The LAN-simulation: A Refactoring Teaching Example

International Workshop on Principles of Software Evolution (IWPSE 2005)

Serge Demeyer, Filip Van Rysselberghe, Tudor Gîrba, Jacek Ratzinger, Radu Marinescu,
Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza,
Matthias Rieger, Harald Gall, Mohammad El-Ramly

Abstract

The notion of refactoring —transforming the source-code of an object-oriented program without changing its external behaviour — has been studied intensively within the last decade. This diversity has created a plethora of toy-examples, cases and code snippets, which make it hard to assess the current state-of-the-art. Moreover, due to this diversity, there is currently no accepted way of teaching good refactoring practices, despite the acknowledgment in the software engineering body of knowledge. Therefore, this paper presents a common example —the LAN simulation— which has been used by a number of European Universities for both research and teaching purposes.

1 Introduction

Numerous scientific studies of large-scale software systems have shown that the bulk of the total software-development cost is devoted to software maintenance. What may seem surprising at first is that this percentage is increasing: "the more modern methods you use in building software, the more time you spend maintaining the resulting product" [9]. The explanation for this observation is that modern software systems —more than their traditional counterparts— must cope with changing requirements, hence must continue to evolve.

Refactoring is widely recognized as one of the principal techniques applied when evolving object-oriented software systems. The key idea is to redistribute instance variables and methods across the class hierarchy in order to prepare the software for future extensions [16, 17]. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [7]. As such, refactoring has received widespread attention within both academic and industrial circles, and is mentioned as a recommended practice in the software engineering body of knowledge [21].

The success of refactoring implies that the topic has been approached from various angles. This has as the unfortunate consequence that refactoring research is scattered over different software engineering fields, among others object-orientation, language engineering, modeling, formal methods, software evolution and reengineering (see [15] for an overview of refactoring research). Therefore, it has been very difficult to compare research which in turn will hamper future progress. Also for teaching purposes this has serious consequences, because the current diversity implies a lack of standardization which goes against the very idea of a *standard* body of knowledge [21].

This problem has been recognized by a number of European Universities and Research Institutes which have founded a research network —named RELEASE— addressing this problem. During four years they have exchanged tools and cases in order to reach a consensus on what represents a typical refactoring scenario to be used for both teaching and research. We aimed for a small yet realistic situation which (a) illustrates some typical steps in a refactoring sequence and (b) demonstrates the applicability of available tools and tool prototypes. Thus, the goal of the example is not to serve as a benchmark for finding "the best" tool, but rather illustrating how tools may complement one another. Moreover, the example should be applicable in a classroom setting where we teach some basic refactoring principles, yet allow students to experiment with a series of tools and tool prototypes. This paper argues the pros and cons of the selected example — a simulation of a local area network (LAN-simulation for short)— by comparing the applicability of a selection of representative tools.

The remainder of this paper is organized as follows. In section 2 we provide the necessary details about the LAN-simulation to allow teachers and researchers to reuse and replicate the example. Then, section 3 lists the tools and tool prototypes we apply on the LAN-simulation, organizing them into three categories (predictive, curative and retrospective). Next, section 4 discusses the suitability of the LAN-simulation by evaluating the applicability of each of the different tools and investigating whether tools do complement one another. We conclude the paper with a call for

arms to use the LAN-simulation as a common refactoring example in both research and teaching (section 5).

2 The LAN-Simulation

This paper proposes to use a simulation of a Local Area Network (LAN) as the basis for a typical refactoring sequence. The example starts from a requirement specification (a set of use cases — see Table 1) and with five iterations (1.0 — 1.4) delivers a first increment to the customer. The system is written in Java and iteration 1.4 consists of 4 classes totaling 677 lines of code (see Figure 1). The implementation comes with a set of automated regression tests written with the JUnit testing framework and containing one extra class with 280 lines of code. The use cases, design and code plus the lab assignments for the students can be downloaded from the "Artefacts" page at HTTP://WWW.LORE.UA.AC.BE/.

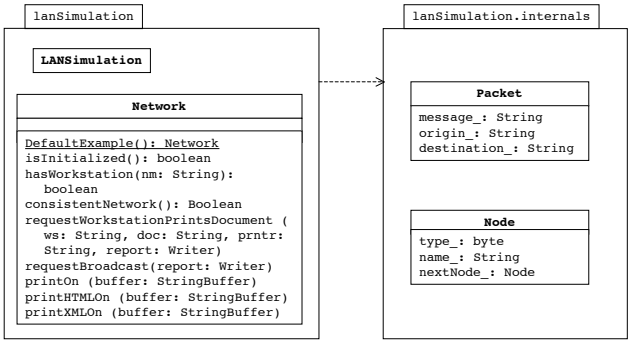

Figure 1. The UML Class Diagram at the end of the first increment — the god class NETWORK should be refactored

To illustrate a realistic refactoring situation, the 1.4 release is done in a procedural style. Most of the functionality is implemented in a single class NETWORK and the other classes mainly serve as data-holders – in refactoring parlance such a single class monopolizing control is called a "god class". Obviously, this class shows some typical code-smells (duplicated code, nested conditionals, navigation code) that warrant attention. Moreover, the use-cases we are expected to add in the second release will even enlarge the god class, so it is is best to refactor it now before starting increment 2.0. The JUnit tests should be used to demonstrate that the system does not regress during the refactoring process. However, students must make an assessment whether the provided tests are adequate (i.e. sufficient coverage of use cases 1.0 — 1.4) to serve as a safety net during refactoring.

Of course there are several paths a software engineer can follow when refactoring this design; the following is just one possibility based on some reengineering patterns [2]: (a) EXTRACT METHOD to remove some duplicated code (release 1.5) (b) MOVE BEHAVIOUR, to move methods close to the data they operate upon (release 1.6), (c) ELIMINATE NAVIGATION CODE to reduce the coupling between some classes, and finally (release 1.7) (d) TRANSFORM SELF TYPE CHECKS to change switch statements into polymorphism (release 1.8). After those refactorings, a lot of the code in NETWORK will be moved onto the class NODE and its newly created subclasses PRINTER and WORKSTATION; some code will be moved onto PACKET as well.

2.1 Teaching Experience

The LAN simulation was originally used by consultants as a way to teach programmers trained in procedural style programming the principles of good object-oriented design. Back then, the simulation was mainly used in CRC-card sessions. Afterwards, the LAN simulation has been picked up by some universities to be used in several introductory programming courses to illustrate and teach good object-oriented design. This is mainly because the example is sufficiently simple for illustrative purposes, yet covers most of the interesting constructs of the object-oriented programming paradigm (inheritance, late binding, super calls, method overriding), hence we used it in prior occasions [3, 14].

Yet, when trying the LAN in an advanced software reengineering course, we found that students trained in object-oriented principles found the sequence of refactoring steps too artificial. Hence, we completely revised the example to reflect a more realistic refactoring situation and this is the one we propose here. This version has been used as assignment for a refactoring lab session, where students are given the requirements specification, the code up until release 1.4 and the request to refactor it in order to accomodate use-cases 2.0 — 3.0 (see Table 1). The main goal of the lab is to have a practical hands-on experience with refactoring tools and relate it with other software reengineering skills, such as redesign (redistribute responsibalities), problem detection, regression testing. The lab requires students with good knowledge of object-oriented design principles (inheritance, polymorphism, encapsulation) and which are able to evaluate several design alternatives.

The lab has been used with students from the University of Antwerp — Belgium(12 students in 2004, 9 students in 2005), the University of Mons-Hainaut — Belgium (29 students in 2005), the University of Groningen — The Netherlands (3 students 2005). Moreover, the lab has been used in training sessions with professionals as well. We did not conduct any formal questionnaires, yet many students spontaneously told us that they enjoyed the session and that it

changed the way they think about reengineering in general and refactoring in particular.

3 Selected Tools and Techniques

Of course, there is a vast number of techniques and associated tools that are applicable in a refactoring context, and it is not feasible to have students experiment with all of them. Therefore, teachers should necessarily restrict themselves to a few representative ones. We argue that to be sufficiently representative, the selected tools should cover three main categories of software evolution tools, namely predictive, curative and retrospective [4].
(a) Predictive (= before evolution). This category covers all techniques and tools that allow maintainers to make decisions concerning the parts of the software that should be improved. In refactoring terms, this means tools that are able to detect the so-called *code smells*, i.e. symptoms for code that should be refactored. We selected three tools that allow a software engineer to identify those parts of the software that need to be evolved due to a lack of quality.

- DUPLOC [5] Duplicated code is one of the most commonly cited code-smells, hence the inclusion of a clone detection tool.
- CODECRAWLER [1, 11] Visualisation tools are often used to inspect class hierarchies and detect potential cases of abusive inheritance, god classes, etc.
- INSIDER [13] Quality metrics are also often used to identify potential design problems. At this point we selected INSIDER (part of the IPLASMA toolkit for quality assurance), because of its explicit support for a list of typical object-oriented design flaws. Additionally INSIDER does also provide support for the detection of code duplication, and allows an easy correlation of detected clones with additional structural information (e.g., duplication detected between sibling classes).

(b) Curative (= during evolution). This category concerns techniques and tools that support the actual changes to the software system. Here we selected a passive (i.e., infrastructure that allows to keep track of the changes) and two active tools (i.e. tools that support the actual change process).

- CVS was selected to keep track of the different versions, mainly because it is the de facto standard which is well integrated with other tools.
- ECLIPSE is currently the most popular refactoring engine, hence is an obvious inclusion.
- JUNIT —also very popular— is used as the regression testing harness that is used to verify whether the refactorings did preserve the behavior of the system.

(c) Retrospective (= after evolution). This category includes techniques and tools that allow to analyze where, how and why a software system has evolved in the past. Here we selected four tools that analyze the changes by inspecting the change log.

- CVS CHANGE SCATTERPLOT [19] The scatterplot visualisation is used to get a first overview of which files changed most frequently.
- CLASS EVOLUTION MATRIX [10] Next, we study the class evolution matrix to see when and how the classes have changed.
- HIERARCHY EVOLUTION [8] Then, the hierarchy evolution is used to assess the stability of the inheritance hierarchy.
- EVOLENS [18] Finally, the Lens-View provides insight into the evolution of classes and their relationships.

4 Evaluation

This section reports the results produced by the tools selected in Section 3. Note that we used the order in which students would normally apply the tools.

4.1 Predictive tools

4.1.1 Duploc

As a first problem detection tool, we used Duploc, a simple line-based clone detection tool [5]. Duploc removes white space from all lines of code and then compares all lines against all other lines using exact string matching. It reports matches in a so-called dot-plot — therein, a duplicated line of code appears as a dot and duplicated statement sequences appear as little diagonals. Figure 2 zooms in on the file showing the most duplication, which is in this case NETWORK. The segments of duplicated code concern the logging of messages (use case 1.1), the print accounting (use case 1.3) and the various reports describing the network status in ASCII, HTML and XML (use case 1.1). Note that the try-catch block surrounding sequences of Java `write` statements, causes small disruptions in duplicated lines, so clone detection tools should be able to deal with those.

Since we knew beforehand where pieces of duplicated code occur, we could verify whether the tool missed any clones. We did not discover any false negatives although one had to take care to also inspect diagonals where little

Table 1. Overview of the use cases for the LAN-simulation.

1.0	BASIC DOCUMENT PRINTING	A workstation requests token ring network to deliver document to a printer.
	REPORT NETWORK STATUS	Print a report of the network status as ASCII, HTML or XML
1.1	LOG PACKET SENDING	Each time a node sends a packet to the next one, log it on a log file.
1.2	POSTSCRIPT PRINTING	A packet may start with "!PS" to invoke a postscript printing job.
1.3	PRINT ACCOUNTING	Printers register author and title of document being printed for accounting.
1.4	BROADCAST PACKET	A special type of packet "BROADCAST'" is accepted by all nodes.
2.0	READ FROM FILE	Read network configuration and network actions from a file.
2.1	GATEWAY NODE	Introduce a special "gateway" node, which can defer packets with an addressee outside the current subnetwork.
2.1	COMPILE LIST OF NODES	Gateway uses BROADCAST PACKET to periodically collect all addresses on the subnetwork.
3.0	SHOW ANIMATION	Have a GUI showing an animation while the simulation is running.

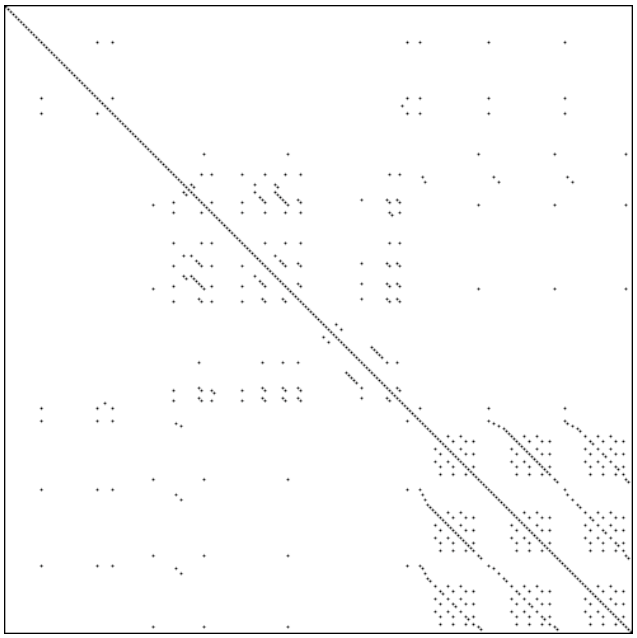

Figure 2. Duploc dotplot, showing that the class NETWORK contains various clones.

holes appear as those were the ones caused by the surrounding try-catch blocks. Duploc did also detect quite a lot of duplicated code in the tests and in the main class, which students are not supposed to refactor. This illustrates a major weakness of Duploc (and many other clone detection tools), i.e. they cannot prioritize the detected clones.

4.1.2 CodeCrawler

Next we tried CodeCrawler which provides various ways of identifying potential design problems in class hierarchies [1, 11]. Figure 3 shows an overview of the inheritance tree, where each rectangle represents a class. The width of a class corresponds to the number of attributes, the height to the number of methods and the gray value to the number of lines of code.

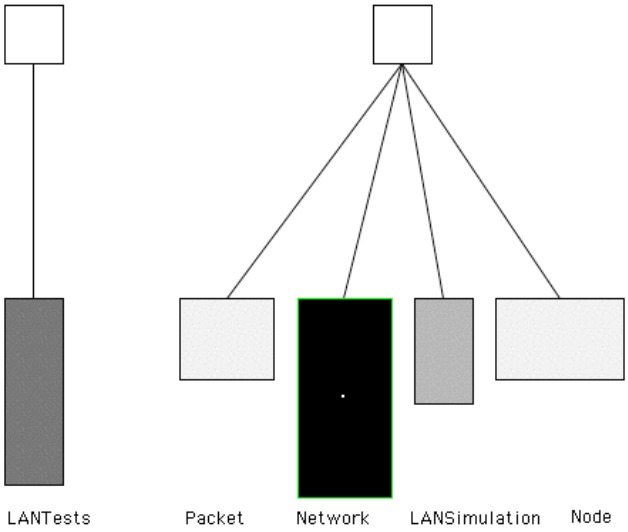

Figure 3. CodeCrawler system complexity view, illustrating that the class NETWORK — the tall black rectangle in the middle of the picture— shows the symptoms of a god class.

This view illustrates the procedural style of the 1.4 release; almost no inheritance and a single class containing most of the code — NETWORK, which is the tall black rectangle in the middle. The second tallest class is the one to the left in the diagram — it contains all unit tests and is therefore less problematic. The two light rectangles are PACKET and NODE; there square shape indicates that there are approximately as many attributes as methods, which suggests that they are mere data holders. Thus, we may conclude that

the CodeCrawler visualization indeed detects the expected problematic class. Nevertheless, one of the advantages of the LAN-simulation shows up as a major weakness here: it is too small. Indeed, the difference in size between the god class and the data holders is small, thus the god class does not look as bad as it actually is.

4.1.3 Insider

Finally, we applied a tool specifically designed to detect design flaws in object oriented systems, named INSIDER[13]. The tool allows us to automatically apply *detection strategies*[12], metrics-based rules for detecting design fragments (e.g., classes, methods, hierarchies) which are likely affected by design problems, and which consequently might require refactorings. For instance, the tool is able to recognize a "Brain Method" (a method that is long, with many branches, deep nesting level, and using many variables), a "Data Class" (a class which exposes its attributes and defines almost no functionality) and "Intraclass Duplication" (the duplication between methods of the same class). We first applied INSIDER on version 1.4 to see whether the tool actually detects the design flaws planted in the code, and then applied it on version 1.8 to see whether the tool confirms that the design has been improved.

On Version 1.4, the tool rightfully detected one "Brain Method" (PRINTDOCUMENT of class NETWORK), two "Data Classes" (NODE and PACKET) and three cases of "Intraclass Duplication" involving 5 methods. Thus, the tool confirms that NETWORK is centralizing a lot of intelligence (especially the PRINTDOCUMENT method), while NODE and PACKET are simple data providers, with reduced functionality.

On version 1.8 , the tool reports that the "Brain Method", the two "Data Classes" and all of the duplication from class NETWORK has disappeared and that no new design flaws were introduced. The only aspect that remained untouched in version 1.8 is the duplication inside the method SIMULATE. By comparing the measurements of the classes NETWORK, NODE and PACKET the tool also confirmed that there was a transfer of intelligence from NETWORK to PACKET and NODE and the two subclasses of NODE.

4.2 Curative

4.2.1 Eclipse

To execute the sequence of refactoring steps we relied on the refactoring engine integrated into the Eclipse Integrated Development environment. We used version 3.0 and learned that most of the refactorings work smoothly, yet noticed some small drawbacks with the execution of some of them. Also, the whole refactoring process itself involves some manual recoding, which stresses again the need to also rely on regression tests for demonstrating the preservation of behavior.

4.2.2 CVS

CVS was used to store the 9 releases of the system. It was more than adequate for this refactoring scenario, but this is not a surprise due to the small scale of the system. The main observation to make here was that CVS has the extra bonus of being well integrated with other tools, but other versioning systems could serve here just as well. See HTTP://BETTER-SCM.BERLIOS.DE/ for an overview of recent version control systems.

4.2.3 JUnit

A refactoring tool must verify the preconditions of a refactoring transformation to avoid the introduction of errors. However, programming languages typically have ways to circumvent normal compiler verification —for instance, the reflection mechanism in java— so no refactoring tool can provide an absolute guarantee. Moreover, most refactorings are combined with manual code transformations, which risks to introduce errors as well. Therefore, regression tests are necessary to verify whether the refactorings did preserve the behavior.

Using JUnit has the obvious advantage of being well-integrated with the Eclipse IDE. While refactoring the LAN-simulation, the regression tests where able to catch a few minor mistakes during the manual coding part. However to teach the importance of the quality of the tests, they were deliberately designed poorly, i.e. as simple input-output tests verifying whether the generated report is exactly the same as a previously generated and manually verified "correct" report. This is a cheap and reliable way to cover all use cases (or all code for that matter). Unfortunately, if a test fails this only shows up as a difference in output from one test-run to another, which makes it very inefficient to locate the source of the error.

4.3 Retrospective

4.3.1 CVS Change Scatterplot

To get a first overview of which files received most changes, we use the scatterplot described in [19]. In this plot, the horizontal axis shows all files and the vertical axis represents time. Each time a file is changed when committed on a particular time, a dot is plotted. Figure 4 shows that NETWORK is indeed changed in every release; combined with the fact that it is probably a god class this makes it a prime refactoring candidate. Note that the visualization does not show the distinction between adding functionality (the first 5 horizontal lines; releases 1.0 — 1.4) and the refactoring

steps (the next 4 horizontal lines; releases 1.5 - 1.8. Here as well, the smallness of the case (only 8 releases) becomes a weakness as the scatterplot is really aimed for large systems where many files have been committed over a long period of time.

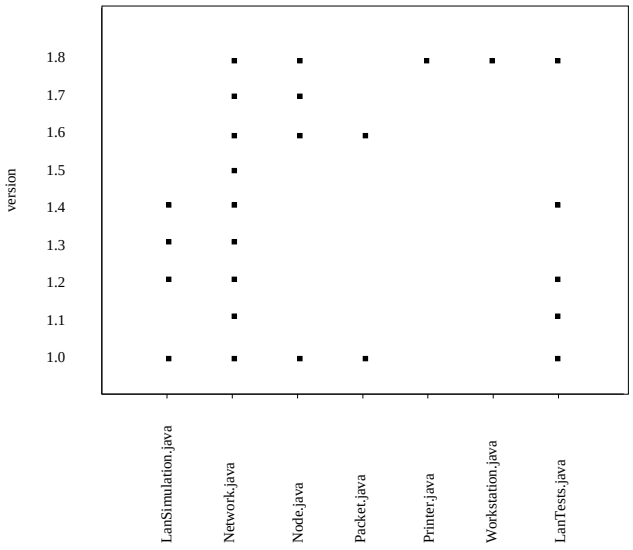

Figure 4. CVS Scatterplot, showing that the class NETWORK is changed in every release, i.e. the second vertical line shows a change (a dot) for every commit.

4.3.2 Class Evolution Matrix

The class evolution matrix shows a complete overview of all the releases and all the classes of a system. Every column in the matrix represents a release of a system, and every row shows the evolution of one class. Classes are shown as little rectangles where the width corresponds to the number of methods, the height to the number of attributes and the color to the number of lines of code.

In Figure 5, we see that NETWORK and LANTESTS are the largest classes. However, NETWORK becomes smaller in the last three releases (the color becomes lighter, thus the number of lines of code shrinks) and classes Packet and Node grow correspondingly. This suggests that functionality is moved from NETWORK class onto the others, which is indeed what happened. The visualisation also shows that in the very last release two classes are added. Note that this visualisation introduces one class PRECONDITIONVIOLATIONTESTCASE which does not appear in some other reports (most noteworthy Figure 3 and Figure 7). This is caused by the fact that PreconditionViolationTestCase is a nested class, and not all tools are able to represent them.

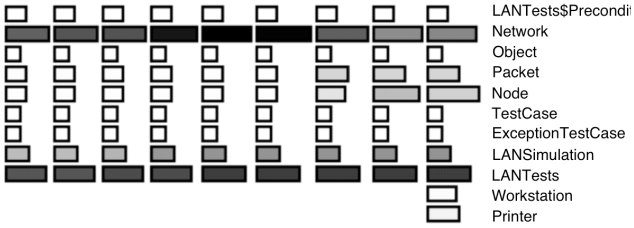

Figure 5. The class evolution matrix confirms that the class NETWORK starts of as a god class (i.e. wide dark rectangle in the first column), grows to reach a maximum size in release 1.4 (completely black in the fifth column) but shrinks towards the end (i.e. light gray in the last column).

4.3.3 Hierarchy Evolution

The Hierarchy Evolution Complexity View [8] is inspired by the System Complexity View from Figure 3 but now emphasizes the evolution. Thus, the width of a node shows the evolution in number of methods, i.e. the sum of the absolute differences between the number of methods in each release. Similarly, the height of a node represents the evolution in number of statements. The color of the node represents the age of the class. Both the thickness and the color of the inheritance relationship indicates how old and how often this has been changed. A removed class or inheritance would appear in cyan.

Figure 6 shows that all classes except WORKSTATION and PRINTER are in the system from the beginning (because they appear black). It also shows that the inheritance hierarchy is almost left unchanged during the refactoring steps and only the classes WORKSTATION and PRINTER are added in a later release (because both the classes and the inheritances appear light gray). Furthermore, the class NETWORK appears as being tall. This reveals that in that class many statements were added or removed in its evolution. This view shows that the NETWORK class suffered the largest overall changes. Combined with the Evolution Matrix we can see that first, the code was added and then removed from the NETWORK class.

4.3.4 EvoLens

The next technique called EvoLens [18] describes how classes evolved together. In Figure 7 the Evolution through all nine releases is visualized. The nested graph describes the containment hierarchy of the entire case study. Within the rectangles describing packages classes are depicted as ellipses. The color of the classes indicates their growth met-

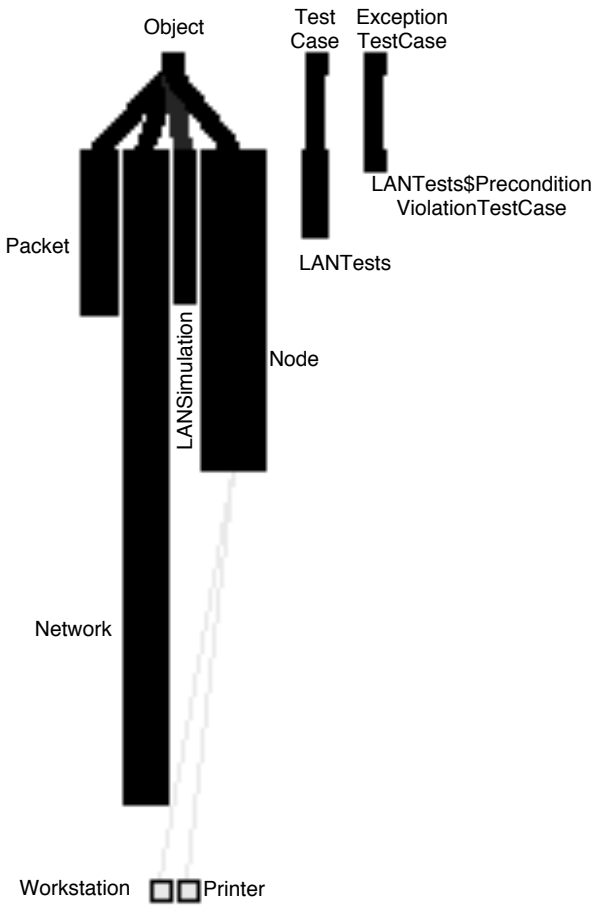

Figure 6. The hierarchy evolution shows that we deal with a very stable inheritance hierarchy, where only WORKSTATION and PRINTER are introduced as new subclasses.

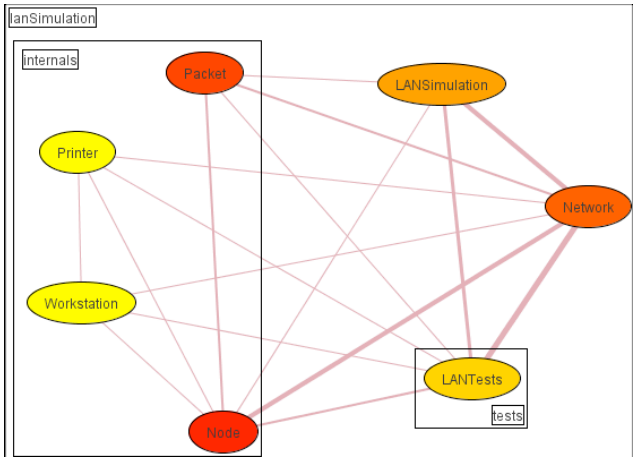

Figure 7. EvoLens shows that Packet and Node typically grow together (indeed, they are the main targets of behaviour removed from the god class) and that LANTests and LANSimulation change together but do not grow in size.

rics. Light yellow represents low growth, measured based on lines of code added within the releases, while dark red represents high growth values.

Figure 7 demonstrates that PRINTER and WORKSTATION where once introduced into the system, but where not enriched with much functionality. Also worthwhile to report is that during the development in release 1.0 through 1.4 tests were changed heavily together with NETWORK, but afterwards not changed very much. This confirms the test design based on input-output tests. After the development and the refactoring phases NETWORK remains quite large, which is indicated through the overall high to moderate growth value. Edges connecting ellipses within the nested graph shown in Figure 7 describe common changes to the classes. The striking outlier is NETWORK, which is changed in half of the releases together with LANSIMULATION, LANTESTS, and NODE. The mitigation of the god class is an important step in the design of the case study. Although NODE and PACKET have the highest growth values within the example, only NODE had to be often changed together with NETWORK. The refactoring seemed to be more focused on these two classes. The movement of the functionality that had to be done from NETWORK to PACKET could be fulfilled within a small number of releases. The almost complete graph of couplings is not a good sign for the structure of the LANSIMULATION. Every class of the example was changed together with almost every other class. The ideal of separating functional units with clean interfaces seems to be missed. However, reliable statements are difficult because of the small size of the case study. When inspecting large industrial case studies the threshold for co-change couplings is set to a higher level to obtain only strong couplings that influence the evolution of the entire system.

4.4 Lessons Learned

A first observation that can be made is that all tools were applicable in the context of this refactoring scenario. That is, (a) the predictive tools have been able to detect the code-smells that lead to the identification of the god class, (b) the curative tools have been able to support the stepwise refactoring process, (c) the retrospective tools have been able to report where, how and to some degree why the LANsimulation has been refactored. Obviously, some tools did

not perform that well, but this was mainly due to the small scale of the LAN-simulation — some tools have difficulties to scale down.

Secondly, the tools are indeed complementary. For instance, the predictive tools identified some common code-smells, but none of the tools discovered all of the code-smells. Similarly, the retrospective tools have all been able to analyze the release history, but all offered different insights that were all equally valid.

Thirdly, —and most importantly— the experiment showed the benefits for research groups to work on common examples. By comparing the results obtained by the tools, each participating research group gained a better understanding of the strengths and weaknesses. Moreover, discussing the refactoring scenario itself, we also gained a better understanding of what is a typical refactoring scenario. Of course, such results can only be obtained when the necessary resources are in place. We have obtained funding from the European Science Foundation to create a research network and used it to organize numerous workshops and research visits between the participating labs over the past three years. The results are encouraging, however to make real progress we should continue to convince funding agencies to support this kind of research.

5 Conclusion

We proposed the LAN-simulation as an example of a typical refactoring scenario that mimics realistic circumstances. The example has been used in several lab sessions within different universities and students have responded positively about this approach to teach the do's and don'ts of refactoring. In this paper, we went one step further and illustrated the applicability of the example by applying various tools supporting different aspects of the refactoring process. Thus, (a) we searched for code-smells (i.e. symptoms of poorly designed code that could be refactored), (b) we refactored the problematic code, and (c) we studied the release history. We learned that the tools under study do support parts of the refactoring process, yet that none of the tools is able to cover the whole. Thus, the example is capable of demonstrating how tools may complement one another. Based on this experiment, we conclude that the LAN-simulation is a good example to demonstrate where a certain tool may be used in the refactoring process. As such the LAN-simulation is ideally suited to achieve a better understanding between different research groups, and —more importantly— to serve as a common example for teaching students the necessary skills that are needed when refactoring.

Note that it is accepted practice for researchers to use small "exemplars" to illustrate a particular aspect of a certain technique. Such an exemplar must be small, otherwise it can't fit in a single research paper, yet is intended to be representative for real-world tasks. Requirement Engineering as a research field for instance frequently makes use of such exemplars and Feather et al. argue that their primary purposes are "(a) advancing a single research effort (that of an individual, or single research group); (b) promoting research and understanding among multiple researchers or research groups; (c) contributing to the advancement of software development practices" [6]. The LAN-simulation described here started out as a an effort to advance a single research effort (see [3, 14]), but now has achieved the status of being used within several research groups. As such, the LAN-simulation may serve as a first step in the necessary community building that is required to establish a common benchmark [20, 4]. Therefore, we encourage teachers and researchers all over the world to consider the LAN-simulations as a common example. Not only will it help us to demonstrate the subtle craft of evolving existing software; in the long run it will also help us understand and improve our research.

References

[1] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[3] S. Demeyer, D. Janssens, and T. Mens. Simulation of a LAN. *Electronic Notes in Theoretical Computer Science*, 72(4), 2002.

[4] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In T. Tamai, M. Aoyama, and K. Bennett, editors, *Proceedings IWPSE'2001 (4th International Workshop on Principles of Software Evolution)*, pages 147–177. ACM Press, Sept. 2001.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, Sept. 1999.

[6] M. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde. Requirements and specification exemplars. *Automated Software Engineering*, 4(4), 1997.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[8] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, pages 2–11, 2005.

[9] R. L. Glass. Maintenance: Less is not more. *IEEE Software*, July/August 1989.

[10] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[11] M. Lanza and S. Ducasse. Codecrawler - an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005.

[12] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, "Politehnica" University of Timişoara, Oct. 2002.

[13] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 350–359. IEEE Computer Society Press, 2004.

[14] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings ICGT2002 (First International Conference on Graph Transformation)*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. [Acceptance ratio: 26/45 = 57%].

[15] T. Mens and T. Tourwé. A survey of software refactoring. *Transactions on Software Engineering*, 30(2), 2004.

[16] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] W. Opdyke and R. Johnson. Creating abstract superclasses by refactoring. In *Proc. ACM Computer Science Conference*, pages 66–73. ACM Press, 1993.

[18] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. Technical Report TUV-1841-2004-26, Vienna University of Technology, December 2004.

[19] F. V. Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings ICSM'04 (International Conference on Software Maintenance)*, pages 328–337. IEEE Press, 2004.

[20] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 74–83, 2003.

[21] P. P. C. I. C. Society. *Guide to the Software Engineering Body of Knowledge — 2004 version*. IEEE Computer Society, 2003.

Acknowledgements

This work has been sponsored by the European Science Foundation by means of the project "Research Links to Explore and Advance Software Evolution (RELEASE)" and by the European Union via the Research Training Network "SegraVis". Other sponsoring was provided by the Belgian National Fund for Scientific Research (FWO) under grants "Foundations of Software Evolution" and "A Formal Foundation for Software Refactoring".