Refactoring With UML

Dave Astels

Saorsa Development Inc.
P.O. Box 397
Wolfville, Nova Scotia
Canada, B0P 1X0
+1 902 542 0365
dave@saorsa.com

Abstract

Refactoring is a core eXtreme Programming practice. It involves, at the lowest level, moving pieces of code about. As such is generally done using a text editor or text based development environment, with or without explicit refactoring support. This works well for small granularity refactorings. In this paper I propose using a Unified Modeling Language (UML) tool as an aid in finding smells and performing some of the larger granularity refactorings. Working in UML is also advantageous when performing some of the smaller refactorings, but does not apply to refactorings involving blocks of code smaller than an individual class member. The purpose of this paper is to give a taste of what is possible when you use UML as a tool for refactoring, not to be an exhaustive catalogue.

Keywords

Refactoring, Unified Modeling Language, UML, visual

INTRODUCTION

In the Smalltalk arena the Refactoring Browser has been available for some time, and now we are seeing refactoring support becoming available in Java development tools. These editor/browser based tools work well for small granularity refactoring (e.g. Extract Method, Rename Member) but aren't as intuitive or as easy to use for larger granularity refactorings such as Extract Hierarchy. The approach I propose in this paper is to use a Unified Modeling Language (UML) tool to perform these large refactorings, as well as aid in detecting code smells.

You need a UML tool that bases it's class diagram directly on code, and allows you to manipulate the code by directly manipulating the diagram. The tool I use is Together. Licensed versions of recent releases of Together have other features that aid in refactoring and smell detection, however, most of what I discuss in this paper can be performed using older releases, including the free whiteboard editions.

I propose three main reasons why refactoring in UML works and is worth exploring:

1. many people are visually oriented and like to be able to visualize the classes and their relationships;
2. being able to directly manipulate code at a higher level of granularity (i.e. methods, variables, and classes rather than characters) can make refactoring more efficient. This increase in efficiency is due to being able to quickly grab & move something directly that would entail selecting a run of text, and possibly multiple runs (consider performing Move Field when you want to include a possible accessor and/or mutator); and
3. being able to visualize code, specifically the content of classes and the relationships between them, can help in detecting smells.

REFACTORING

Refactoring is finally getting the attention it deserves. As defined by Fowler, and others, it provides a common language of patterns for manipulating source in such a way that the behavior is preserved.

Refactoring provides four major benefits. Specifically, it:

- improves the design of software,
- makes the code easier to understand,
- helps you find trouble, and
- speeds you up.

UNIFIED MODELING LANGUAGE

UML provides many different types of diagrams two of which prove especially useful for refactoring: Class, and Sequence. Class diagrams give a static view of the system (what classes make up the system, their contents, and their relationships), while sequence diagrams give a dynamic view of a specific sequence of.

SMELL DETECTION

Before we can refactor we need to know what to refactor and how we should proceed (i.e. which refactoring is indicated). This is done by detecting smells in the code. In [1], Beck and Fowler describe a code smell as *"certain structure in code that suggest the possibility of refactoring"*. In many cases, visualizing code using UML diagrams makes these structures more evident.

The following sections provide several examples of using UML to detect some common smells. This selection of examples is by no means complete, and should be con-

sidered a example of what can be learned from examining visualized code.

Data Class

To recognize a data class on a class diagram look for classes that contain significantly more data than behaviour. One thing to be careful of is accessor and mutator methods (i.e. getters and setters). They need to be disregarded when evaluating the size of the operations section of a class, as they are just to provide external access to the data, and are not behaviour. Classes that have getters and setters for most or all of their data members should jump out as data classes.

Together has a nice feature to help with this. By turning on JavaBean recognition, you can have Together group appropriately named attributes, accessors and mutators as a single property. Be careful, however, because a property is some combination of appropriately named attribute, getter, and setter. It doesn't necessarily imply an attribute. Figure 1 shows the difference in the appearance of a specific class (taken from an actual project).

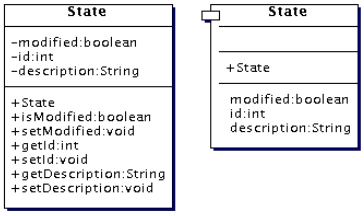

Figure 1. Class with properties showing bean recognition

Another indication of a data class is the presence of public attributes. Public attributes are a code stink. Classes with public attributes are often degenerate data classes, typically containing little, if any, behaviour. An example is shown in Figure 2.

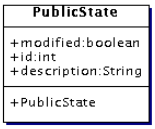

Figure 2. A public data class

Large Class

Finding large classes is very easy when looking at a class diagram with all members shown (you may want to hide accessors and mutators if possible). Details are not important, in fact getting an overview in which you can see the relative sizes of the classes is most illuminating. Consider Figure 3 (taken from an actual project).

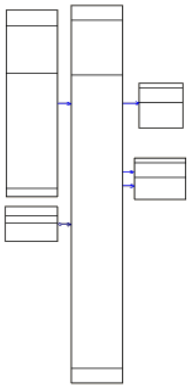

Figure 3. Large Class

You can immediately see that the central class is large relative to the others. This is not conclusive proof that refactoring is required, but it gives you an indication that there may be a problem. It may be that the surrounding classes are inordinately small, which is also a smell. In any case, the significant difference in the relative sizes of the classes is the important thing.

Another smell indicated by Figure 3 is that the large class may be acting as a controller for the surrounding classes when it should be delegating more to them.

Lazy Class

In many ways this is the opposite of a Large Class. Lazy classes are small, having few methods and little behaviour. They stand out in a class diagram because they are so small.

Middle Man

A Middle Man is a class that sits between two others and most just forwards method calls. Middle Men can be found by looking at a sequence diagram that involves them. See Figure 4 for a simplified example. A Middle Man is apparent by the pattern of messages simply being delegated to another class.

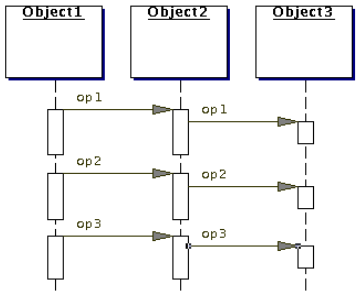

Figure 4. The sequence pattern for Middle Man

REFACTORINGS

In this section I will explore how several refactorings can be made easier by performing them in UML. The refactorings that this applies most to are those that involve multiple classes. One reason working in a class diagram can make these easier to perform is that you have all the classes in front of you at the same time.

Move Method
Here is a case where the direct manipulation of a class diagram really speeds things up. You simply grab the method in the diagram and drag-n-drop it onto the class where it should be.

Move Field
This works the same as moving a method. If your UML tool supports the ability to group attributes and the associates accessors/mutators, you get the added ability to drag along the supporting code as well.

Make Inner Class Freestanding
This is a refactoring I've thrown in because it so nicely shows the power of the technique. In Together, inner classes are shown in a section of their own in the class box. See Figure 5 for an example. The Java corresponding to the diagram is;

```
public class Outer {
   public class Inner {
   }
}
```

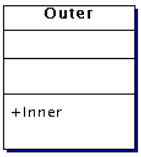

Figure 5. Example of an inner class

When the inner class would benefit from being freestanding it can be dragged out and dropped on the background of the diagram. All references are updated as required.

Replace Inheritance With Delegation
A good UML tool will allow you to manipulate the links between classes as well as the classes themselves. You perform this refactoring by removing the inheritance link, adding an association, and adding a delegating method.

Replace Delegation With Inheritance
This is the opposite of the last refactoring. You need to remove the delegating method and the association. Finally add the inheritance link.

REFACTORING TO PATTERNS

This section is specific to the use of Together, as far as I know. Together supports the automation of design patterns in two modes of operation:

1. building a pattern instance from scratch, creating the required classes, and
2. applying a pattern to existing classes.

As Gamma points out [5] and Kerievsky expands upon [6], patterns should be a target for refactoring, i.e. something that the design evolves into, not something that is chosen up front. This is especially important if you are doing XP. For this reason I won't explore mode 1, as it is a BDUF (Big Design Up Front) practice.

As an example of what is possible with this capability, I will show how a small piece of a design (I'll use the composite example from [5]). could be refactored into a composite. For reasons of simplicity and space, class members not related to the pattern are left out. Figure 6 shows the before state, a simple graphics structure. Your task involves in making Pictures nestable. The composite pattern is a natural for this. Figure 7 shows the section of the pattern dialog that allows you to configure the Composite pattern, selecting which classes fill which roles in the pattern. Figure 8 shows the result. This facility must be used with care, in keeping with YAGNI (i.e. "You Aren't Going to Need It").

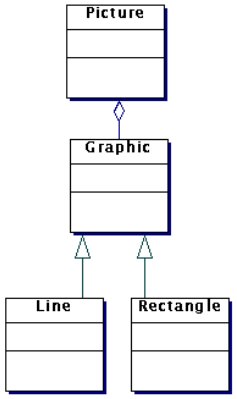

Figure 6. Before applying the Composite pattern

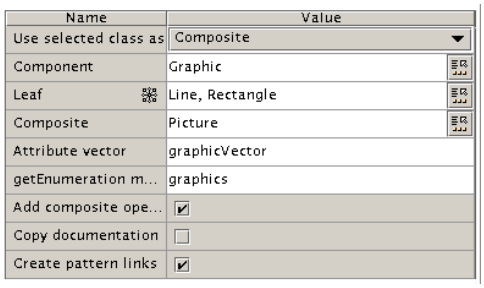

Figure 7. Composite pattern settings dialog

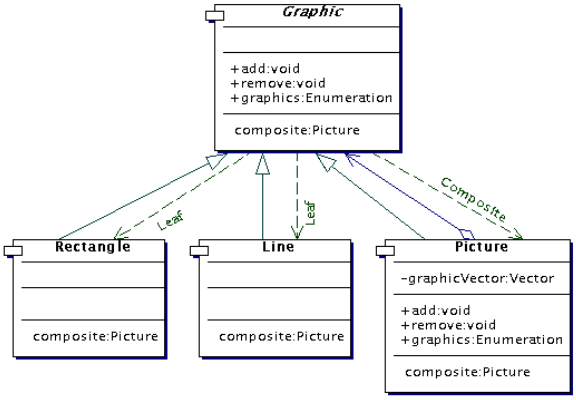

Figure 8. After applying the composite pattern

INFORMATION AND QUESTIONS

I hope this article makes you consider the value of using UML diagrams when performing smell detection and refactoring. As developers we are used to working with

textual source code. And as it says in [7], plain text is best. Sometimes, though, it can be enlightening to look at things in a different way. Viewing your code as a class or sequence diagram can cause things to jump out at you that you may not otherwise see (or not see as easily). In order to be able to do this in an effective and low cost manner, you need a UML tool that can generate diagrams from code, and allow you to control the level of detail and granularity so that you are not swamped with information. To enable you to perform refactorings directly on a UML diagram the tool needs to keep the code and model in sync, in realtime. These features are becoming more common in the available UML tools.

Using UML in the way presented here embodies the Agile Modeling idea of modeling to understand [AM]. The more ways you use to look at your code, the more information you can get out of it. The more information you have, the more courageous you can be, and the faster you can go.

Just as it makes sense to have code standards to make your code easier to understand and extract information from, it makes sense to have some standards when you are making UML diagrams. A well laid out diagram will communicate more easily to others as well as better enable the detection of graphical patterns that indicate code smells. Evitts [2] presents several useful patterns that you can apply when making diagrams (e.g. "Inheritance Goes Up"). Scott Ambler has also set up a site for "Online Tips and Techniques for creating better UML diagrams" [8].

As mentioned in the introduction, recent releases of Together have other features that enhance smell detection and refactoring. The latest version adds some explicit refactoring capabilities.

For some time Together has included an auditing capability which is valuable for smell detection. Many standard audit functions are included such as checking for public fields and coupling complexity. These can be used to guide you to areas in the code that may benefit from refactoring. Alas, auditing is only available in licensed versions.

On the Saorsa website (www.saorsa.com) I have put an HTML version of this paper which includes screen-capture animations showing the various refactorings being performed. It is hard to show in a paper how easily some of these refactorings are performed, so I invite you to visit our site and view the animations. I consider it a living document, and will be extending it with other refactorings.

ACKNOWLEDGEMENTS

There are a few people who I must acknowledge:

Michele Marchesi - for encouraging me to submit a paper;

Miroslav Novak - for inspiring the topic;

Scott Ambler - who taught me that modeling could be agile and used in the context of XP; and

Peter Coad, Dietrich Carrissius, and their team - for envisioning and creating the Together line of development tools, which makes possible many new and different ways to work with code.

REFERENCES

1. Agile Modeling website, www.agilemodeling.com

2. P. Evitts. *A UML Pattern Language*. MacMillian Technical Publishing, 2000

3. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999

4. M. Fowler with K. Scott. *UML Distilled - 2nd ed.* Addison-Wesley, 2000

5. E. Gamma, R. Helm, R. Johnson, J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

6. J. Kerievsky. *Patterns and XP*. in *Extreme Programming Examined*, G. Succi, M. Marchesi ed., Addison-Wesley, 2001

7. D. Thomas, A. Hunt. *The Pragmatic Programmer*. Addison-Wesley, 2000

8. Modeling Style website, www.modelingstyle.info