Spring 5-1-2009

Analysis of software evolution over time

Travis Alexander Rupp-Greene
University of Colorado Boulder

Analysis of Software Evolution Over Time

A thesis submitted in partial satisfaction

of the requirements for the degree

Bachelors of Science in Computer Science

at the University of Colorado at Boulder

by

Travis Rupp-Greene

Advisor

Amer Diwan

Committee Members

Kenneth Anderson

Jeremy Siek

Michael Main

Christoph Reichenbach

1. Introduction

There have been many efforts to develop tools to assist software development over the years. Things such as refactorings are held in high regard as tools that make software engineering easier and more efficient. Research has been done to evaluate these tools in software engineering situations, but there has been very little research in to how software evolves over time. There have been efforts to develop automated tools to detect changes in software, but these are based upon a presupposed notion of how software is changing, such as the tool developed and tested in the paper “Automatic Inference of Structural Changes for Matching Across Program Versions” (Kim, Notkin, Grossman, 2007) which used a rule-based system to match changes in program versions. This can check for the presence of something specific in program changes, but this does not help with an overall understanding of how software evolves.

This paper aims to qualitatively analyze the changes in software over time to gain an understanding of how software evolves. My primary goal is to discover how software changes, and to create a language to describe these changes. My secondary goal is to see what existing software engineering tools (like refactorings) are being used in open source software and how often they occur. Knowledge of how software evolves over time allows for more effective tools to be built. For this task I chose two projects with different aims, HtmlUnit and JEdit, and cataloged the changes that took place between revisions.

2. Methodology

I needed projects that kept a history of all code changes, for this reason I picked projects that were managed by subversion code repositories. Subversion is a tool which provides a centralized repository for software projects to store all of their code. Users check out the files that they will use to a local copy on their computer, and then when they want to update the copy in the repository they commit their changes with a comment. Subversion gives each commit a numeric identifier that is incremented with each commit, and this number is consistent across everything in the repository, so revision 10 will always be after revision 9 no matter where in the repository it was committed.

I used the "diff" tool built in to the subversion client which displays textual differences between specified revisions of the project broken down on a per-file basis. I started evaluating individual revisions but after some investigation I moved to evaluating every other revision because it was faster to evaluate. I recorded the file that changes occurred in and a high level description of what changed. These descriptions were concise descriptions that were based on my computer science knowledge, such as "added import <package>" or "extracted code to method <method name>". I used these revisions to look for overall patterns and structured changes within the projects.

For each project I manually categorized 150 revisions in two blocks of 75 from points either at the beginning of the project or after major versions. The goal of this was that I would capture interesting points in the software's evolution by comparing how the software was developed under similar circumstances but after a long intervening period of development, and hopefully find overarching patterns and steps which characterized software evolution.

3. Systems and Findings

My criteria for projects to evaluate were that they were written in Java, had a subversion repository, were open source, were widely used, and had been around for long enough to have multiple major version releases. I chose these criteria because they represent successful software projects that use development methodology that will sustain the project. To find these projects I drew from what I had used personally and from asking peers about what open source projects they use. The two projects I decided on using were HtmlUnit and jEdit. At the time of writing, HtmlUnit has over four thousand revisions, has had two major releases, and has 8 developers associated with the project. jEdit has over fourteen thousand revisions, has had four major releases, and has 158 developers associated with the project.

3.1. HtmlUnit

HtmlUnit is a headless open-source browser that has been developed to aid automated testing. It is written entirely in Java with some XML configuration. HtmlUnit provides a full Javascript engine and the ability to simulate multiple windows being opened. This makes it an ideal tool for Internet application developers who want to test using existing automated test tools, such as JUnit or TestNG. HtmlUnit leverages existing open source projects to provide functionality without reinventing the wheel such as Mozilla's "Rhino" Javascript engine and the "CyberNeko" HTML parser. HtmlUnit is hosted on sourceforge, which is a website that provides services to open-source development teams such as a subversion repository and developer mailing lists. I cataloged the first 75 revisions from 1 and then 75 revisions after the 2.0 release.

3.1.1 HtmlUnit Start

A notable element of HtmlUnit's development from the start was the concurrent development of tests and features. These tests were unit tests in most cases and had class names that related to what they were testing. Also from the start it was evident that they used the ant build system given the inclusion of a build.xml file.

Starting at revision 7 HtmlUnit used a changes.xml and a todo.xml. The changes.xml was updated with an action type of "add", "remove", "fix", or "update", what developer committed the change, an optional issue number, and a description of the change. These were later grouped by release, and are updated automatically when a change is committed to subversion. The todo.xml was an HTML document and updated infrequently (four updates total) with things that needed to be done, but there were no further updates to it after version 44.

The next notable software evolution change was in revision 17-18. Here there were three local List variables which kept track of things relevant to the method, in this case a list of available properties, unavailable properties, and functions. These were set based on method calls off of a configuration instance variable to get the lists. In the method there were several conditions based on if something was in one of these lists. These lists were removed and functions were created on the configuration object that served the same purpose (determining if something exists on the lists). In addition to this the old

methods that were used to get the lists had their scope changed to private. This reduced the amount of code in the original class, making it easier to understand, and although two of the functions for determining if something is present were direct copies of what existed in the original class, the third used a different method of going about determining if something was present or not.

In revisions 20-22 and 26-28 a boolean variable was split into a set of states. The method that started as "boolean isValidFunctionName()" was changed to be "int getFunctionNameState()" where "VALID" and "INVALID" were possible return values for getFunctionNameState and an additional state was added called "NOT_FOUND". This change happened once more in 20-22 and then three times in 26-28. These changes showed a case where additional information was needed about a state of an object than a simple true or false could provide, and the developers decided to condense it in to one function rather than have a series of "isSomeProperty()." In the example here they would have had to add a method isFunctionNameNotFound to get behavior similar to the new way of doing it. The clients of these methods usually used these in an if statement, with an else clause for if it was not a valid function name. These were changed to have a local variable that was set based on getFunctionNameState, and then an if statement if that matched the state the old function tested for, while the else was changed to an else if for whichever one of the new states it cared about. The new state checked for in the else if was a fairly even split between two other types.

In revisions 34-36 a supertype of several classes added a method that the subclasses already had. In this case it was the HtmlInput class adding a "click" method that HtmlSubmitInput and HtmlImageInput already had on them. The bodies of the child classes "click" methods were extracted to the super type's "click" method, and the child classes then called the supertype's click method. This condensed the repeated code so that general changes would affect any of the implementing classes, but it also still allowed for logic specific to the class to be easily added in. This is basically an Extract Method/Pull Up Method Refactoring combination, but with the addition of pulling up all the methods from the subtypes. None of the other classes that implemented HtmlInput changed anything related to the "click" method until revisions 54-56 where two more classes added a "click" method which again delegated to the supertype's "click" method.

A similar change to this occurred in revision 42-44, but in a more top down fashion. An interface added a "reset" method, and everything that implemented that interface added an empty "reset" method that consisted of logging the fact that reset was not implemented for that class. This was added to five classes in 42-44. In the following revisions (44-46) there were many more reset methods added (twelve). Not all of these were blank, some of them had the functionality to reset the field, and two of the original added reset fields were fleshed out with functionality beyond debug logging.

The next notable change was in revisions 62-64 where a layer of functionality was added to an object provided through the Java libraries. In this case it was the URL object where they needed to use a different constructor based on the contents of the string that defined the URL. This was accomplished by making a new static method called "makeUrl" which returned a URL object and had logic inside for picking the proper constructor. This required that two other methods on the object were made static as well. One of these methods contained several URL constructor invocations, all of which were replaced

by "makeUrl", and the other was a support function that would perform a null check. Also in these revisions was the addition of the "TextUtils" class which added a method that extended functionality of the String class that was used throughout HtmlUnit. Because the String class is marked as final the developers could not create a String subclass to add this functionality to.

This block of revisions wrapped up with an audit of the javadoc comments and imports used across the project. This involved adding comments to methods or classes which did not have any, and making sure that the method names, parameters, and return values for method javadoc comments were all correct. The imports involved removing imports that were no longer used by the classes.

3.1.2 HtmlUnit Post-2.0

HtmlUnit Post-2.0 started with the integration of the Rhino javascript engine at revision 2843. The developers chose to include all of the source of the Rhino engine in a subdirectory of the trunk. From the comments and associated files this was done so that the HtmlUnit authors could modify the Rhino engine in order to suit their needs. This could present problems in the future if Rhino were to be updated with new features or bug fixes then the either the changes that HtmlUnit made or the changes in the new version of Rhino would have to be integrated manually. In the post-2.0 block only one change to Rhino was recorded. The process of fully integrating Rhino lasted until revision 2859 when changes to the Rhino branch stopped being made.

HtmlUnit displayed a change in their development and testing strategy sometime between the start and post-2.0, where test classes and normal classes used a scheme to determine if a feature was implemented or not. This was first observable in the post-2.0 block at revision 2861. It went about this in a slightly complicated way. Many tests extended a base test class that included a method called "notYetImplemented()". What notYetImplemented() does is check to see if a static thread local notYetImplementedFlag is set. If it is then it returns true. Otherwise it sets the flag to false and determines what test called notYetImplemented(), and re-runs the test assuming that it will fail. If it does not fail then notYetImplemented() makes it fail with a message that the method was expected to fail and did not. Otherwise the test passes. This allows for incomplete tests and features to exist in live and working code and when the test fails provides an easy feedback mechanism to the developers when a feature that formerly did not work works. With a bit of further investigation by these revisions calling notYetImplemented explicitly had been deprecated in favor of the @NotYetImplemented annotation, which followed the same logic but without requiring extra code in the test.

Changes in the testing technology were revealed again in revision 2869-2871 with the use of annotations. The annotations were introduced sometime in the period before post-2.0, but this was the first time any change relating to them appeared in the Post-2.0 jEdit. In this revision two annotations were removed from a method, @Browser and @Alerts. Looking in to it the @Browser annotation determines what simulated browser the code should be run against. This support for mimicking different browsers is a huge boon to developers using HtmlUnit because browser incompatibility issues have plagued web development from the beginning. The @Alerts annotation lists a set of Javascript pop up alerts that are expected to be received when the method is run. This made HtmlUnit's tests very

short in a lot of cases, consisting of a string that held the html to generate the specified test behavior and a statement to execute the page in the HtmlUnit interpreter.

In revisions 2901-2903 a number of functions were marked as deprecated through the @Deprecated annotation. All 26 of the methods that had the @Deprecated annotation added to them were already marked with a javadoc deprecated comment of the style " * @deprecated ..." which is notable because this adds metadata that the code can interact with rather than just being something that a developer knows. But, if there was anything built in to HtmlUnit to handle the @Deprecated annotation it was not changed within this block.

Immediately following that in revisions 2903-2905 an automatic code cleanup tool was run. This tool ran through what appeared to be a set of steps to eliminate extraneous code. It removed unnecessary casts where what was being cast was already of the correct type. This made code a lot friendlier to read, as HtmlUnit contained a lot of casts. It also removed else cases where all the paths in the if/else if/else construct would return from the function. It also removed unused classes, methods, exceptions thrown, and imports. This is notable because all of the changes were functionally the same as what they were replacing, this just made things look nicer on a fairly large scale (the diff for this was 4067 lines long).

Finally revisions 2913 through 2919 were the lead up and cutting of version 2.1. They changed version numbers in relevant files from 2.0.1-SNAPSHOT to 2.1. The date of the last stable build was updated (from April 7th 2008 to April 15th 2008). There was a cleanup change in the removal of an unused plugin from the pom.xml directory. Then the last revision was when the 2.1 release branch was tagged.

3.2. jEdit

jEdit is a text editor written entirely in Java and targeted at programmers. It differentiates itself from other text editors by having a flexible plugin system which allowed it to have a wide range of functionality with little development time. It attempts to be a jack-of-all-trades program editor and provides syntax highlighting and keyword recognition for many languages including but not limited to Java, C, C++, and FORTRAN. The project uses the Swing library to provide its graphic user interface and has the BeanShell library integrated to provide scripting functionality and a command line within the application. It also utilizes xml files to store descriptions of how to tokenize different languages for its syntax highlighting. jEdit also maintains multiple APIs from their Java code, the BeanShell API and their external java plugin API. jEdit started out hosted on Giant Java Tree, an open source CVS repository, for a few years before moving to sourceforge and their SVN repositories. Because of this the first code in the jEdit sourceforge subversion repository is late in the version 3 releases. Because of that I started my evaluation of jEdit from the period after their 4.0 release by examining seventy five revisions, and then again after their 4.2 release with another seventy five revisions.

3.2.1 jEdit Post-4.0

jEdit used a more primitive form of change tracking than HtmlUnit. All throughout jEdit Post-4.0 most revisions included a change to the "CHANGES.txt" and/or "TODO.txt". "CHANGES.txt" in most cases included a short summary of what change occurred, such as "Removed BufferUpdate.ENCODING_CHANGED, FOLD_HANDLER_CHANGED messages; replaced with generic BufferUpdate.PROPERTIES_CHANGED" which gave details as to what the purpose of the change was. Additionally the "TODO.txt" was used as a de-facto bug list which was added to when whomever committed the change discovered something new that needed to be fixed or added, and was removed from when those tasks were accomplished. Also, according to the copyright notice on the Java files this set of changes occurred during 2002.

Early in Post-4.0 jEdit (4148-4150) there was a notable change. The developers split what was the BufferPrinter class in to two different BufferPrinter classes (BufferPrinter1_3 and BufferPrinter1_4) for compatibility with different versions of java. This shows the project changing to take advantage of new features in newer versions of Java while still maintaining official support for older versions of Java. The actual determination of which class to use based on the Java version was done in one of the support files (action.xml) rather than specifically in Java code. There were no other changes in that area specific to Java versions; I am assuming that if there were they would also have been split in a similar fashion to maintain compatibility with Java 1.3.

Part of revisions 4160-4162 included what the comments said "This is a bit silly... but WheelMouse seems to be unmaintained so the best solution is to add a hack here." The code that that comment explained did a name check to see if the plugin it was trying to load was named "WheelMouse" and if so it then returned an error that the plugin was obsolete. This is strange because this is the API responding to something developed for it directly, which is not what is usually expected from an API. In addition to force-obsoleting the plugin, there are other changes that implement mouse wheel scrolling (which I am assuming is what that plugin did). This demonstrates the application integrating a feature that was provided by a plugin and making it standard, while making sure that the non-updated plugin doesn't get loaded and conflict.

Another change in the 4162-4164 that stood out to me was when regular expressions were integrated in to their tokenizing scheme. This reduced a lot of code that they had written specifically to handle lots of edge cases and doing the parsing itself, so overall their code was a lot cleaner and more readable afterwards, and depending on the regular expression implementation they used it may have also been more efficient.

Stylistically jEdit is very different from HtmlUnit; one notable difference is its use of inner classes. One of the more complicated inner class related changes was the migration of Buffer.TokenList in to the TokenHandler interface, and the TokenList implementation becoming the DefaultTokenHandler. This all took place within one set of revisions (4168-4170) and involved creating the new interface (TokenHandler), creating the implementation of that interface (DefaultTokenHandler), and finally creating an empty inner class of Buffer called "TokenList" which extended DefaultTokenHandler to preserve existing references to the class. Any of these references would have been external, because all the references inside jEdit core code were changed.

Following this in 4171 was a change that reduced how much information needed to be specified in one of the supporting xml files that described a language. The developers added in automatic whitespace determination which up until that point had to be specified manually within the xml files, leading to two lines that defined space and tab as whitespace. This was a very simple change that reduced the total amount of code, and also removed something that had to be in every support file. As a sidenote, this also removed the possibility of user-defined whitespace rules, so if for some reason a language definition included something other than space or tab as whitespace it now would be unable to define that.

In revision 4202-4204 one of the consequences of library integration that was discussed in section 3.1.2 was realized. In one of these revisions the developers updated the BeanShell library from version 1.2b5 to 1.2b6, which involved updates to a very large number of classes that BeanShell used (over 6000 lines in the diff). Immediately after that in revisions 4204-4206 there were changes in the main jEdit codebase that responded to changes in the BeanShell library. Notably one of the classes that was private became public so a lot of logic around interfacing with BeanShell could be eliminated or simplified. It seems like it would have been possible to make this change to the copy of the BeanShell that jEdit includes, and then integrate the new version on top of their changes and get the same functionality with an easier development process. Also in the 4204-4206 section a large number of the scripts themselves were changed.

3.2.2 jEdit Post-4.2

According to copyright notices in the top of Java files jEdit Post-4.2 started in late 2004 and went through early 2005. Two years later the project still dealt with the same areas of the codebase, and there were a lot of similar changes in this block. Post-4.2 immediately (5113) started with the somewhat sweeping change of removing support for Java 1.3. This consisted of removing the Java14 class which used objects in Java 1.4 that were not in Java 1.3 for both Java-provided libraries and Swing. Other classes within jEdit would obtain references to methods on the Java14 class through reflection if the Java version was at least 1.4. This was done in such a way that people compiling jEdit on Java 1.3 would not need to include the Java14 class in their compilation. The fallout of this change made the effected code a great deal simpler, because they could simply call the methods that Java14 called, rather than having to go through the process of determine if Java version was over 1.3, obtaining a handle to the method, determining if the handle was not null, and then finally calling the method. There were not any additions that used functionality specific to Java 1.5 that was not included in Java 1.4.

Shortly after that (5121) another inner class was migrated to its own public class, following a similar path to the TokenHandler inner class to public class from the Post-4.0. This one was different because it did not involve preserving a deprecated reference in the source class. This change involved three separate inner classes removed from the same class. There were several other such changes at revisions 5171, 5177, and 5173. One of these changes was similar to the Post-4.0 change where the old inner class was changed to extend the new outer class and marked deprecated.

Directly after that in revision 5123 a branch was tagged entitled "before-screen-line-refactoring". This tags a subversion revision by name rather than the built in numeric method. This happened three other times with the tags "before-selection-manager" (5127), "jedit-4-3-pre1" (5163), and "before-fast-scroll" (5181). This was a difference in development methodology from the Post-4.0 block, and there is also some internal inconsistency as the changes for the screen line refactoring changed a lot less than some other non-tagged changes.

Starting at roughly at this point and as a continuing theme throughout all of the post-4.2 set of revisions the TODO.txt and CHANGES.txt were updated more sporadically. A possible reason for this is that additional developers may not be accustomed to the development process and do not update these files. All the major changes, like the changes that the branches were tagged for, did contain updates to CHANGES.txt. Another notable development was that the TODO.txt was very consistently added to a lot more than removed from.

Starting in revision 5165 there was trend to take objects that were composed of static variables with static method calls and restructure them to be singleton objects. This consisted of changing the method signatures on all of the methods within the object to no longer be static along with making all the static variables instance variables. They also added in a new static variable to hold the singleton instance, and a method to get the singleton instance, which if the current instance was null it would create. Any calls to the old static methods were replaced with a call to the getInstance method and then calling the method on the returned value from that function. This happened multiple times in fairly quick succession.

Additionally in revision 5165 a number of methods marked as deprecated were removed. None of these methods were deprecated in the initial block of jEdit revisions, which leads me to believe that jEdit follows a cycle of deprecation and then a final removal later after plugins have been given fair warning that the methods would no longer be used. There were no BeanShell script changes associated with these removals, so presumably none of the jEdit packaged scripts used any of the deprecated functionality.

In revisions 5169-5171 there were a great many variables that were changed from type Vector to type List, using an ArrayList implementation. This went along with other changes to use new functionality provided by the List interface. There were several for loops that used numeric indexes to access the data in a Vector that were changed to be while loops that used an iterator to access the elements of the List. Other changes relevant to this were method changes, like "insertElementAt" to "add." Also in revision 5169-5171 were several changes from general package-level imports to specific class imports, but only within the org.gjt.sp.jedit package. Other package-level imports remained untouched.

4. Broader Findings

4.1. Steps of Evolution

Across both projects I was looking to categorize the changes that were made in each revision into set of steps that could be used to describe how the code changed. Steps of evolution bear some semblance to Refactorings, but Refactorings are entirely behavior preserving. Steps of Evolution seek to describe changes in functionality, where functionality is defined as the scope of things which the program is able to do. For example if a new conditional is added to deal with negative numbers, then the functionality of the program has increased. Conversely if that same conditional is later removed the functionality has decreased. There are some steps of evolution that do not change functionality, and Refactorings are a subset of these. These steps aim to be able to completely describe the differences between two versions of a java file, and are ordered from the least specific to most specific changes that occurred. I consider changes which occurred in at least one of the two projects. With just two case studies that does not exclude them from being in other projects. I also kept track of the number of times changes occurred, some changes however are abundant and became so numerous after a while that any recording beyond knowing that there were a lot of that change added no additional insight. I split this list in to three categories, ones that always increase functionality, ones that always reduce functionality, and ones that change functionality, with the possibility of increasing, decreasing, or leaving functionality unchanged. Additionally at the end of this section in Table 4.1.1 there is a list of the steps and their occurrences.

4.1.1. Functionality Increasing Steps of Evolution

Add Code - This step is very general and categorizes when code is added that wasn't there before. This is an abundant change.

Add Method - This is the step of adding a new method to a class. This added new functionality and usually was associated with added method calls or other changes across the project. This step is abundant.

Add Variable - This step adds a variable to a class or method that wasn't there before. This usually went along with added methods or other structural changes such as the Create Branch evolution step, and is abundant.

Uncomment Code - The inverse of Comment Code described in section 4.1.2. Abundant.

Rich Add Variable - This is like the Add Variable step but with the addition of adding get and set methods and initializing the variable in the constructor. This occurred once in HtmlUnit in the starting block, 5 times in jEdit post-4.0, and 1 time in jEdit post-4.2.

Encapsulate Existing Code - This step is when existing code gets placed inside new code. This adds some minor functionality behind a new wall, such as capping a value at a minimum so that logic depending on that value will not encounter a divide by zero case or some other unwanted occurrence, as illustrated in Figure 4.1. This occurred 3 times in post-4.2 jEdit.

Before

```
int value = object.getValue();
```

After

```
int value = Math.max(1,object.getValue());
```

Figure 4.1 – Example of Encapsulate Existing Code

Create Branch - This amalgamation of Add Conditional, Add Code, and Push Down Logic was seen together frequently in the early part of jEdit. There would be existing code that needed to still work, but new functionality that was being added as well, so a conditional to select between the added code and the existing code was added, along with the old as is illustrated in Figure 4.2. This occurred 9 times in post-4.0 jEdit and did not appear in any other revision sets.

Before

```
...

doSomething(variable);

...
```

After

```
...

if(flag) {
    doSomething(variable);
else {
    doSomethingNew(variable, otherVariable);
}

...
```

Figure 4.2 – Example of Create Branch

Extract and Modify - This is an extension of the Extract Method refactoring. It manifests itself as a number of lines which get extracted to a new method and then modified to have additional functionality. This occurred 1 time in HtmlUnit from the start, not at all in HtmlUnit post-2.0, 11 times in jEdit post-4.0, and 13 times in jEdit post-4.2.

Expand Object - This type of change happens when a layer of functionality is added between what an object does and what a specific class needs it to do. This change treats an object as a black box that it needs to get additional functionality out of, and it inserts a layer between using the object it is trying to use and itself. Two examples of this from HtmlUnit are the creation of the TextUtil class to validate inputs so that NullPointerExceptions would not be thrown at run time from the String object. The other example is when the WebClient class added a static method to get a Url object which delegated to one of two possible constructors based on the input. Calls to this method replaced previous calls to one of the constructors. This was the most numerous change in HtmlUnit with 6

appearances in the initial revisions, and 3 appearances in the post-2.0 revisions. This appeared 4 times in post-4.0 jEdit, and 4 times again in post-4.2 jEdit.

Boolean to State - This change happens when a method or variable which used to return or contain a boolean value gets changed to a state variable where what was returned previously is one of the states. For example in HtmlUnit a function isEnabled was changed in to getState, where IS_ENABLED was one of the states, see Figure 4.3. This preserves the existing functionality as long as everything is updated to use the new method/state, and allows for new states to be added and maintained easily. This appeared five separate times in the initial revisions of HtmlUnit all in a brief period of time, and never in jEdit. Dig and Johnson also noted an occurrence of this.

Before

```
...
boolean enabled;
...

boolean isEnabled() {
    return this.enabled;
}
...
```

After

```
...
static int DISABLED = 0;
static int ENABLED = 1;
static int OTHER = 2;

int state;

...
int getState() {
    return this.state;
}
...
```

Figure 4.3 – Example of Boolean to State

Split For Type - This happens when something is changed from assuming a type to having different logic based on the type. This is exemplified in HTMLInputElement's jsxFunction_select method. This initially got passed an object, assumed the object's type, and then called one of the object's methods. It was changed to check the object's type to determine what method to call. This preserves existing functionality while allowing for new functionality to be implemented. This is something of an inverse of the Refactoring Replace Conditional With Polymorphism. This only appeared once in HtmlUnit in the initial revisions, and not at all in jEdit.

4.1.2. Functionality Reducing Steps of Evolution

Remove Code - This is the opposite step to Add Code, and abundant.

Remove Method - The inverse step to Add Method, and like Add Method is abundant.

Remove Variable - The inverse of Add Variable and abundant.

Remove Class - The opposite step to Add Class. Either the code in the removed object is no longer needed, or the parts that are needed are merged in to an existing object, and is abundant.

Comment Code - This comments out a block of code, while still leaving it in the source file. This removes whatever effect the commented out code has on the rest of the project, but keeps it around in case it is needed later. In practice this showed up with non-functional code that was being removed until it could be fixed or to preserve an older way of doing things in case it needed to be used again. This change was abundant.

4.1.3. Functionality Changing Steps of Evolution

Add Class - The process of adding a new class to the project. This can be totally new code, or as seen often in jEdit it can be the subset of an existing class that all related to a specific piece of functionality. This subset of an existing class is why this step is included in this section as opposed to the section 4.1.1.

Move Code - This step moves code within a file. This can change functionality, such as if code that sets a variable is moved above or below code that uses that variable. It can also not change functionality at all, such as re-organizing the order of methods within a class.

Change Scope - This step happens when a variable or method changes its scope within a class or a method. This includes changing from static to instance (and the opposite) as well.

Change Code - This is a vaguely structured change that occurs when one segment of code is changed so that it does something different. This is a very general change and classifies places where one block of logic is replaced with another. For example, in jEdit the change in Figure 4.4 took place. In this example both lines accomplish essentially the same function, but one uses the existing contentPane rather than creating a new blank one. The motivation for this would be to do the same thing in a different way (like the Substitute Algorithm refactoring), or to do a similar thing in a different way so the new code either has more capabilities or is more fault tolerant. This is also abundant.

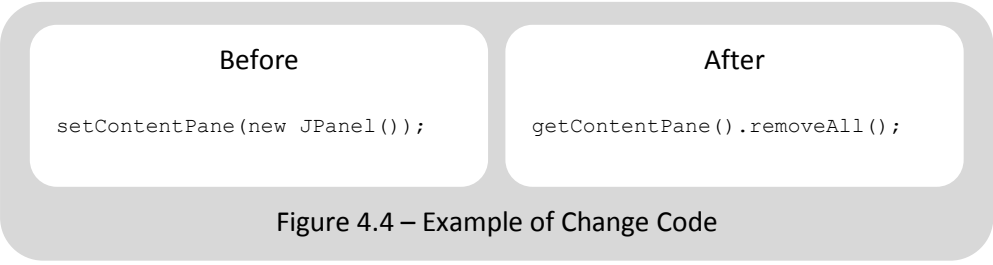

Figure 4.4 – Example of Change Code

Add Conditional - This is a simple change where a conditional block of some type (if, else if, else) is added with new functionality inside of it. At its simplest level it is adding something like what is detailed in Figure 4.5.

```
if(a.equals(b)) {
    object.something();
}
```

Figure 4.5 – An example of what Add Conditional adds

This adds new functionality behind a determination mechanism to see if the new code should be executed or not. This does not attempt to preserve existing functionality in any way and is purely for new functionality. It is also abundant.

Change Method Signature - This happens when most of a method's signature remains the same, but some of them change. In java a method's signature consists of its visibility and modifiers, its return type, its name, its parameters, and any exceptions that it throws. Its signature changes if some of these elements change.

Before

```
public void doSomething(Parameter one)
```

After

```
public static void doSomething(Parameter one, Parameter two)
```

Figure 4.6 – Example of Change Method Signature

Changes like the example in Figure 4.6 usually have a lot of fallout in the codebase because every occurrence of the old version of doSomething must now be updated to call the new version. In jEdit the fallout from Change Method Signature would make up most of the changes in files in a revision where this step occurred. There was one revision in jEdit where there were four different layers of delegate functions, so all the methods that somewhere along the line called the final method had to be changed to include the additional parameter. This occurred 1 time in HtmlUnit from the start, 0 times in HtmlUnit post-2.0, 38 times in jEdit post-4.0, and 28 times in jEdit post-4.2.

Push Down Logic - This is defined as when a segment of code is taken from a broader scope and moved to a more specific scope. This change would be motivated by the need to only do something some of the time, or in the case of loops needing to do it every time. In figure 4.7 this is demonstrated with regards to a screen repaint, such as if the object.something call changed something that needed to

be repainted every time it changed, like a progress bar. This was not observed in HtmlUnit, and was observed 30 times in post-4.0 jEdit and 6 times in post-4.2 jEdit.

Before

```
screen.repaint("blah", x);
for(int i = 0; i < 10; i++) {
    object.something(i);
}
```

After

```
for(int i = 0; i < 10; i++) {
    screen.repaint("blah", x);
    object.something(i);
}
```

Figure 4.7 – Example of Push Down Logic

Pull Up Logic - This is the inverse of push down logic, where a segment of code is taken from a more specific scope and moved to a more general scope. It occurred 9 times in post-4.0 jEdit and 6 times in post-4.2 jEdit. It was not observed in HtmlUnit.

Consolidate Logic - This step takes something that would be executed multiple times with minimal variation and extracts it so that it is only executed once. Doing this allows for one functional bit of code, such as a method call, to be consolidated at one point, so if anything about that line of code changes it only needs to be changed in one place. You can easily see the parallel structure of the object.method calls in Figure 4.8.

Before

```
if(a.equals("3")) {
    object.method(a, "parameter1");
} else if(a.equals("blah")) {
    object.method(a, "parameter2");
} else {
    object.method(a, "parameter3");
}
```

After

```
String parameter = "";
if(a.equals("3")) {
    parameter = "parameter1";
} else if(a.equals("blah")) {
    parameter = "parameter2";
} else {
    parameter = "parameter3";
}
object.method(a, parameter);
```

Figure 4.8 – Example of Condense Logic

This step preserves functionality of the code, and reduces the number of times that the method is called, so any future changes to the method only need to be changed in one place. This process can also be done with a loop and a variable that holds the input/output pairs and be iterated over to select the correct output. This did not occur in HtmlUnit, but occurred 2 times in post-4.0 jEdit and 1 time in post-4.2 jEdit.

Static to Singleton - This is a complicated step that involves converting a static class or a class with all static variables and methods in to a class that follows the singleton pattern. This change involved

changing all static methods and variables to non-static methods or variables on the class. An additional static method (usually called getInstance) is added to the class along with a static variable to hold the instance. The getInstance method's body contains the code to initialize the instance if it is null, otherwise it returns the current instance. Anywhere one of the formerly static methods were called the logic was changed to first call getInstance, and then call whichever function was called before on the return value from getInstance. This did not occur at all in HtmlUnit, and appeared twice in the post-4.2 jEdit.

Before

```
public class SomeObject {
        public static boolean flag;

        public static void doSomething() {
                ...
        }
}
```

After

```
public class SomeObject {
    public boolean flag;
    public static Object instance;

    public void doSomething() {
        ...
    }

    public static SomeObject getInstance()
    {
        if(instance == null) {
            instance = new SomeObject();
        }
        return instance;
    }

}
```

Figure 4.9 – Example of Static to Singleton

	HtmlUnit Start	HtmlUnit Post-2.0	jEdit Post-4.0	jEdit Post-4.2
Add Code	Abundant	Abundant	Abundant	Abundant
Remove Code	Abundant	Abundant	Abundant	Abundant
Add Method	Abundant	Abundant	Abundant	Abundant
Remove Method	Abundant	Abundant	Abundant	Abundant
Add Variable	Abundant	Abundant	Abundant	Abundant
Remove Variable	Abundant	Abundant	Abundant	Abundant
Add Class	Abundant	Abundant	Abundant	Abundant
Remove Class	Abundant	Abundant	Abundant	Abundant
Move Code	Abundant	Abundant	Abundant	Abundant
Change Scope	Abundant	Abundant	Abundant	Abundant
Comment Code	Abundant	Abundant	Abundant	Abundant
Uncomment Code	Abundant	Abundant	Abundant	Abundant
Change Code	Abundant	Abundant	Abundant	Abundant
Add Conditional	Abundant	Abundant	Abundant	Abundant
Change Method Signature	1	0	38	28
Rich Add Variable	1	0	5	1
Encapsulate Existing Code	0	0	0	3
Push Down Logic	0	0	30	6
Pull Up Logic	0	0	9	6
Create Branch	0	0	9	0
Consolidate Logic	0	0	2	1
Extract & Modify	1	0	11	13
Expand Object	6	3	4	4
Static to Singleton	0	0	0	2
Boolean to State	5	0	0	0
Split for Type	1	0	0	0

Table 4.1: Evolution Steps Observed

4.2. Existing Tools

I also found evidence of existing software development tools being used across both projects. Both HtmlUnit and jEdit used javadoc-style comments to describe class and method functionality. Both projects used the ant build system. HtmlUnit used the Apache Maven dependency management system. Both projects utilized xml-heavy support files to configure the project. Both projects integrated external libraries to add functionality to their project, HtmlUnit with Rhino and jEdit with BeanShell. The refactorings observed are listed in Table 4.2 below.

	HtmlUnit Start	HtmlUnit Post-2.0	jEdit Post-4.0	jEdit Post-4.2
Rename	1	2	11	5
Extract Method	3	0	2	17
Introduce Explaining Variable	0	0	4	3
Move Method	1	0	12	0
Replace Conditional With Polymorphism	0	1	0	1
Inline Method	1	0	0	0
Pull-Up Method	1	0	0	0

Table 4.2: Refactorings Observed

5. Future Work

This work scratches the surface of this topic. There is a lot more to be explored here, starting with more hand categorization of revisions. Two projects don't make a trend, but it is a good start. This would strengthen any of the evolutionary steps already detailed, and could lead to a richer and more complete set of evolutionary steps. In addition to this more data would help break the abundant steps in to sets of more specific steps. Surveying projects with languages other than Java would be helpful as well to find commonalities across software engineering as a whole, or if some steps are more characteristic of one type of language. Another step in this direction would be to build tools that can match the more structured evolutionary steps and better identify places where one of less matchable evolutionary steps could be at work. More manually categorized revisions would help with this process since it would give it a larger base to test against. Furthermore categorizing revisions by hand is a very time consuming process, each 75 revision block took approximately 40 hours a piece, so any additional efficiency gain in determining if evolutionary steps show up in a project would be very helpful overall.

The concept of evolutionary steps is a good starting point for developing tools to make software developers' lives easier. Many of the evolutionary steps lend themselves to automation, such as Boolean

to State or Consolidate Logic. Automated versions of these steps could be included in plugins to IDEs like Eclipse, possibly with some mechanism to track their usage so that new changes to programs could annotate themselves with what evolutionary steps they were using.

6. Conclusion

Based on observation of software, I give a set of Steps of Evolution that can be used to categorize the transformations between software revisions. These steps are by no means complete and are open to more information and seek to categorize at the most detailed level what transformations occurred. The frequency of a given step can vary greatly across both time in the project's life and between different projects, but there are common structured changes like Expand Object. Existing tools such as the ant build system, Refactorings, and assorted Java libraries are used to help write mature open source software projects.

Works Cited

Miryung Kim , David Notkin , Dan Grossman, Automatic Inference of Structural Changes for Matching across Program Versions, Proceedings of the 29th international conference on Software Engineering, p.333-343, May 20-26, 2007

D. Dig and R. Johnson. How do APIs evolve? A story of refactoring, Journal of Software Maintenance, 18(2):83–107, 2006.