Connection c) throws SQLException {
        Statement stmt = c.createStatement();
        return stmt.executeQuery(getSqlForEntity());
    }

    abstract protected void addEntityToListFromResultSet(ResultSet rs)
            throws SQLException;

    public void populate() throws Exception {
        Connection c = null;
        try {
            c = getDatabaseConnection();
            ResultSet rs = createResultSet(c);
            while (rs.next())
                addEntityToListFromResultSet(rs);
        } finally {
            c.close();
        }
    }

}
```

Once I've pulled all those methods up to the parent class, the `PartDb` class has been greatly simplified, as shown in Listing 5:

Listing 5. The simplified, refactored PartDb class

```
public class PartDb extends BoundaryBase {
    private static final int DEFAULT_INITIAL_LIST_SIZE = 40;
    private static final String SQL_SELECT_PARTS =
            "select name, brand, retail_price from parts";
    private static final Part[] TEMPLATE = new Part[0];
    private ArrayList partList;

    public PartDb() {
        partList = new ArrayList(DEFAULT_INITIAL_LIST_SIZE);
    }

    public Part[] getParts() {
        return (Part[]) partList.toArray(TEMPLATE);
    }

    protected String getSqlForEntity() {
        return SQL_SELECT_PARTS;
    }

    protected void addEntityToListFromResultSet(ResultSet rs)
            throws SQLException {
        Part p = new Part();
        p.setName(rs.getString("name"));
        p.setBrand(rs.getString("brand"));
        p.setRetailPrice(rs.getDouble("retail_price"));
        partList.add(p);
    }
}
```

What have I achieved by going through this refactoring exercise? First, I now have two classes that are much more focused on their particular jobs than before. All the methods in both classes are concise, which makes them easy to understand. Second, notice that the `PartDb` class pertains to *parts* and nothing else. All the generic, boilerplate connectivity code has moved into the parent class. Third, all these methods are testable now: each method (except `populate()`) does just one thing. The `populate()` method is the real workflow method of these classes. It uses all the other (private) methods to perform work, and it reads like an outline of the steps performed. Fourth, now that I have small building blocks, method reuse becomes easier because now I can mix and match them. The chances of using a big method like the original `populate()` method are small: it's highly unlikely that I'll need to do those exact things, in exactly that order, in a subsequent class. Having atomic methods allows you to mix and match functionality.

Extracted vs. preemptive frameworks

The best frameworks tend to be those extracted from working code rather than preemptively designed. Someone sitting down to design a framework must anticipate all the ways developers might want to use it. The framework ends up including lots of features, with a high chance that any given user won't use all of them. But you still must take the unused features of your chosen framework into account, because they add to your applications' accidental complexity; this could mean doing something as simple as making extra entries in a configuration document, or as intrusive as changing the way you'd like to implement a feature. Preemptive frameworks tend to be giant buckets of features, with other (unanticipated) features left out. JavaServer Faces (JSF) is a classic example of a preemptive framework. One of its cool features is the ability to plug in different rendering pipelines, in case you want to emit a format other than HTML. Although this feature is used rarely, all JSF users must understand its impact on a JSF request's life cycle.

Frameworks that grow from working applications tend to offer a more pragmatic set of features, because they address the actual problems someone faced when writing an application. Extracted frameworks tend to have fewer extraneous features. Contrast a preemptive framework like JSF with an extracted framework like Ruby on Rails, which has grown from actual usage.

The really important benefit of this exercise is the ability to harvest reusable code. When you look at the code in Listing 1, you don't see reusable assets; you just see a pile of code. By pulling the olio method apart, I discover reusable assets. But the advantages go beyond reuse. I've also created the foundation for a simple framework to handle persistence in my application. When it comes time to create another simple boundary class to harvest some entity from a database, I already have code to help me do that. This is the essence of *extracting* frameworks rather than building them in an ivory tower.

The harvesting of reusable assets allows your application's overall design to start shining through the pile of code that makes up the application. One of the goals of emergent design is to find idiomatic patterns of use within your application. The combination of `BoundaryBase` and `PartDb` forms a usable pattern that appears over and over in this application. When you have all the moving parts in small pieces, it is easier to see how they fit together.

SLAP

The second part of the definition of composed method states that you should "keep all the operations in a method at the same level of abstraction." An example of applying this principle will help you understand what it means and what kind of impact it can have on design.

Consider the code in Listing 6, taken from a small e-commerce application. The `addOrder()` method takes in several parameters and places order information into the database.

Listing 6. The addOrder() method from an e-commerce site

```
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    Connection c = null;
    PreparedStatement ps = null;
    Statement s = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        s = c.createStatement();
        transactionState = c.getAutoCommit();
        int userKey = getUserKey(userName, c, ps, rs);
        c.setAutoCommit(false);
        addSingleOrder(order, c, ps, userKey);
        int orderKey = getOrderKey(s, rs);
        addLineItems(cart, c, orderKey);
        c.commit();
        order.setOrderKeyFrom(orderKey);
    } catch (SQLException sqlx) {
        s = c.createStatement();
        c.rollback();
        throw sqlx;
    } finally {
        try {
            c.setAutoCommit(transactionState);
            dbPool.release(c);
            if (s != null)
                s.close();
            if (ps != null)
                ps.close();
            if (rs != null)
                rs.close();
        } catch (SQLException ignored) {
        }
    }
}
```

The `addOrder()` method has lots of messy stuff in it. What I'm particularly interested in, though, is the workflow near the beginning of the `try` block. Notice these two consecutive lines:

```
c.setAutoCommit(false);
addSingleOrder(order, c, ps, userKey);
```

These two lines of code illustrate a violation of the SLAP principle. The first (and the methods above it) deals with the low-level details of setting up database infrastructure. The second is a higher-order method, one that a business analyst would understand. The two lines come from two different worlds. It is difficult to read the code when you have to jump mentally between abstraction levels, which is what the SLAP principle tries to avoid. A corollary to the readability problem lies in the difficulty understanding the underlying design of what the code does, making it harder to isolate the idiomatic patterns for this particular application.

To improve the code in Listing 6, I'm going to refactor it with SLAP in mind. After a couple of rounds of *extract method* refactorings, I'm left with the code in Listing 7:

Listing 7. Improved abstraction for the addOrder() method

```
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();
    }
}

private void setupDataInfrastructure() throws SQLException {
    _db = new HashMap();
    Connection c = dbPool.getConnection();
    _db.put("connection", c);
    _db.put("transaction state",
            Boolean.valueOf(setupTransactionStateFor(c)));
}

private void cleanUp() throws SQLException {
    Connection connection = (Connection) _db.get("connection");
    boolean transactionState = ((Boolean)
            _db.get("transation state")).booleanValue();
    Statement s = (Statement) _db.get("statement");
    PreparedStatement ps = (PreparedStatement)
            _db.get("prepared statement");
    ResultSet rs = (ResultSet) _db.get("result set");
    connection.setAutoCommit(transactionState);
    dbPool.release(connection);
    if (s != null)
        s.close();
    if (ps != null)
        ps.close();
    if (rs != null)
        rs.close();
}

private void rollbackTransaction()
        throws SQLException {
    ((Connection) _db.get("connection")).rollback();
}

private void completeTransaction()
        throws SQLException {
    ((Connection) _db.get("connection")).commit();
}

private boolean setupTransactionStateFor(Connection c)
        throws SQLException {
    boolean transactionState = c.getAutoCommit();
    c.setAutoCommit(false);
    return transactionState;
}
```

The method is now much more readable. Its main body adheres to the goal of composed method: it reads like an outline of the steps it performs. The methods are at such a high level now you could almost show them to a nontechnical person to describe what this method does. If you look closely at the `completeTransaction()` method, you'll notice that it is one line of code. Couldn't I put that one line of code back in the `addOrder()` method? Not without harming the code's readability and abstraction level. Jumping from high-order business workflow to the nitty-gritty details of transactions violates the SLAP principle. Having a `completeTransaction()` method abstracts my code toward the conceptual and away from concrete details. If in the future I change the way I do database access, I can change the contents of the `completeTransaction()` method without touching the calling code.

The SLAP principle aims to make your code easier to read and understand. But it also helps you discover the idiomatic patterns that exist in your code. Notice that one such pattern emerges in the way the updates are protected with transaction blocks. You could further refactor the `addOrder()` method toward the combination of methods shown in Listing 8:

Listing 8. Transactional access pattern

```
public void wrapInTransaction(Command c) throws SQLException {
    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() throws SQLException{
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

I've added a `wrapInTransaction()` method that implements this common pattern in my application, using an inline version of the Command Design Pattern from the Gang of Four (see Resources). The `wrapInTransaction()` method does all the plumbing required to ensure that my code works correctly. I'm left with some ugly boilerplate code because of the anonymous inner class wrapping this method's real purpose — the two lines of code that appear in the body of the `addOrderFrom()` method. This resource-protection block will appear over and over in your code, so it is a likely candidate for moving up into the hierarchy.

The reason I implemented the `wrapInTransaction()` code using an anonymous inner class highlights an important point about expressiveness of language syntax. If you write this code in Groovy, you can use the native closure blocks to create a prettier version of the same thing, as shown in Listing 9:

Listing 9. Wrapping transactional access using Groovy closures

```
public 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()
      }
    }
 }
```

Groovy's advanced language syntax and features (see Resources) make for more readable code, especially when combined with the complementary techniques of composed method and SLAP.

Conclusion

In this installment, I've looked at two important patterns of code design and readability. The first step in attacking poor design for existing code is to mold it into something you can work with. A 300-line method is useless from a design or reuse standpoint because you can't focus on the important constituent parts. By refactoring it to atomic pieces, you can see what assets you have. Once you can see them clearly, you can harvest the reusable parts and apply idiomatic design principles.

In the next installment, I'll discuss refactoring toward design, building on the concepts of composed method and the SLAP principle. In it, I'll discuss how to discover design already lurking in your code base.

Resources

Learn

- *Smalltalk Best Practice Patterns* (Kent Beck, Prentice Hall, 1996): Learn more about the *composed method* pattern.
- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): A longer example of both composed method and SLAP appears in Part 2 of this book.
- *Design Patterns* (Erich Gamma et al., Addison-Wesley, 1995): The classic work on design patterns, including the Command pattern.
- Spring Framework: Spring is a good example of a harvested framework.
- "Groovy: A DSL for Java programmers" (Scott Davis, developerWorks, February 2009): Start reading the newly revived *Practically Groovy* series to learn how Groovy's advanced syntax lets you write more readable (and less) code.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford
Neal Ford is Software Architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He is also the designer and developer of applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.

Trademarks