Senior Thesis

“The In-Sync Problem”

In Software Engineering

Extended Abstract

Version 1.0

By Amer Hasan Obeidah

Advised By Lynn Robert Carter, PhD

School of Computer Science

Carnegie Mellon University

Qatar Campus

Software development companies spend major amount of time planning and designing the software product before sending it to the implementation phase. Design is a very important phase since design documents can be created more quickly than the code and when they are effective, they allow professionals to communicate far more quickly and correctly than is possible with code.

However, in the implementation phase software developers concentrate more on code, and they spend a lot of time testing the artifacts and fixing the encountered bugs. Specifically, when developers are pressured, they tend to get stuck in a coding/testing/debugging loop where they find the bugs in code, fix them, and start the process again. They tend to forget and postpone updating the design documents introducing a problem which we call the “in-sync” problem. This paper discusses aspects of the “in-sync” problem, some obvious solutions, and the issues that keep many of these obvious solutions from work. We will then propose an extensible framework to support people struggling with the in-sync problem and will begin to populate that framework with a set of workable solutions. The paper then concludes with an analysis of the work and how others can extend it.

While great deal of work has been done on comparison of programs, we've been unable to find work that specifically addresses the “in-sync” problem of a program and its design documentation. On the one hand, some CASE tools [MSVISIO] can take design and produce some (if not all) of the code. Also, some CASE tools [RS] can take the source code and produce some (if not all) the design documents. On the other hand, none of the CASE tools that we have been able to find will compare the design documentation against the source code and point out where there may be in-sync problems.

Moreover, many papers addressed topics which are related to the “in-sync” problem but are not equivalent. One of those topics is the Process of Reverse Engineering of Class Diagrams, where the inspected source code is converted back into a Class Diagram [DDV06] and manually checked against design documentation. Another topic is the concept of design differencing. In this process the design documents are diffed against each other [OODD] and different reports are produced showing the captured differences. However, these topics do not directly address the comparison between a program’s source code and its design documents which is the subject of this thesis.

We are interested in exploring this issue in more details since in Software development, design is considered to be the starting point of the development life-cycle. From an analysis of books [DAJU02] and [UMLIA05] about software design, we have concluded that UML appears to be the most popular notation to encode the design. Also, we have learned that Class and Sequence Diagrams are the most commonly used. More specifically, software developers start off their design by creating Class Diagrams. These Diagrams show the major components of the system and their interrelationships [IBMW]. Also, they show some details about class members such as attributes (Fields) and operations (Methods) upon them. As a second step, the software developers will need to know how the system’s objects (classes) will interact with each other. Also, they will be interested in knowing the flow and timing of such interactions. For these reasons, the software developers will build Sequence Diagrams since they can learn more about the interactions between objects in a sequential order.

In general, we can say that the main problem we are trying to solve is to help software developers to close, as much as possible, the gap between a program's source code and its design. However, we will not be trying to solve every aspect of the problem since design is too complicated to be addressed in a single thesis. Instead, in this thesis we will be focusing on Class Diagrams and Sequence Diagrams because they are considered the heart of design.

The "in-sync" problem is a broad term that can be used to describe the differences between all the types of design documents of a program and its source code. But, since we are concentrating on Class and Sequence Diagrams, we have structured our solution approach according to three main categories which are described later on.

Our approach to solve this problem is divided into a number of steps. First, we need to have the design and code in a comparable format. In order to do that, we have used the [OMG] XMI (XML Metadata Interchange) notation to encode design. XMI is standard notation where design is serialized in a structured order using XML elements and tags. The usage of XML in XMI encouraged many CASE tools manufactures to integrate it with their products which make it easy to convert design documents into textual representations. [AUM08]

Once we have the XMI file which represents the design, we parse both the source code and the design (XMI file). We will parse the source code to build a data structure which stores the needed information about the code elements. For example, we can store information about classes such as class name, visibility, class members, method calls etc. Also, we will parse the design to transform the XMI representation into a data structure storing information about the design and the elements it represents.

The third step will compare the two data structures created in step two identifying the three main categories on which our solution is based. The first category is called "things we know they are correct". This category reflects the fact that what the design represents is compatible with what the code represents. For instance, having a boolean variable named "stop" in the design and a boolean variable named "stop" in the code indicated design-code compatibility.

The second category is called "things we know they are wrong". This category itself is divided into three sub categories. The first sub-category represents those elements that are found in both design and code but are incompatible. For example, a declaration of an integer variable named balance is not compatible with a declaration of a double variable named balance. i.e. elements names and types in the design must agree with those in code. The second sub-category includes elements that appear in code but not in design. For example, having some variables declared in the code without equivalents in the design. The last sub-category shows those elements which appear in design but do not appear in code.

The last main category in the third step is what we call "things we don't know". There are many reasons why we call this category as" things we don't know". First of all there are notations used in design that cannot be represented in code such as many-many multiplicity. In design multiplicity tells us how many times each class can instantiate another one. However, in code we cannot tell such a thing because the

answer for this question regarding code is entirely dependent on running time and user input which are things we cannot know beforehand.

Moreover, there are certain aspects about code which makes it difficult to analyze. For example, inheritance and methods overriding can cause confusion during the comparison process. More specifically, suppose we have two classes A and B where class A inherits members of class B. In design, this can be shown by drawing an arrow headed from class A towards class B and list the inherited members under class A, or the design can only show the mentioned arrow alone. Both representations are valid. So, if we are dealing with the first representation, then this can create a problem since source code does not have and implementation for the inherited members in class A.

In addition to inheritance, methods overriding can cause a similar problem which is a comparison problem. In other words if class A implements a method called “equals” and class B also implements the same method but using different implementation, then a method call to “equals” can be confusing since we cannot tell from static parsing, if a call to the “equals” method was associated with an instance from class A or class B. So such issues will be classified under “things we don’t know” because to know the exact type, we need a compiled version of the source code and this is out of the thesis scope. [SOOT]

Moreover, there is another problem that is related directly to Sequence diagrams. This is the Control Flow Problem. Specifically, Sequence Diagrams are used to show the sequential flow of messages between the System’s objects. These messages in design are equivalent to methods calls in code. However, the appearance of methods calls inside a control flow structure inside the code makes it impossible for us to know whether the messages which appear in design are compatible with those calls that appear in code. Mainly, we cannot tell from static parsing if the condition of an if-else statement will evaluate to true or false. Another example would be having a sequence of messages that shows a call for a method “X” in the sequence diagram. Equivalently, the code can represent that in a loop structure which can be guarded by a dynamic condition, which cannot be determined until run-time.

Therefore, we have identified three main sections when analyzing the body of a method in the source code. The first section is what we call a “Prefix” includes all the statements that appear in the method’s body and before a control structure. In the “Prefix,” methods calls should appear in the sequence diagrams since there are not issues related to run-time or dynamic binding. The second section is the “Control Structure.” This is the section where we cannot grantee the compatibility of design and code since design lacks the notation for representing such structures. Lastly, we have the third section which we call the “Suffix.” The “Suffix” section includes the statements that appear after the “Control Structure.” Like in the “Prefix” section, the calls that appear in the “Suffix” section must also appear in the Sequence Diagram to be considered compatible.

We will build a GUI tool which demonstrates examples on the main categories mentioned above. The way in which the tool functions is exactly as describes above in our proposed approach. Mainly, the tool will have code and design parsing capabilities in which it builds the mentioned data structures and perform a

comparison between them. The tool can be considered as a starting point of an extendable framework that addresses the “in-sync” problem. It will be an open source tool and available online so that people can download it, enhance our current solution and add more functionality that help in shirking the size of “things we don’t know” section. Moreover, we are going to try to make the tool more interactive where it can ask the human to interfere specifically when dealing with “things we don’t know” section.

Finally, it is important to mention that the problems we mentioned above prevent us from applying obvious solutions. Also, some people may think that we are trying to solve the Halting problem while we are not. On the contrary, we are classifying things that cannot be determined beforehand, under the category of “things we don’t know.” Moreover, for design to be compatible with code, we are forcing certain restrictions such as the necessity of having both the name and type of an element to be the same in code and design in order to be considered compatible.

References:

1. [MSVISIO] http://office.microsoft.com/en-us/visio/default.aspx ; Microsoft VISIO, CASE tool.
2. [RS] http://www-306.ibm.com/software/rational/ ; IBM Rational Rose, CASE tool
3. [DDV06] Girschick, Martin; *Difference Detection and Visualization in UML Class Diagrams*; 2006
4. [OODD] Xing, Zhenchang and Eleni, Stroulia; *UML Diff: An Algorithm for Object-Oriented Design Differencing*
5. [DAJU02] Paul R. Read, Jr. Developing Applications with JAVA and UML; Addision-Wesley ISBN 0-201-70252-5
6. [UMLAI05] Grassle, Patrick; Baumann, Henriette; Baumann, Philippe; UML 2.0 in Action; PACKT ISBN 1-904811-55-8
7. [IBMW] http://www.ibm.com/developerworks/rational/library/3101.html; IBM Library Website
8. [OMG] http://www.omg.org/technology/xml/index.htm; Object Management Group, XMI Technology
9. [AUM08] http://www.altova.com/products/umodel/uml_tool.html; ALTOVA UModel Enterprise Edition 2008, CASE tool
10. [SOOT] http://www.sable.mcgill.ca/soot/; Java optimization framework.