

JRipples: A Tool for Program Comprehension during Incremental Change

Jonathan Buckner, Joseph Buchta, Maksym Petrenko, Václav Rajlich

Department of Computer Science

Wayne State University

Detroit, Michigan USA 48202

Jbuckner@3dcs.com, {JBuchta,max,Rajlich}@wayne.edu

Abstract

Incremental software change adds new functionality to software. It is the foundation of software evolution, maintenance, iterative development, agile development, and other software processes. Highly interactive tool JRipples provides the programmer with the organizational support that makes the incremental change process easier and more systematic. JRipples supports impact analysis and change propagation, the two most difficult activities of the incremental change.

1. Introduction

Recently, new processes emerged as viable ways to develop and enhance software. They include iterative development, agile development, and software evolution. An essential task within these processes is the incremental software change (IC) that adds new and previously unavailable functionality to existing software. IC is also an important task of the classical process of software maintenance and plays a key role in practical software engineering: software engineers spend a large part of their effort working on ICs.

While IC consists of several activities, impact analysis and change propagation play a key role. *Impact analysis* is a part of IC design and it is a traditional topic of the program comprehension field. Its purpose is to decide on the strategy for the change and to find all classes that are going to be affected. *Change propagation* is part of IC implementation and guarantees that all required changes in all affected classes were made. These activities are based on the programmers' comprehension of the individual software components.

Tool JRipples assists the programmer in two ways: It supports relevant program analysis, and it manages organization of the steps that comprise the impact analysis and subsequent change propagation. This step organizer is the novel feature of JRipples.

2. Tool JRipples

Tool JRipples is based on the philosophy of “intelligent assistance” [7] which requires close cooperation between the programmer and the tool. The tool analyzes the program, keeps track of the inconsistencies, and automatically marks the classes to be visited by the programmer. It covers the algorithmic tasks that are often difficult or error-prone for humans. This frees the programmer to concentrate on important decisions, rather than on the minutiae of the IC process.

The programmer selects, visits, comprehends, and/or modifies the classes that appear in the backlog of things to do, and attends to the tasks for which algorithms either do not exist or are difficult to implement. The programmer also corrects inaccuracies of the JRipples analysis in a timely fashion and avoids accumulation of the analysis error.

JRipples is implemented as a plug-in for the Eclipse platform [10]. It consists of three Java packages that implement the parser, database with organizational rules, and user interface.

The parser analyzes a system that consists of a set of Java files and extracts class dependencies. The extracted dependencies are stored in JRipples database. The database keeps these dependencies and the status of the IC process.

The snapshot of the Eclipse interface with active JRipples view is presented in the Figure 1. (JRipples view is in the lower right part.) The view displays the list of the classes sorted by their status. The class list always indicates what has been accomplished and what is to be done and hence provides methodological guidance to the user. This monitoring and management of the class statuses through the IC process is the novel feature of JRipples.

Clicking on a class in the list allows the user to display and then edit this class code, make changes in the class status, undo previous actions, etc. The

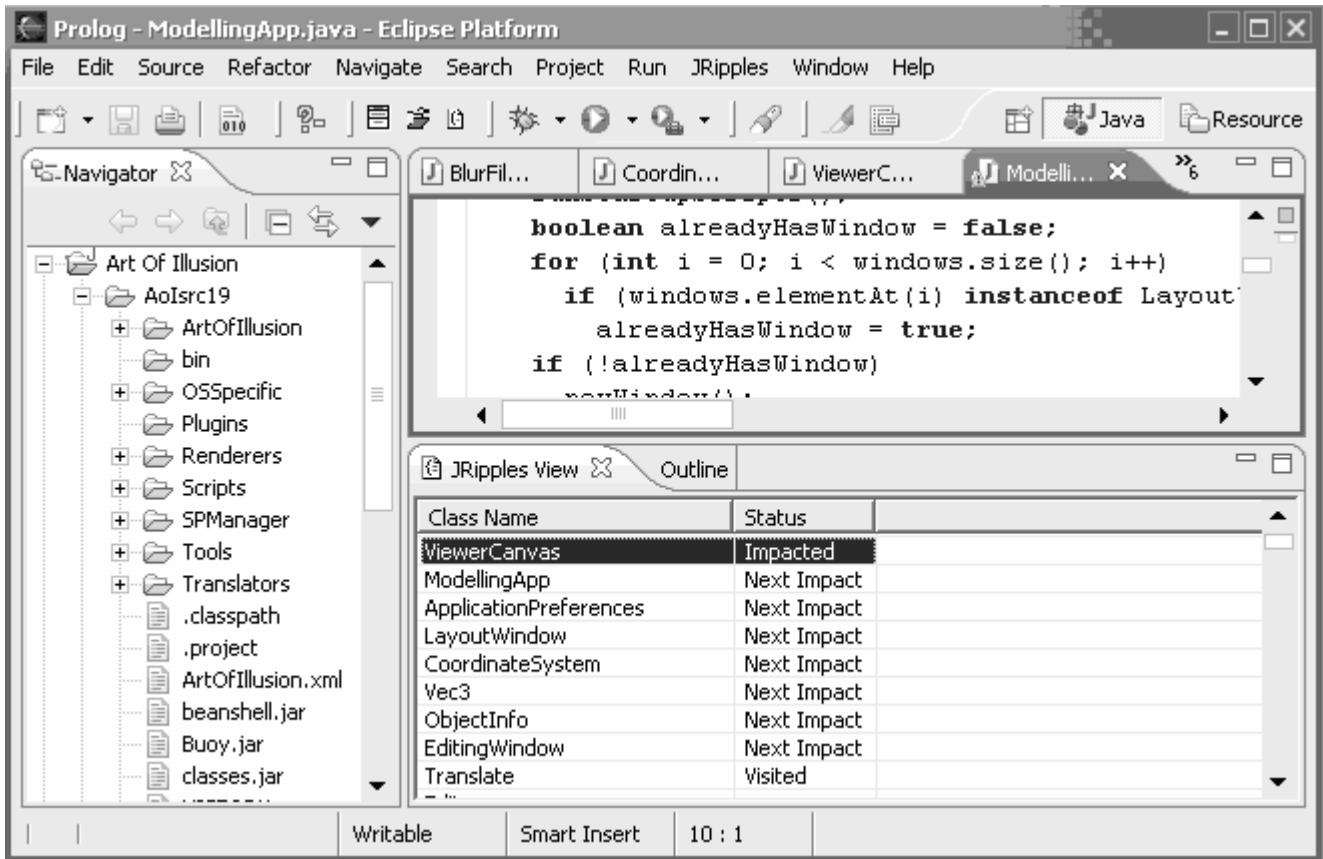


Figure 1. JRipples View as a part of the Eclipse Environment

organizational logic of these user actions was defined by graph grammar in [16].

The programmer starts impact analysis by identifying at least one class of the impact set. After that, the remaining classes of the impact set are discovered in a step-by-step look at the neighbors. JRipples view of Figure 1 presents the middle of the impact analysis process. Class status “Impacted” indicates that the programmer visited the class and concluded that the class is going to change in the IC. Class status “Visited” indicates that the programmer also visited the class but concluded that the class will not be changed. Class status “Next Impact” is a temporary status that indicates that this is a neighbor of an “Impacted” class. The programmer has to visit this class and decide whether its final status will be “Visited” or “Impacted”. The impact analysis ends when there are no longer any classes with the status of “Next Impact”.

Support for change propagation is similar. Figure 2 contains a JRipples view in the middle of the change propagation. Class status “Changed” indicates that the programmer changed the class. Class status “Visited” indicates that the programmer visited the class and concluded that the class will not be changed. Class status “Next Change” is a temporary status that indicates that a

neighbor class has changed, and as a consequence this class may change also. The programmer has to visit this class and decide whether to change it or not. The change propagation ends when there are no classes with the status of “Next Change”.

Presently, JRipples supports analysis on the granularity of classes, but can be easily adapted to support other granularities. The granularity of classes is a natural granularity for impact analysis and change propagation in object-oriented programs, because a well-designed class has a strong internal cohesion and weak external coupling, and hence it is a natural unit of scope for a change.

Any imprecision that arises during the analysis process is immediately corrected through the close interaction between the programmer and the tool.

3. Demonstration of JRipples

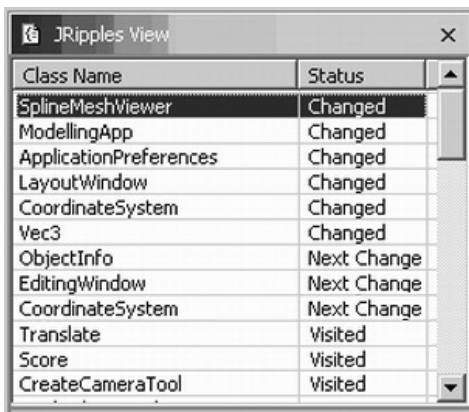
In order to demonstrate JRipples we will perform a complete IC on the open source project Art of Illusion (AOI) [9].

AOI is a 3D modeling, rendering, and animation studio written entirely in Java. Currently, there are four different modes of object display: Wireframe, Shaded,

Smooth, and Textured. Our change request will add another display mode called Pointframe. This mode is similar to the Wireframe, but it displays a set of points that comprise an image instead of the set of lines.

The demonstration emphasizes impact analysis and change propagation. The JRipples view in Figure 1 shows the situation after the first step of impact analysis, where the programmer identified the class *ViewerCanvas* as the first class of the impact set, and *JRipples* identified all classes interacting with it as the next classes to be visited. Figure 2 shows JRipples view in the middle of the change propagation where some classes have been changed and other classes are scheduled for a change.

When the change is completed, AOI will be recompiled and run with the new functionality. We will also discuss the algorithms that are part of JRipples, and the classroom experience with JRipples.



Class Name	Status
SplineMeshViewer	Changed
ModellingApp	Changed
ApplicationPreferences	Changed
LayoutWindow	Changed
CoordinateSystem	Changed
Vec3	Changed
ObjectInfo	Next Change
EditingWindow	Next Change
CoordinateSystem	Next Change
Translate	Visited
Score	Visited
CreateCameraTool	Visited

Figure 2. JRipples view showing intermediary results of change propagation

4. Related work

Past case studies [8], [17] investigated IC in both functional and Object Oriented software technology and provided the understanding on which JRipples development is based.

IC consists of several activities. One of them, concept location, is an activity that indicates where the core of the change might be found in the source code. The input to the concept location is the change request that contains domain or solution concepts, often expressed in natural language. The output is a set of software components that implements the concept. The concept location finds an initial subset of the impact set that is later expanded by impact analysis into the full impact set.

Concept location has traditionally been an intuitive and informal process. Biggerstaff et al. [4] discussed concept location through textual search of the code, where the user finds the names of the concept or their

synonyms in program identifiers or comments. We proposed an approach based on a search of the dependency graph of the program [8]. The Software Reconnaissance method [22] is one of the earliest examples of utilizing dynamic information to support concept location. We also compared these approaches in a case study and concluded that all three approaches are effective [21].

One of the early papers on impact analysis [15] proposes an incremental sequence of actions that is based on program dependencies and is similar to the philosophy presented by JRipples, although that paper does not take any steps towards implementation. The early impact analysis papers were summarized in [5].

More recent work appears in [3], where a software tool for impact analysis is proposed. The tool conducts analysis of program dependencies, but does not support the alternating roles of the tool and the programmer as implemented in JRipples. Paper [6] traces impact analysis in UML models that consist of several documents tied together by consistency rules. Several recent papers present algorithms that estimate the impact of a change on tests [12], [18]. It was also noted in [13] that unaided programmers tend to underestimate the size of the impact set.

Early paper on change propagation [23] established basic terminology. We proposed a theoretical model of change propagation based on graph rewriting in [16] that became a theoretical foundation for JRipples.

Static source code analysis plays a supporting, but crucial role in the development of JRipples. In our work, we relied on the results of other researchers in this important field. Among the analysis algorithms, of particular interest are algorithms that deal with granularity of classes and produce object-relation diagrams (ORD) [2],[14],[20]. An incremental algorithm for the construction of call graphs is described in [19].

Several industrial tools extract program dependencies that play an important role in both impact analysis and change propagation. Examples of such tools are Codesurfer [1] and Imagix 4D [11]. CodeSurfer extracts fine-grained control and data-flow dependencies while Imagix 4D presents higher granularity of functions. These tools represent the current state-of-the-art tool support. However these tools do not support an organization of the programmer actions, therefore they do not support methodological aspects of impact analysis and change propagation.

5. Future work

The main novelty of the current JRipples prototype is the support of IC process and early data indicate that the support is effective. Based on our experience, we plan additional operations that would further improve the tool-programmer interactions and transfer additional

responsibilities from the programmer to the tool. In particular, we want to add support for concept location that precedes impact analysis, support for opportunistic refactoring that is applied before change propagation starts, and so forth.

Although the program analysis is not the main thrust of our effort, experience with JRipples indicates several needs in that area. Among the most conventional directions, we plan to incorporate other granularities of the analysis into JRipples and allow the programmer to choose the granularity that is appropriate for the particular IC.

We will also categorize interactions that propagate different kinds of change, and use this categorization to improve precision of the analysis. We will employ coupling metrics to suggest which classes are most likely to be involved in a change, and that will give further aid to the programmer.

Another topic for the future work is to improve recall of the analysis [24]. Our experience indicates that recall is an important factor in the quality of the analysis and low recall is harder for the programmer to correct than low precision.

Based on the results of the case studies of JRipples, we believe that JRipples can significantly help the programmer during IC. Since IC is an important part of programmer's work, JRipples can make a positive impact on the productivity and quality of this work. We believe that JRipples can particularly help novices and students to learn the art of IC, and therefore it can also turn into an important educational tool.

6. References

- [1] Anderson, P., Reps, T., Teitelbaum, T., and Zarins, M., "Tool Support for Fine-Grained Software Inspection", *IEEE Software*, 20, 4, August 2003, pp. 43-50.
- [2] Barowski, L. A. and Cross, J. H., "Extraction and Use of Class Dependency Information for Java", in *Proceedings Working Conference on Reverse Engineering*, 2002, pp. 309 - 316.
- [3] Bianchi, A., Visaggio, G., and Fasolino, A. R., "An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models", in *Proceedings International Workshop on Program Comprehension*, 2000, pp. 149-158.
- [4] Biggerstaff, T. J., Mitbender, B. G., and Webster, D. E., "Program understanding and the concept assignment problem", *Communications of the ACM*, 37, 5, May 1994, pp. 72-82.
- [5] Bohner, S. and Arnold, R., *Software Change Impact Analysis*, Los Alamitos, CA, IEEE Computer Society Press, 1996.
- [6] Briand, L. C., Labiche, Y., and O'Sullivan, L., "Impact Analysis and Change Management of UML Models", in *Proceedings International Conference on Software Maintenance*, 2003, pp. 256- 265.
- [7] Brooks, F. P. J., "The Computer Scientist as Toolsmith - II", *Computer Graphics*, 28, November, 1994, pp. 281-287.
- [8] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependency Graph", in *Proceedings International Workshop on Program Comprehension*, 2000, pp. 241-249.
- [9] Eastman, P., "Art of Illusion", Date Accessed: 09/13/2004, Online at <http://aoi.sourceforge.net/>.
- [10] Eclipse, "Eclipse", Date Accessed: 01/15/2004, Online at <http://www.eclipse.org>.
- [11] Imagix, "Imagix 4D visualization tool", Date Accessed: 02/05/04, Online at <http://www.imagix.com/products/imagix4d.html>.
- [12] Law, J. and Rothermel, G., "Whole Program Path-Based Dynamic Impact Analysis", in *Proceedings International Conference on Software Engineering*, 2003, pp. 308 - 318.
- [13] Lindvall, M. and Sandahl, K., "How Well do Experienced Software Developers Predict Software Change?" *Journal of Systems and Software*, 43, 1, 1998, pp. 19-27.
- [14] Milanova, A., Rountev, A., and Ryder, B., "Constructing Precise Object Relation Diagrams", in *Proceedings International Conference on Software Maintenance*, 2002, pp. 586- 595.
- [15] Queille, J.-P., Voidrot, J.-F., Wilde, N., Munro, M., "The Impact Analysis Task in Software Maintenance: A Model and a Case Study", in *Proceedings International Conference on Software Maintenance*, 1994, pp. 234 - 242.
- [16] Rajlich, V., "Modeling Software Evolution by Evolving Interoperation Graphs", *Annals of Software Engineering*, 2000, pp. 235-248.
- [17] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", *IEEE Software*, 21, 4, July 2004, pp. 62-69.
- [18] Rountev, A., Milanova, A., and Ryder, B., G., "Points-to analysis for Java using annotated constraints", in *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 43-55.
- [19] Souter, A. L. and Pollock, L. L., "Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance", in *Proceedings International Conference on Software Maintenance*, 2001, pp. 682 - 691.
- [20] Tip, F. and Palsberg, J., "Scalable propagation-based call graph construction algorithms", in *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 281-293.
- [21] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, 65, 2, 2003, pp. 105-114.
- [22] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, 7, 1995, pp. 49-62.
- [23] Yau, S. S., Nicholl, R. A., Tsai, J. J., and Liu, S., "An Integrated Life-Cycle Model for Software Maintenance", *IEEE Transactions On Software Engineering*, 15, 7, 1988, pp. 58-95.
- [24] Yu, Z. and Rajlich, V., "Hidden Dependencies in Program Comprehension and Change Propagation", in *Proceedings International Workshop on Program Comprehension*, 2001, pp. 293 - 299.