Difference Visualization: Impact Interaction between Code and Model

Susanne Jucknath-John
Technical University of Berlin
Faculty IV, Department of Software Engineering
10587 Berlin, Franklinstr.28-29
Email: susannej@cs.tu-berlin.de

Sebastian Doltze
Technical University of Berlin
Faculty IV, Department of Software Engineering
10587 Berlin, Franklinstr.28-29
Email: Sebastian.Doltze@gmx.de

***Abstract*— Reverse Engineering makes it easier to keep source code and model synchronized, in contrast to former times when it was much more effort to update a hand-made model. Only a problem might arise in situations, when we definitely don‘t want to have code and model synchronized but to see the gap between them. For example to see the (possible) impact of a code change to the model or how much of the code will be affected by a change on model level.**

We present in this paper two approaches to visualize this gap: one, quick, simple analysis and visualization based on a cross-table. And one more sophisticated visualization (based on a matching between model-graph and code-graph). We discuss also some weak spots: our decision to use sequence diagrams as the model in which most oo changes happen during development. Another weak spot is the analysis of a model-to-code matching which can be extremely complicated, but is not described in full length in this paper. And a last one: our testdata for our prototype implementation was taken from open source projects, which didn‘t come with complete model information.

But since State-of-the-Art Tools like Rational Rose offers only a very poor visualization (very large text file) to display the differences between two models, we think that besides all weak spots a visualization of this (possible) gap between model and implementation is useful and needed.

I. Introduction

Every IT-Project contains several levels of information, each level used by different users for their own requirements. It is surely not recommendable to put every piece of information into one document. One reason might be that each user group prefers its own notation, another reason is that large documents tend to be unreadable. Therefore it is common agreement to express the same information about a project in different notations - as long as consistency is given. If we only develop in one direction, as in forward engineering, this consistency is given by declaring the latest development as decisive and binding. But if we allow to make changes on the model as well as in the implementation, like in round-trip engineering, we don‘t have only to adjust this change on one level of information with every other level of information. We also have to communicate (and justify) this change from one group of users to other groups of users, with maybe a very different notation, understanding and view of priorities. A change which seems obvious for a programmer on code level, might have impact on model level and must be explained to the model developers. Because some code changes need a deeper understanding of the abilities of a programming language, this discussion might be laborious. And - not in every case, but often - the only interesting information for him might be how much of his scope is affected. On the other hand, a decision from the bird-view of a model seems to be a good idea on management level, but the programmers might be more interested in how much of the existing code can be preserved. Of course, this problem occurs only in large projects with several kind of developers, who continues over a long time period with numerous changes. A small project with a small group of developers can be handled without these problems.

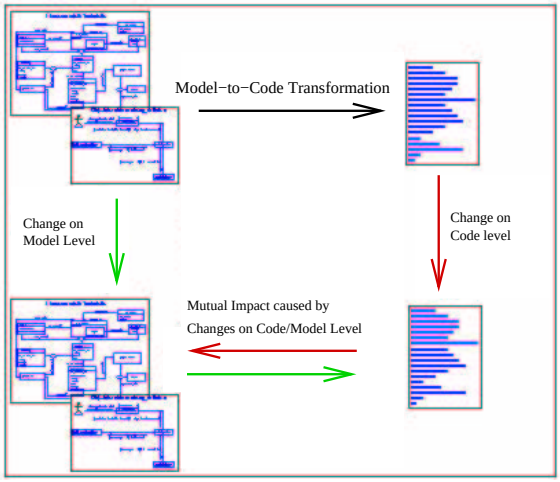

Fig. 1. Problem Description

To summarize our goal: We would like to have an estimation how much a change on one side affects the other side and a satisfying visualization to communicate these changes and discuss them. We would like to call this visualization of the possible impact of the change of one information level (e.g. code) to another information level (e.g. model) for short a *Difference Visualization*.

The paper is is organized as follows: we take a closer look on existing techniques and tools in Section 2, focused on the question if they could provide us with this kind of information (difference visualization). We present a first, coarsely-grained analysis and visualization in Section 3. The main weak spot

of this approach and a possible improvement are presented in Section 4. A second, more complex analysis is presented in Section 5. Some thoughts about visualization for this more complex analysis are discussed in Section 6. The general discussion follows in Section 7 and conclusion in Section 8.

II. State of the Art Tools

There is common developing support for object-oriented languages, especially CASE-Tools like Eclipse or Rational Rose, which allows the user to develop a model as well as an implementation, keeping both of them synchronized. This synchronization might have its limits, as auto-generation itself has it limits, but works well in the long run. Our first question was, whether one of these tools allows a difference visualization. The second question was, if one of these tools can provide us with the needed information about a model change to generate a difference visualization ourselves.

A. IBM Rational Rose Enterprise

Rational Rose is an established CASE-Tool since several years. It can be used to generate a model out of given source code, but in their very own format [1]. This means that no diagrams exist directly after the Reverse-Engineering step (although a class model can be easily generated) but an internal list of IDs. Rational Rose offers a Model Integrator to compare and combine different models into one. But this works only well if just one of these models is generated by Rational Rose and every other model derived from this first model. E.g. we generated one model A out of a project, deleted only one line of code, generated a complete new model B out of this project and ask the Model Integrator about any differences between model A and model B. The tool found more than 800 Differences, nearly all of them about different IDs, listed them in a very large text-file where any trace of the one, real difference (the deleted line of code) was lost. To be fair, this works well for groups which stick to this tool for the whole project where the distribution of IDs never really changes. But even then, the visualization of differences between two models as a large text is not suitable.

B. Borland Together Edition for Eclipse 6.3

Borland Together does not include a difference visualization. It offers design patterns and several functions for a code analysis (like metrics and audit-elements) to minimize the risk of common faults and errors. But it has no focus on an estimation of the possible impact caused by a change on one side to the other side.

C. Omondo EclipseUML 1.0.0 Studio

This relatively slim CASE-Tool (compared with Rational Rose and Borland Together) offers no difference visualization, but a real-time synchronization of class model and code, which made it difficult to see a difference. (It also offers a simple generation of sequence diagrams via Reverse Engineering which was the reason to use it later to fill-up some open-source-projects with model information to test our own algorithmen.

		Model (Sequence Diagram)				
		SeqDiag1	SeqDiag2	SeqDiag3	SeqDiag4	SeqDiag5
Code (Method Calls)	doSym()	0	1	1	0	1
	toString()	0	1	1	1	0
	create()	1	0	0	1	1
	init()	0	1	0	1	0
	destroy()	1	0	1	1	0

Fig. 2. Cross-Table Visualization

III. Coarsely-Grained Analysis and Visualization

We can summarize from the last section that a difference visualization is not in focus of the main CASE-Tools at the moment, although most of them provide us with the information we need to set up a difference visualization ourself.

A first approach for a difference visualization is to select one part of the model and relate it to one part of the code. The decision which part of the model should be chosen depends on the given audience. User-Stories or Use-Cases might be a good start for a typical decision basis on model level. But the underlying sequence diagrams have the benefit to be directly related to the code, because the names of the used methods should be the same. Therefore we choose sequence diagrams to present an extract from the model. Another benefit of sequence diagrams is, that they can present changes on method-level, and this is there most changes during oo-development happen.

The first visualization of this relation between sequence diagrams and method names is a cross-table, where the rows hold the code information and the columns the model information (see Fig.2). A *zero* describes that this method is not used in this sequence diagram, a *one* describes the opposite.

In this example, the method *doSym()* is used in the sequence diagrams *SeqDiag2*, *SeqDiag3* and *SeqDiag5*. In the same example it is easy to see, that the sequence diagram *SeqDiag2* uses *init()*, *doSym()* and *toString()*. So, if we changes the method *doSym()* it might have an effect on *SeqDiag2*, *SeqDiag3* and *SeqDiag5*. If we change the sequence diagram *SeqDiag2* it might have consequences on the methods *init()*, *doSym()* and *doString()*. They might not be needed anymore in this sequence diagram for example, so it won‘t be necessary to be considerate with future changes of this method. This visualization can be enhanced by two new symbols to a difference visualization: a *plus* for a newly added method/sequence diagram relationship, a *minus* for a removed relationship. The benefit is that this visualization is very intuitive and gives a direct impression of the amount of impact as well as of the amount of information at the whole. One drawback is the extreme growth of the cross-table within a large project. (As seen in Figure 3, taken from the analysis of a real world

project.) This can be handled, if only an extract of the cross-table is shown (only the rows and column, which contain at least one plus or minus). Another drawback, more serious this time, is the missing information about any order of events. This will be discussed in the next section.

IV. Improved Coarsely-Grained Analysis

One obvious drawback of the coarse-grained analysis was the absence of any sequence order information. Therefore it also misses every information on indirect effects like a change on one method caused by a change from an underlying method. For example, if the method *doSym()* uses a method *doSum()*, but the information about *doSum()* is missing in the sequence diagram *SeqDiag2* for the sake of information limitation. This can be handled by giving the cross-table more information. Not only the information which methods are listed in the sequence diagrams, but also which methods are used by other methods. This will lead us to the methods which will be always used by a given method and the conditional use of methods, which are not always used. What is needed at this stage is a deeper analysis about the dependency between methods [2]. This leads to program slicing on a object-oriented level and to dependency graphs.

V. Finely-Grained Analysis

We stated in the last section that a first glance about the impact caused by a change can be realized with a cross-table containing information from sequence diagrams as well as from method calls in the source code. We stated also that this cross-table offers a very limited amount of information and that for a deeper insight we need a dependency graph on code-side as well as on model-side. Dependency graphs [3] are originally designed for imperative programming languages, where for each instruction a list of direct following/influenced instructions can be derived. So every node in this graph holds one instruction and every edge symbolizes a direct dependency. If a instruction is changed, the whole underlying subgraph from its node can be affected. Dependency graphs tend to be more complicated when it comes to object-oriented source code and even more complicated for object-oriented models [4]. The next challenge after preparing a code-graph and a model-graph is to find a sound mapping between them. A first approach would be to map between method calls on code-graph and model-graph. But what we really expect from a good dependency graph is the information that if one node is affected, the whole subgraph is affected. It is not as easy as to say, that a direct change on a node in the model graph (plus its underlying consequences on the following subgraph) can be matched to one node in the code graph (plus its underlying consequences). Neither the analysis problem has been solved completely yet - nor the visualization problem of this information. We implemented a prototype, which is able to set up a model graph and a code graph with a simple mapping by method name, as well as three different kind of visualizations. These were the cross-table visualization mentioned before (Figure 3), a tree-map visualization (Figure 4), and a hybrid visualization (Figure 5) offering text information combined with a pie chart view. These visualizations can also be used with a more complex mapping and are justified in Section 7. The risk in using this simple mapping is the false security that with additional information we may have a clear answer about possible impact and not only an estimation. We are not able to do this. We are able to identify the changes, which have definitely no impact to the other side, which have surely an impact, and which might have an impact.

VI. Realization

Both approaches have been implemented in Java and tested at least on two different projects, both chosen from the Open-Source repository *sourceforge*. The underlying principle was to transform source code to XML (via *Java2XML*) as well as the sequence diagrams to XML (via *Omondo*) and navigate between them (via *DOM*) to apply our algorithms.

a) Java Source Code to XML: We used a small program called *Java2XML*[5]. It was enough to compare two XML-Files to recognize a change on code level, because non-interesting changes caused by pretty-printing or additional comments won‘t be visible this way.

b) Sequence Diagrams to XML: We used Omondo to generate the sequence diagrams, because on the one hand, it was important to use real projects to test our approaches. On the other hand free projects on the web, as in *sourceforge*, were mostly not available with a model. This worked well for the simple approach, the cross-table. But *Omondo* turns out to have problems to give method calls the right numeration in a sequence diagram which involves conditional elements. Some numbers were missing.

c) Navigation in XML: The *Document Object Model* (DOM) provides an API for XML-Documents and is like XML itself a specification from W3C. DOM defines a program structure of this documents, a possibility to navigate and manipulate them. For XML is strictly hierarchical, the generated DOM document is a tree.

VII. Alternative Visualization

We have presented so far two visualizations, both very close to the underlying analysis method. One visualization was the enriched cross-table and the basis of the other visualization was a graph structure: a model graph, a code graph and their mapping. It would be easy to say that the best visualization for a graph structure is a graph itself. But given the common amount of information in real world projects, having 500 class files and more, the question arise, whether it might be easier to use a more abstract visualization. For this reason, we implemented a Tree-Map view and a hybrid visualization containing pie charts. Both visualizations give the user an impression how much of of the whole package (or project) might be concerned. We discuss in this section shortly our alternatives for a visualization.

A large project may include over 500 class files, thus a visualization as a large graph is hard to memorize for the

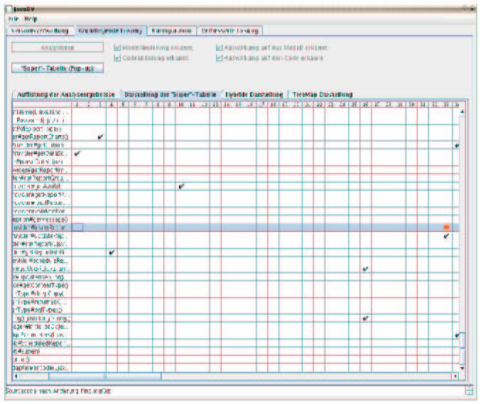

Fig. 3. Cross-table Visualization

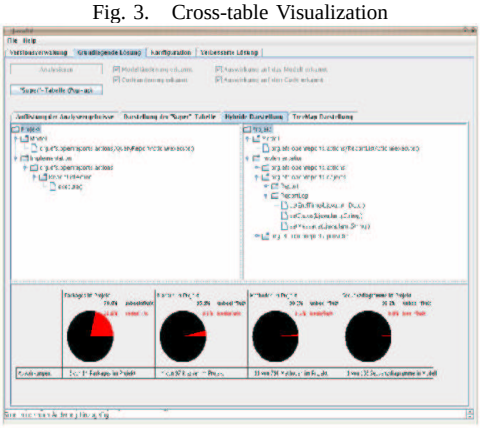

Fig. 4. Hybrid Visualization: List and Pie Graphic

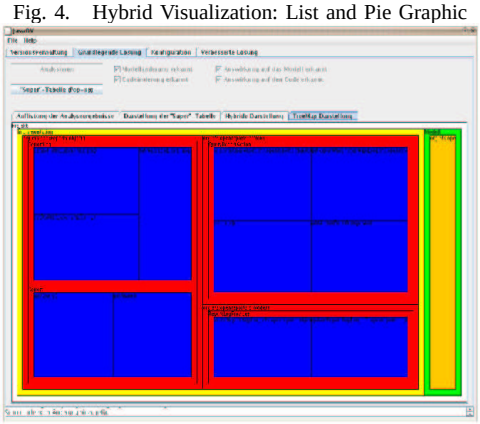

Fig. 5. TreeMap Visualization

human mind. Although it is useful to code another information layer into the layout algorithm - e.g. coupling information about two classes by their layout interval - this is not helpful for normal sized projects [6]. It is to say, that most graph-based visualizations suffer from the challenges of large data [7]. For the same reason a visualization on Lines-Of-Code (LOC)-Level is also difficult. It is possible to give each LOC a color related to the information in this line of code. It is also possible to shrink this line of code to a real line in this color, even to a single pixel in this color [8]. But with a 22000-LOC project we still have a hard time to catch the difference between one visualization and the next one.

The next step would be to choose the information which should be shown - without showing every LOC and every class. E.g. closely related to the Evolution Metric [9] is the Evolution Matrix [10], where the rows describe versions and the columns classes. Every class in a given version is symbolized by a square in this matrix, where the square size depends on the metric. But if we need to visualize more than one metric value per item, but less than approximately 20, a star glyph or a kiviat diagram might be a good choice. In a star glyph every class is symbolized by a single star, each star has as many rays as involved metrics. For a certain class the length of every ray is related to this class value of its metric. A kiviat diagram is similar but contains also information about a development over different releases [11]. In our case the information shown in an item would be e.g. a single sequence diagram or a use-case on model side, and the metrics which gives this item its shape would be derived from the code side. This could be the number of involved methods, method calls or even LOC, plus the number of changed methods, changed method calls and previous LOC. But beside the loss of detailed information, this visualization works again best on a smaller group of items (e.g. class files or packages).

For these reasons we implemented two different visualizations side by side: One to hold only information about the changed parts and the direct impact to the other side (as seen in Fig.3). And another one to show the amount of impact to the whole system (Fig.4 and Fig.5). The later one can be taken out of a wide variety of visualization techniques, like a pie visualization or a TreeMap visualization [12]. A TreeMap divides a given rectangle by the number of children on the first level into smaller rectangles. Each smaller rectangle will be divided also by the number of children on the next level related to this node. And so on, until the last level is reached. Changed items or estimated impact can be visualized by a different colored rectangle. It can be seen in our example (Fig.5), which presents the visualization of our first test-project on package level, that we have two nodes (yellow/green) on the first level, four nodes (red) on the second level and three + four + two + two nodes (blue) on the next level. The estimated impact is visualized by the green rectangle, as in contrast to the yellow one on the same level.

It is not important in this case to show details but to give the user a feeling about the size of upcoming changes and work.

VIII. Discussion

We needed to test our algorithm on model- and source code-information from large software projects. The only available data came from open-source projects, which did not deliver model information. Thus we had to generate the needed sequence diagrams by reverse engineering. One might say, that the surprise is limited if model- and code-information came from exactly the same source. For this is to say, that our tests show more a proof of concept: We can process any sequence diagram in a given XML style, and of course

also sequence diagrams from another source. We tested our algorithm on smaller changes and our results were better compared to the results produced by Rational Rose. Plus, we used our prototype on different versions of the same class and different releases of the same package to see how the model changes over time. This leads also to the question, how the developer dependencies evolve over time, as well as the distribution of knowledge, which we also discussed with a version mining approach in [13].

Furthermore, it lead to the question how representative sequence diagrams are. We have chosen them, because most smaller changes during object-oriented programming concern the order of method calls, adding or deleting them. Another benefit of sequence diagrams is that they are easy to match and reverse engineer from given source code, unlike e.g. use-cases. But although they allow a deeper insight than use-cases or class diagrams, they have the drawback to be used as work material for programmers. Which is not nearly as binding for programmers as use-cases are. This is a kind of Catch-22 situation: If we use use-cases, we will not see the direct consequences of a change on the code-side because they are too high-level. If we use sequence diagrams, they might bear too much low-level information to be interesting for a model developer. But as we stated in the Introduction we need a common ground to visualize the impact information, which is interesting and understandable for both - model developer and code developer.

IX. Conclusion And Future Work

We motivated in this paper the need for a visualization of impact interaction between model and code caused by a change on one side and an estimation about the impact on the other side. This visualization should be used as a discussion basis for the involved developers. In a large project they might come from different areas, where each of them is an expert for his scope but not for every other scope. So if a change on one level (e.g. code) has an impact to another level (e.g. model) it is useful to give an estimation beforehand to the involved members of the other team. Although this analysis and visualization might be useful, it is not part of three different state-of-the-art-tools, Rational Rose, Borland Together and Omondo, (and to our best knowledge to none other tool). And more important, we haven‘t seen this idea in an academic discussion yet. The underlying analysis techniques like program slicing are well known, but haven‘t been used to this problem (especially not with a proper visualization).

For this reason we offered two approaches to this problem. One simple approach combining information from model (sequence diagrams) and code (method calls) into a cross-table. This is a very coarse-grainly analysis but good enough for a quick glance between different developer teams. The main problem with this approach was the lack of any sequence order information. If we add this information we have also to mind conditional effects, which lead us more or less directly to dependency graphs. We implemented in our prototype an algorithm to set up a model graph, a code graph and a very simple matching between them. We implemented also three different ways to visualize these informations (cross-table, TreeMap, hybrid visualization) and discussed several open questions. We gave finally an outlook to use this difference visualization not only to keep track during regular development but also for a quality control during refactoring, as follows:

The information about a gap between model and code may not only be interesting for a better communication between developer teams, it may also be a good estimation on the (possibly) changed behavior as it is interesting for refactoring. There are already a number of approaches to use information from version management systems [14,15,16], but none of them are related to the question about a gap between model and code, evolving over time. It might be interesting to combine their work with our approach. And for the visualization: After finishing the prototype for the difference visualization, we developed an evolutionary graph layout algorithm for graph transformation [17]. This algorithm can also be used well for class diagrams, as a colored class diagram might contain the impact analysis information. The benefit might be that the class diagram is smaller at the begin of a project, so that the user can memorize the whole graph. And if the graph layout allows the user to keep track of the changes of the graph, it might be possible to memorize even larger graphs.

References

[1] Wendy Boggs and Michael Boggs, *UML mit Rational Rose, Verlag moderne industrie buch, 2003*

[2] Robert Arnold and Shawn Bohner, *Software Change Impact Analysis, Wiley-IEEE Computer Society Press, 1996*

[3] D.Binkley and K.Gallagher, *Program Slicing, Volume 43, Academic Press San Diego, 1996*

[4] Loren Larsen and Mary Jean Harrold, *Slicing object-oriented software, 18th international conference on Software Engineering, pages 495 - 505, IEEE Society, 1996*

[5] Harsh Jain, *The Java2XML Project, Creative Commons Licence*

[6] Frank Simon, Frank Steinbruecker, Claus Lewerentz, *Metrics based refactoring, Proc. European Conf. on Software Maintenance and Reengineering, March 2001*

[7] Ric Holt, *Gase: Visualizing Software evolution-in-the-large, Proc. of WCRE 96, pages 163-167, IEEE Computer Society, 1996*

[8] Thomas A.Ball, Stephen G.Eick, *Software Visualization in the Large, IEEE Computer, 29(4), April 1996*

[9] Tom Mens and Serge Demeyer, *Evolution Metrics, Proc. of the 4th Int. Workshop on Principles of Software Evolution, pages 83-86, 2001*

[10] Michele Lanza, *The evolution matrix: Recovering Software Evolution using software visualization techniques, International Workshop on Principles of Software Evolution*

[11] Martin Pinzger, Harald Gall, Michael Fischer, Michele Lanza, *Visualizing Multiple Evolution Metrics*

[12] Ben Shneiderman, *Tree visualization with tree-maps: 2d space-filling approach, ACM Transactions on Graphics, volume 11, pages 92-99, 1992*

[13] Susanne Jucknath-John, Jacek Bochnia, *Developer Dependencies meet Code Dependencies, IASTED International Conference on Software Engineering - SE 2006, February 2006*

[14] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, Andreas Zeller, *Mining version histories to guide software changes, Proc.Int. Conf. on Software Engineering, May 2004*

[15] Michael Fischer, Martin Pinzger, Harald Gall, *Populating a release history database from version control and bug tracking systems International Conference on Software Maintenance, pages 23-32, September 2003*

[16] Jacek Ratzinger, Michael Fischer, *Improving Evolvability through Refactoring Proc. Int. Workshop on Mining Software Repositories, ACM 2005*

[17] Susanne Jucknath-John, Dennis Graf, Gabi Taentzer, *Evolutionary Graph Layout, Submitted*