Towards a Proper Integration of Large Refactorings in Agile Software Development

Martin Lippert

University of Hamburg, Software Engineering Group & it-wps GmbH
Vogt-Kölln-Straße 30
22527 Hamburg, Germany
lippert@acm.org

Abstract. Refactoring is a key element of many agile software development methods. While most developers associate small design changes with the term refactoring (as described by Martin Fowler and William F. Opdyke), everyday development practice in medium- to large-sized projects calls for more than fine-grained refactorings. Such projects involve more complex refactorings, running for several hours or days and sometimes consisting of a huge number of steps. This paper discusses the problems posed by large refactorings and presents an approach that allows agile teams to integrate large refactorings into their daily work.

1 Introduction

Refactoring is part of everyday programming practice in agile software development[1]. The use of small-scale refactorings such as *Rename Method* or *Extract Interface* is well understood (see [6], [12]), many of them now being directly supported and automated by an Integrated Development Environment (IDE).

Of greater complexity are refactorings that introduce or remove pattern-like structures into a software system. The Refactoring to Patterns catalogue by Joshua Kerievsky provides an overview and handbook for some of the GoF patterns in [9]. Alur, Crupi and Malks describe J2EE-oriented pattern refactorings in [4]. Initial prototypes for automating these refactorings using specialized tools have appeared within the research community (see [3, 16]).

This paper focuses on refactorings that go beyond these small or pattern-based refactorings. In medium- to large-scale projects, we sometimes have refactorings that cannot be realized by means of a few renames, etc. For example, a refactoring that restructures the central inheritance hierarchy of a non-small system might affect several hundred or several thousand references to these classes. Such a refactoring could easily take several days or weeks, maybe even months to complete.

[1] This paper focuses on agile software development. Refactoring may also be part of any other development method.

While the scope and complexity of these refactorings is highly diverse, the term "large refactoring" has different connotations to different people. We therefore define some basic terminology before going on to introduce large refactorings and discuss them in more detail.

1.1 Integration Steps

An important concept in agile software development projects, especially when using Extreme Programming (see [1], [11]), is the idea of continuous integration. This means that changes and improvements to the system are realized in small steps. This paper subsumes all the changes made by one programmer (or pair of programmers) between two integrations under the term *Integration Step*.

Integration steps are not allowed to take more than one day each, the guideline for many agile development methods being to integrate by the end of the day (or throw the code away). Their duration thus ranges from minutes to hours. Each integration step must result in a properly running system and is integrated into the team's common code base.

Consequently, every task, requirement, feature or user story must be realized by proceeding in integration steps. Everything has to be done within this framework.

1.2 Small Refactorings

Many refactorings described in [6] can be realized within a single integration step. Such refactorings are called *Small Refactorings* in this paper. Examples are *Rename Class* (with a proper IDE) or *Extract Method*.

2 Large Refactorings

Some design changes and improvements cannot be realized completely within a single integration step[2]. Kent Beck and Martin Fowler describe this in their chapter on *Big Refactorings* in [6]. To fit these refactorings into the general concept of integration steps, they have to be split into smaller chunks. This is already a common task for user stories within Extreme Programming. The same is necessary for big refactorings to enable them to be handled within an agile development project[3].

It is quite difficult to decide why and when refactoring is big rather than small. Basing this purely on the number of changes to the system seems inappropriate. Modern IDEs offer automated refactoring support allowing several hundred places in the

[2] The reasons why these larger design changes occur even in the presence of merciless refactorings are not analyzed in detail in this paper.

[3] We do not discuss the possibility of realizing large refactorings in a separate branch of the system because the paper's focus is on integrating large refactorings into everyday development practice.

code to be changed in a few seconds. While the impact on the team is most significant in a big refactoring, it is becoming more and more apparent that big refactorings are best characterized by the time the team takes to complete them. The term *Large Refactoring* is thus defined to reflect this.

Definition:
Large Refactorings are refactorings that cannot be realized within a single integration step.

This definition of large refactorings includes refactorings that span only two or three days. It might be an exaggeration to call them "large" refactorings. The real focus of this paper is on refactorings that take weeks or months rather than a few days to complete. Nevertheless, many of the problems we have observed with three-month refactorings (see below) also occur with refactorings that span only a few days – only on a much smaller scale. Thus ideas on how to deal with these problems are just as applicable to two- or three-day as they are to three-month refactorings. Of course, the proposed approach becomes more important, the more time the refactoring takes.

2.1 Why Are Large Refactorings More Problematic Than Small Ones?

Large refactorings differ from small ones not only in terms of their size or the time they take. Beck and Fowler emphasize in [6], for their big refactorings, that it is of crucial importance that all members of the team are aware of the big refactoring, that they know where it is going and how it affects their daily work. This is important because large refactorings have to be split into a number of steps (as discussed above). Each step of the large refactoring is realized and integrated into the common source-code repository of the system.

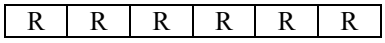

Figure 1: A large refactoring split into small steps. Each R describes a refactoring step. Time runs from left to right.

If integrated into the general development process, this is done parallel to other developers of the team working on the system. The complete integration flow of the team may look like this:

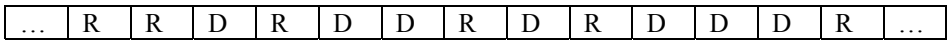

Figure 2: A complete integration flow. The steps for the large refactoring and the normal development (D) are interlocked.

This situation can cause a number of difficulties, especially if the large refactoring is complicated and runs for several weeks or months. Frequently observed problems are:

- **Interim states of large refactorings:** Interim states of large refactorings become visible to the team. This means that all developers may be confronted with changes made to the common code base as a result of the large refactoring. In this case, the system typically contains code parts that follow the new structure as well as code parts that are not yet adapted to it. By-passes in the code are often used to make this possible. Such situations – dealing with new and old parts of the code's structure – can confuse developers who are not familiar with the details of the refactoring. In addition, it is hard for the developers to keep track of all the by-passes and different code states.
- **Teams get lost:** Sometimes teams get lost in large refactorings. This often happens because the team has to implement a large number of changes over a lengthy period of time. After several weeks of doing the refactoring alongside the daily feature development, and faced with hundreds of changes, a huge number of deprecated methods and different parts of the system following different designs, individual team developers may get confused. Sometimes they even end up forgetting the main goal of the refactoring, resulting in an unfinished refactoring.
- **Unfinished refactorings:** One risk with large refactorings is that they never get finished. Developers simply forget to finish the refactoring completely, perhaps because major parts of the refactoring are finished or other things distract them. This mostly results in code-structure flaws. Parts of the system conform to the new structure, while other parts follow the old one. This situation can even result in a code structure that is, overall, worse than before the refactoring.
- **More complex planning:** A large refactoring is much more difficult to plan and predict than small refactorings. While a team is doing a large refactoring, the rest of the system changes, too. Team members implement new features or do small refactorings at the same time that other team members are working on the large refactoring. Changes to the system can have an impact on future large-refactoring steps.

Another important planning issue with large refactorings is that they need to be integrated somehow into the release and/or iteration planning. This is necessary to reserve development time for the refactoring and to concentrate the work on such bigger design changes.

2.2 Consequences

Faced with the challenge of more complex design changes, many projects opt for one of the following alternatives:

- They avoid more complex changes to the structure and make do with a bad system design.
- They stop normal system development to concentrate exclusively on the large refactoring.

Since both alternatives appear unsuitable in agile software development, this paper analyzes in more detail the issues surrounding large refactorings. The goal is to work out a way of dealing with large refactorings so as to make them manageable in the daily development practice of agile projects.

3 Explicit Refactoring Routes

As described earlier, a large refactoring has to be split into a number of smaller steps. These steps are not chosen randomly. They describe a route from the current to the desired design. This route is called a *Refactoring Route*. Its key features are:

- A refactoring route subsumes a number of steps that lead from the current to the desired design.
- Each step should be realizable within one or more integration steps.

Following the definition of integration step (see Section 1.1), this means that a large refactoring has to be split into a number of steps, where

- each step results in a running system
- each step can be realized in a maximum of one day

This relates directly to the mechanics sections for each refactoring in [6], especially for big refactorings. But such sections are written generically, e.g.: "Decide which job is more important and is to be retained in the current hierarchy and which is to be moved to another hierarchy" from the *Tease Apart Inheritance* refactoring ([6], pp. 362ff). With a concrete large refactoring, the refactoring route could be described in a much more concrete and meaningful way for the team using the concrete class names and concrete concerns of the system.

In Extreme Programming projects, the individual steps for a large refactoring can be written on separate task cards – enhanced by an overall card describing the large refactoring as a whole. But experience with large refactorings has shown that this is often not enough. The above-mentioned problems still remain.

3.1 A Refactoring Plan

This paper proposes enhancing refactoring mechanics and tasks cards for large refactorings. Key to this is the concept of an explicit refactoring route written in the form of a *Refactoring Plan*.

A refactoring plan consists of a sequence of *Refactoring Steps*. A refactoring step is of the same scope as one or multiple integration steps. An example of a refactoring step is: "Analyze all usages of class A and shift them to usages of class B, where possible". Depending on the size of the project, a refactoring step may have to be split into multiple integration steps or can be done within a single integration step.

The entries of a refactoring plan reflect the concrete system and the route that makes sense for the large refactoring in the concrete situation. The team thus arranges the refactoring steps in the order in which they are to be realized.

To track the progress of the large refactoring, each step of a refactoring plan can be marked as finished, work-in-progress or open. The steps of a refactoring plan can be rearranged, deleted or adapted, if necessary[4].

3.2 Refactoring Plans in Practice

Refactoring plans serve two different purposes. On the one hand, the team can use refactoring plans to discuss, rethink or replan large refactorings. They are thus vital elements in the development process. On the other, they allow developers to keep an eye on the refactoring while developing new features, thus serving as a map and a reminder.

Typically, a refactoring plan for a concrete large refactoring is drawn up by the team while discussing what refactoring needs to be done. The plan is initially sketched out on a sheet of flipchart paper and pinned on the wall to make it visible to the whole team.

When the team is working on the refactoring, they usually pick the next open step from the refactoring plan and mark that step as work-in-progress on the paper. Once they finish the refactoring step, they mark it as finished.

It sometimes happens that the steps in the refactoring plan have to be replaced or rearranged. In this case, the team or pair doing the refactoring discuss the changes. As a result, a changed refactoring plan is communicated to the team in the same way the old refactoring plan was.

3.3 Forms of Refactoring Plans

Refactoring plans can take different forms and be at different stages of expansion. Three possible variants are:

- **The Manual Refactoring Plan:** One way of dealing with explicit refactoring plans is a simple, manual approach, using a handwritten plan on a flipchart or whiteboard visible to all members of the team. This is the simplest form of explicit refactoring plan, and one that has been successfully used by us in a project context.
- **The Electronic Refactoring Plan:** Greater potential for team support is offered by an electronic version of a refactoring plan that is part of the project source base. A simple and suitable tool can help to integrate refactoring plans into the IDE to make them directly and easily visible to all project members (e.g. via specialized views in the Eclipse Java Tooling, see [5]). The electronic version makes it easy to modify the plan and facilitates teamwork across different locations (a handwritten plan being more suitable for a single location).

[4] Examples of refactoring plans can be found at [10].

We have also used a wiki page to sketch out and track a refactoring plan. The downside of electronic refactoring plans is that they do not attract the same attention as a big poster-size plan on the wall.

- **Vision – the Connected Refactoring Plan:** In addition, electronic refactoring plans could be connected to the source code to allow navigation from finished refactoring steps to changed parts of the source code and vice versa. This is useful to find information on large refactorings, together with the changes they have introduced into the code. Developers can easily find out whether the large refactoring has affected the code they are going to work on.
- **Vision – the Refactoring Map:** To make it easier for developers to check whether their work is affected by a running large refactoring, the idea of a *Refactoring Map* emerged. A refactoring map displays the complete system in a map-like form. The parts of the system affected by changes due to the refactoring are marked (e.g. in a particular color). The developer can use the map to see at a glance if the large refactoring comes close to the part of the system he is working on.

3.4 Implications of Refactoring Plans

Refactoring plans can change the way developers deal with large refactorings in agile development projects. The anticipated benefits from using refactoring plans include:

- All developers of a team are aware of ongoing large refactorings and can observe the progress.
- Developers can easily see which large refactorings are not yet finished. This prevents the team from forgetting unfinished large refactorings.
- The team can track the progress of a refactoring. This can help to plan the refactoring effort required in current and future iterations.
- The risk of getting lost within a large refactoring is reduced by the refactoring plan. Developers can watch the plan while immersing themselves in the refactoring. They can check whether the current activity really yields a benefit for the overall refactoring or not.
- Developers can recognize changes and by-passes within the code that are introduced as part of a large refactoring (using the electronic version of a refactoring plan).

3.5 Consequences for Project Planning

The discussion of large refactorings reveals that agile development projects need to pay explicit attention to large refactoring tasks. While small refactorings are part of everyday programming practice – and thus not a separate project-planning issue – large refactorings need to be taken into account in the planning process. They must be scheduled somehow during iteration and release planning as they could easily take up a large part of an iteration's development time.

4 Related Work

In [13], Don Roberts and John Brant describe a tool designed to support mass changes to source code automatically. Basically, they took the source-code transformation engine of their Smalltalk Refactoring Browser (see [2]) and used it to automatically modify Smalltalk source code following a user-written script-like list of rules. This rule script is used by the transformation engine to modify the source code.

Unlike us, Roberts and Brant adopt an "all-at-once" approach, in which a large refactoring is basically prototyped using their rule engine. If the complete path through the refactoring is found, they execute the rule-based script for the refactoring in one step. Their approach completely ignores the communication issues of an agile team. The team's developers have to live with situations in which many lines of code change from one day to the next. In addition, the approach of working on a fixed version of the system to do the refactoring (or writing the rewriting rules) involves similar risks to doing the refactoring in a separate branch (merging, major changes to the head version, etc.). Another drawback of their approach is that writing rules on top of parse trees can be quite complicated for developers not used to thinking in terms of parse trees (see [13]).

Nevertheless, using a rewrite engine like the one they propose to realize parts of large refactorings is a conceivable solution. It would be most powerful for refactoring steps with simple transformations but a high number of dependencies on these changes.

Tammo Freese has proposed a way of using *Inline Method* refactoring to facilitate API changes within an application (see [7]). His work demonstrates an elegant way to split API interface changes into smaller steps. This technique could be used to split large refactorings into smaller steps.

In [8], Tammo Freese describes an approach designed to facilitate what he calls global refactorings within agile development teams. The basic goal of his work, with regard to the topic of this paper, is to facilitate automatic refactorings that affect large parts of the system. He proposes a specialized version-management system that is aware of refactorings and is therefore able to merge refactoring results automatically. This approach could be quite useful for developers dealing with large refactorings. While this paper focuses on a different issue, namely how to integrate large refactorings into the daily work of an agile team, individual steps of a large refactoring could be supported by a refactoring-aware version-management system.

The concept of a refactoring plan is derived from the work on process patterns for situated action (see [14], [15]). The authors use process patterns to reify typical work processes in application domains. Their process patterns replace workflow systems with a more flexible way to describe common processes and deal with them individually. Unlike the process patterns, refactoring plans are written for a concrete refactoring only. They cannot be reused for similar refactorings and they do not serve as a template for multiple refactorings.

5 Conclusion

This paper introduces the notion of large refactorings and emphasizes that they are an important issue in today's agile software development methods. The main problems and characteristics of large refactorings are presented and briefly discussed. The paper focuses on the team issues posed when dealing with large refactorings, in contrast to a formal approach designed to somehow automate large refactorings. The focus, then, is on the problems faced by agile teams when dealing with large design changes.

The concept of explicit refactoring plans is presented, which are designed to integrate large refactorings into the daily programming work of an agile software development team. These plans combine the notions of situated process patterns and task planning to create a simple and easy-to-use concept. They aim to help teams manage large refactorings smoothly within an agile development project.

While electronic refactoring plans have yet to be implemented, initial experience with manual refactoring plans has been gained and shows promise. Nevertheless, what the paper presents is more a concept for supporting teams dealing with large refactorings than a proven solution. Further research is needed to verify the suitability of the presented approach in a larger number of projects.

Acknowledgments

My thanks go to Axel Schmolitzky, Holger Breitling and Marko Schulz for their comments on draft versions of this paper, and to the other members of the Software Engineering Group at the University of Hamburg for their comments and discussions on the topic in general. I would also like to thank Stefan Roock for his work and feedback on the topic.

I am particularly indebted to the following participants of the OT 2003 *Workshop on Large Refactorings*: Peter Marks, Erik Groeneveld, Peter Hammond, Alan Francis, Ray Farmer, Pascal Van Cauwenberghe, Peter Schrier, Marc Evers, Willem-Jan van den Ende and Matt Stephenson, as well as to the participants of the OOPSLA 2003 *Workshop on Beyond Greenfield Development*, especially to Kyle Brown for his feedback. My very special thanks go to Brian Barry for his comments and the idea of refactoring maps.

References

1. Beck, K.: *Extreme Programming Explained – Embrace Change*, Addison-Wesley (2001)
2. Brant, J., Roberts, D.: *Smalltalk Refactoring Browser*. http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser.
3. Cinnéide, M. Ó.: *Automated Refactoring to Introduce Design Patterns*, Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000)

4. Crupi, J., Alur, D., Malks, D.: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall PTR (2001)
5. Eclipse Project: http://www.eclipse.org
6. Fowler, M.: *Refactoring – Improving the Design of Existing Code*, Addison-Wesley (1999)
7. Freese, T.: *Inline Method Considered Helpful: An Approach to Interface Evolution*, in: Extreme Programming and Agile Processes in Software Engineering, Proceedings of the 4th International Conference XP 2003, Genova, Italy, LNCS 2675, Springer (2003), 271-278
8. Freese, T.: *Software Configuration Management for Test-Driven Development*, in: Extreme Programming and Agile Processes in Software Engineering, Proceedings of the 4th International Conference XP 2003, Genova, Italy, LNCS 2675, Springer (2003), 431-432
9. Kerievsky, J.: *Refactoring to Patterns*, Addison Wesley (2004)
10. Lippert, M.: *Refactoring-Plans – Examples and Experiences*, http://www.martinlippert.com
11. Lippert, M., Roock, S., Wolf, H.: Extreme Programming in Action – Experiences from Real-World Projects, Wiley & Sons (2002)
12. Opdyke, W. F.: *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science (1992) Tech. Report UIUCDCS-R-92-1759.
13. Roberts, D., Brant, J.: *Tools for Making Impossible Changes*, to be published in IEE Proceedings-Software, Dec. (2003)
14. Suchman, L.: *Plans and Situated Actions. The Problem of Human-Machine Communication*. Cambridge University Press (1987)
15. Wulf, M., Gryczan, G., Züllighoven, H.: *Process Patterns - Supporting Cooperative Work in the Tools & Materials Approach*, Information Systems Research Seminar In Scandinavia: IRIS 19; proceedings, Lökeberg, Sweden, 10-13 August, 1996. Bo Dahlbom et al. (eds.). - Gothenburg: Studies in Informatics, Report 8 (1996), pp. 445 – 460
16. Zannier, C., Maurer, F.: *Tool Support for Complex Refactoring to Design Patterns*, in: Extreme Programming and Agile Processes in Software Engineering, Proceedings of the 4th International Conference XP 2003, Genova, Italy (2003), LNCS 2675, Springer (2003), 123-130