Software Reengineering Project
Evolving ChEOPSJ: from a prototype to a tool

Alejandro Merlo
Viktor Stojkovski
Rafael Ugaz

June, 2013

1 Introduction

2 First Contact

text

2.1 Chat with the mantainers

text

2.2 Read all the code in one hour

For this task we saw that the ChEOPSJ system is split into ten projects, so we decided to divide them among the three members of our group. Each member read the code of their corresponding projects in around one hour and took notes. Then we explained our findings to the rest of the group and merged the notes together to produce the report shown below. Because of the 'time is scarce' principle, we payed more attention to the following items, as it is proposed in [1]:

- Abstract classes and methods, that reveal design intentions.
- Classes high in the hierarchy, which often define domain abstractions; their subclasses introduce variations on a theme.
- Occurrences of the Singleton pattern that may represent information that is constant for the entire execution of a system.
- Surprisingly large structures, which often specify important chunks of functionality.

- Comments, that can reveal a lot about the design intentions behind a particular piece of code, yet may often be misleading.

The notes taken, separated by project and ordered alphabetically, are the following:

2.2.1 be.ac.ua.ansymo.cheopsj

The project *be.ac.ua.ansymo.cheopsj* only contains the feature.xml file with references to all the other plugins which groups them together as a whole as the ChEOPSJ plugin for Eclipse. Most of the other plug-in projects contain an Activator class that controls the project's plug-in life cycle.

2.2.2 be.ac.ua.ansymo.cheopsj.branding

The project *be.ac.ua.ansymo.cheopsj.branding* contains an about.html website and seems to be related to the plugin information displayed when browsing the plugin to install it in Eclipse.

2.2.3 be.ac.ua.ansymo.cheopsj.changerecorders

The project *be.ac.ua.ansymo.cheopsj.changerecorders* is the first with actual source code. It contains the datastructures used for storing the changes of each type of entity (e.g. class, method or variable). The AbstractEntityRecorder is at the top of the hierarchy subclassed by all recorders in the package. The `StatementRecorder` is also abstract but subclassed only by `LocalVariable` and `MethodInvocation` recorders, it adds extra common methods to them. All recorders inherit the storeChange method which uses the the abstract methods `createAndLinkFamixElement` and `createAndLinkChange` methods, which are implemented differently by each recorder subclass. These two methods, like their names indicate, create and link the famix element object and the change object to the recorder object.

Some classes related to one of the improvements that we must reengineer (support changes of accesses of fields and local variables) are also found in this project but have been excluded from the build path (probably unfinished) so we should look into them later and could use them as a basis.

There are a few comments in the code. Some are meant for the programmer himself, reminding him of things he has to do and some of them are informative but also a bit redundant because the functionality can be easily understood from the (good) naming of the methods or variables.

The tests found check if the change recorders work correctly for additions and removals of the corresponding java element (project, package, class, etc)

2.2.4 be.ac.ua.ansymo.cheopsj.distiller

In *be.ac.ua.ansymo.cheopsj.distiller* one of the main functionalities of ChEOPSJ is implemented. The one of extracting the changes from an existing java project.

This is done by connecting to a SVN repository and then *distilling* the changes that happen in each revision by means of an external library ChangeDistiller from the Evolizer platform.

In the *distiller.cd* package, the class ChangeDistillerProxy accesses the API of the ChangeDistiller library to extract the source code changes from two java files.

In *distiller.popup.actions* the actions taken when the user selects "Distill Changes" and "Distill Additions" from the popup or context menu are implemented.

Finally, *distiller.svnconnection* takes care of connecting and extracting the revisions from the SVN repository. Here we also saw that the SVN url is hard-coded to a path in the machine of a programmer and this could cause problems so we should also look into this in the future.

There are no tests present in this project.

2.2.5 be.ac.ua.ansymo.cheopsj.logger

The *be.ac.ua.ansymo.cheopsj.logger* project takes care of the second main functionaility of ChEOPSJ which is to *log* new changes made to the workspace by the user. In the *logger* package, there are some classes for the initialization of the plug-in on Eclipse. It is important to note that the Cheopsj class employs the singleton pattern which means it is constant during the entire execution of the system.

The *logger.astdiffer* package contains the classes ASTComparator and DiffVisitor, the first class instantiates the second in order to obtain the differences of two input Abstract Syntax Trees (AST), which are instances of the eclipse internal class CompilationUnit. The objective of this classes seems to be to compare two states of a workspace or compilation unit and in this way identify (and log) the changes that the user has made.

In the *logger.listeners* package, we find two classes that take care of the listening to change events and then logging them. From the comments we can infer that it records the fine grained changes made inside the Java editor. The ChangeRecorder class also applies the singleton pattern.

Lastly, the *logger.util* package contains some helper classes for the whole workspace. Although the Constants class is currently not being used anywhere. The project also contains some tests that should look into further.

The project contains some tests that we should look into further.

2.2.6 be.ac.ua.ansymo.cheopsj.model

The project *be.ac.ua.ansymo.cheopsj.model* contains the main change model, which is composed of type of changes as well as type of entities. It also contains a model manager class which seems to be used extensively across the whole system. The ModelManager applies the singleton pattern, which confirms its importance in the system. Among the tasks that the manager performs, are the storing of all the Famix entities in several hash maps as well as all the

changes that have been made to them. It also adds a listener which is an instance of the class ModelManagerListener that responds whenever whenever a ModelManagerEvent is thrown, which happens when a change is added to the model.

The package *model.changes* contains all the types of change possible, with the Change class at the top of the hierarchy and also implementing the IChange interface. One level below in the hierarchy are the AtomicChange and CompositeChange classes and in the lowest the more specific Add, Modify and Remove. Finally the Subject abstract class represents and contains the functionalities of the element affected by the changes.

The *model.famix* package contains the classes that represent all the entities for which changes will be stored. The FamixObject abstract class is at the top of the hierarchy and is extended directly or indirectly by all the classes in the package. It also extends the Subject class from the *model.changes* package explained earlier, which gives all famix objects the functionalities needed to store and manage it's changes. Once we go deeper in the hierarchy, the classes contain more specific methods for the corresponding type of entity they represent (e.g. class, attribute or method).

There are no tests present in this project.

2.2.7 be.ac.ua.ansymo.cheopsj.model.ui

The project *be.ac.ua.ansymo.cheopsj.model.ui* contains the implementation of the user interface. It is divided in 3 packages. *ui.handlers* contains several event handlers for different events. These events are specified in the plugin.xml file and seem to be thrown whenever the corresponding command is called (e.g. for opening the view and to save or load a state).

The *ui.changeinspector* package seems to contain the classes relevant to the change inspector view. The class ChangeSorter implements the functionality to sort (in ascending or descending) the changes in this view. The other classes create the viewer and update the content on it.

Similarly to the change inspector view, the *ui.changegraph* package contains the implementation for the change graph view.

This project deals exclusively with the graphical interface and it should not be necessary to modify in order to add new features.

There are no tests present in this project.

2.2.8 be.ac.ua.ansymo.cheopsj.testtool

The plugin implemented in *be.ac.ua.ansymo.cheopsj.testtool* is used for finding tests relevant to a set of changes. This is another mentioned functionality of ChEOPSJ, namely to provide the user the tests that depend on an entity (e.g. class or method) and that would have to be checked for correctness after that entity has been modified. From the comments we can see that the functionality of the main method findTests is to, first, find the method where the selected

change is in and, second, find the tests that call that method, in other words the relevant tests.

There are no tests present in this project.

2.2.9 be.ac.ua.ansymo.cheopsj.update

The project *be.ac.ua.ansymo.cheopsj.update* contains information about the update site of the ChEOPSJ plugin

2.2.10 org.evolizer.changedistiller

The project *org.evolizer.changedistiller* contains the external library for extracting source code changes from two java files used in the *be.ac.ua.ansymo.cheopsj.distiller* project.

2.2.11 Conclusion

In conclusion, the naming conventions of the whole workspace seem appropriate and help understanding the code much faster. The separation of classes into projects and into packages also states the intention of each group of classes more clearly. The comments could be better, but then again because of the facts just mentioned, they are not always needed. Finally, only two projects (changerecorders and logger) contain tests, the logger is an important part of ChEOPSJ because it implements on of its main functionalities i.e. logging new changes for a project. In the other hand, the distiller project takes care of the other main functionality of ChEOPSJ, which is distilling changes from an existing project, but it does not contain tests. It is possible that we will need to implement tests for this project in the future. The model project seems also very relevant (specially the model manager) and does not have any tests either.

2.3 Skim the Documentation

2.4 Interview During Demo

2.5 Do a Mock Installation

The first time we tried build the system in eclipse, we encountered some errors that prevented us doing it. A few plug-ins were missing, namely the SVN Team Provider and also the SWT library was not found in our Eclipse installation. After we solved this two problems all the errors disappeared and we were able to build the system.

With the system running, we added a mock project to the workspace along with some packages and classes to see the functionality of the system. We noticed from the console that some exceptions were thrown while adding new elements in the java editor saying that the elements could not be found. It would appear that the system tries to locate this elements in real-time while the user is not finished defining them in the editor.

3 Initial Understanding

3.1 Speculate about Design

Because our goal is to restructure ChEOPSJ to support recording changes of more fine-grained level of detail, we focused on the design of the recording of changes. This process is relevant for two of the main functionalities of ChEOPSJ, logging of new changes as well as extracting changes from an existing project.

3.1.1 First class diagram

With our understanding of the system, we recognized three main actors involved in this process: a change, the element that suffers the change and a change recorder that creates these two objects in some way. This can be seen in the first class diagram that we designed, shown in Figure 2.

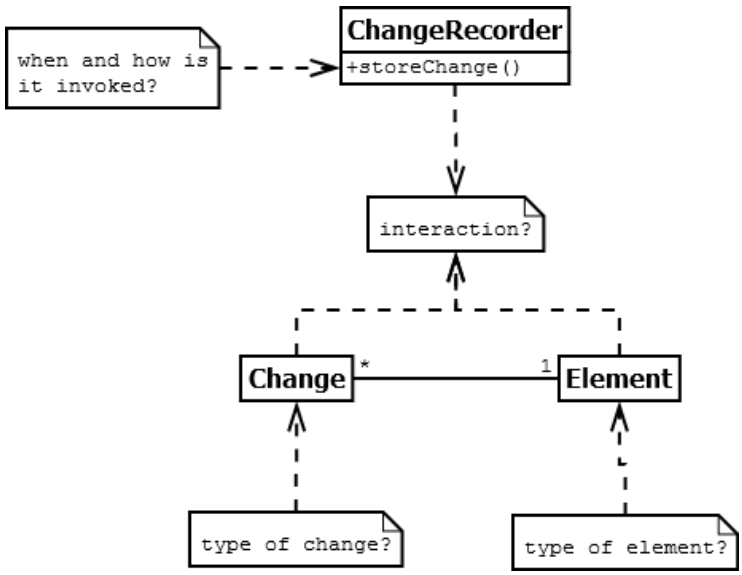

Figure 1: Initial design speculation about recording of changes.

3.1.2 Second class diagram

After examinating the code for mismatches with the initial hypothesis, we adapted the class diagram in several ways, as can be seen in Figure 3:

- First, we renamed the ChangeRecorder to AbstractEntityRecorder, which is at the top of the hierarchy of all change recorder classes. We also added

some of the subclasses of this hierarchy like PackageRecorder, MethodRecorder and ClassRecorder. The way this AbstractEntityRecorder is invoked and by which class is still pending.

- The Element class was renamed to Subject, which is also located at the top of the hierarchy of the elements that suffer the changes, followed by FamixObject and then all the elements like for example FamixClass, FamixPackage and FamixMethod. This was also partially extended in the diagram.
- The Change class was also extended to show the hierarchy of change classes, by adding the subclasses specific types of changes Add, Remove and Modify.
- Finally, the interaction between these three classes was specified, this consists in the ChangeRecorder (AbstractEntityRecorder) creating and also linking together the Change and Subject objects. The storeChange method was confirmed in the source code and is where the creation and linking takes place.

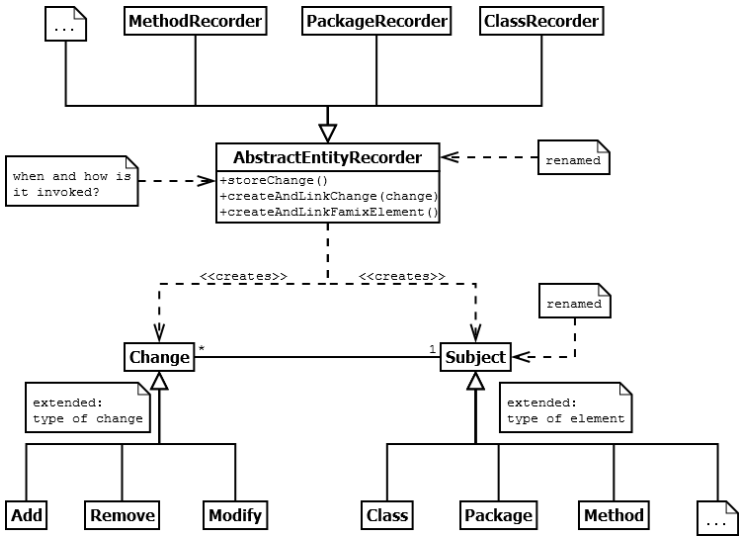

Figure 2: Second design speculation about recording of changes.

3.2 Study the Exceptional Entities

For finding and defining the potential design flaws we have used the tool inFusion which was introduced to us on one of the laboratory sessions. We found this tool very useful in detecting and also graphically representing the results

of its analysis of the structure and design of the ChEOPSJ plugin. In the following section we are going to present and describe our findings regarding the exceptional entities.

InFusion for general grading of the system uses a Quality Deficit Index (QDI), which in our case is 14.1. In the documentation for the tool it is described that the bigger the QDI is, the more significant design problems the tool has found. Among the other important statistics the number of Design Flaws is worth mentioning and inFusion has found 26 of them in our case. The tool has also found 2 God Classes: ModelManager from the package model and the ClassRecorder class from the changerecorders package. This was expected because the both classes are very large and they use many attributes from external classes.

One very useful feature of the inFusion tool is the interpretation of the information collected from the workspace and is based on combination of mathematical formulas which as an input take different measures like Lines of Code (LOC), Number of Packages (NOP) etc. InFusion reported that:

- Class hierarchies are very tall, which means that the inheritance trees have big depths. This statement is true because during the going through the code in one hour we have seen that the hierarchy in the packages model.famix and changerecoreders is pretty large.

- The classes have an average number of methods and also there are not many classes per package, which is true and is positive aspect of the system. It means that the functionalities of the classes are very good divided in methods and also the classes are very well grouped and divided in packages according to their functionalities.

- Methods are long because there are approximately 12 lines of code per method, but on the other hand their logic is simple because there are not many conditional branches in the methods, which is true. Another anomaly that the methods have is the high coupling intensity which means that many other methods are called inside the current method and that makes the code rather complex.

Furthermore, as we suspected inFusion discovered the class ModelManager from the package cheopsj.model as a God class. This class is one of the largest and most complex classes in the whole system. Its job is to store and maintain every famix element created and also take care of all the changes created by the change recorders, that is the reason why this class is very complex and its coupling with the rest of the system is very high. On Figure 3 the graphical representation of the whole model package can be seen, where the ModelManager class is located in the left-bottom corner, colored with strong red color. Judging by the hight, width and color of the class (which represent the number of methods, attributes and design flaws respectfully) it can be concluded that the class ModelManager is a God Class indeed.

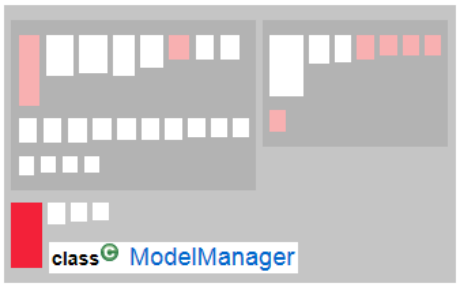

Figure 3: Graphical representation of the package cheopsj.model with inFusion

InFusion has brought up some facts about the god class and suggested some actions that could be taken. The first fact was that ModelManager uses a lot of attributes from external classes which was very obvious from the first look at the code, because those attributes are necessary for correct functioning of the numerous methods used for managing of the processes in the model. The second fact was that this class is excessively large and complex and the reason for that are the methods, which have high cyclomatic complexity (a lot of branching in the code) and high nesting level. The biggest and most complex method is printGraphForGroove(), which has a heavy code branching and deep nesting. Also this method has very weak encapsulation because it calls a lot of external accessors.

The second most complex and also God Class is ClassRecorder, which is located in the changerecorders package and its job is to record every change done to the famixClass element. This class has very weak cohesion with the system because its methods are rarely called and on the other hand it calls a large number of external methods. ClassRecorder is also non-cohesive because its methods does not use its attributes very often, for example the method findParentName is using none of the attributes of the class.

The usage of inFusion has helped us a lot for better understanding of the complexity of the system and detection of some of the design flaws. What really makes the job easier is the graphical representation of some of the most important features of an object-oriented code in our case Java, like encapsulation, cohesion, size, complexity, hierarchies. The small drawback of this method is that there are so many statistics and they are increasing with every other tool, that the developer can sometimes get carried away in a direction away from the essential problems of the design. Also the tools like inCode or inFusion only represent the information collected from the source code, but none of them offers some kind of a solution to some of the design flaws encountered.

4 Reengineering

text

5 References

References

[1] Demeyer, Serge, Stéphane Ducasse, and Oscar Nierstrasz, *Object-oriented reengineering patterns*. Morgan Kaufmann, 2002.