Literature review

Refactoring in the Context of Programming Language Evolution

Svetlana Omelkova

November 18, 2012

Abstract

All successful programming languages evolve over time. They tend to become more complex, some features appear and some become outdated or rarely used. Old code becomes hard to maintain, since some of outdated features are unfamiliar to modern programmers. Moreover frameworks and libraries also tend to change APIs. Migrating the software to the new API could be tedious and slow down development process. Some of this problems could be solved with the help of refactoring tools that can transform programs to use the modern form. In this literature review we look closer to how actually language evolve, understand how language developers cope with this evolution and backwards compatibility problems. Separately we consider the number of researches devoted to refactoring tools and show that refactoring tools can ease the cost of program evolution.

1 Introduction

Refactoring is a process of making changes to source code that do not have a net effect on the program’s observed behavior[13]. The main goals are to improve performance, maintainability, readability and reduce the complexity of code. The simplest example is when one wants to rename variable or function, cut program to subprograms or convert array of structures to the structure of arrays.

The history of refactoring already consists of 20 years. It starts in 1991 with seminal thesis by Opdyke [13] and Griswold [8] and continue with well known book by Fowler[7] in 1999. Opdyke considered refactoring as changes of object-oriented programs on the intermediate level. Fowler believes that usually design comes first and then comes good coding, while refactoring is the opposite practice. One can starts with bad and chaotic design and later rework it into well-designed code.

In [14]Opdyke proposed several cases where refactoring might be applied:

- Extracting reusable component. For example, if customer of a system needs new components, but with the user interface, that is compatible with the older system.
• **Improving consistency between components.** If two modules of the system were initially considered as distinct, but later they have to share some common abstractions.

• **Supporting the iterative design of an Object-Oriented Application Framework.** To support abstract design of an application and ease design-level reuse.

Another important topic in refactoring is *Static program analysis*. Static program analysis is the analysis of program code without actually executing this code. This term is usually applied to the analysis performed by automated tools, while human analysis called *program understanding* or *code review*. Cousots in [3] distinguished 4 basic methods for compositional separate modular static analysis of programs by abstract interpretation: simplification-based separate analysis; worst-case separate analysis; separate analysis with user-provided interfaces; symbolic relational separate analysis. This modular approach allows to analyze huge programs (up to 40 millions lines of code).

But in this review we take a closer look to the problem of programming language evolution. Successful programming languages tend to evolve as they age. Thus some features become outdated and rarely used. The main inconvenience is that some outdated features are not removed to maintain backward compatibility, thus the language becomes increasingly complex over time. Overbey and Johnson in [15] propose to use automated refactoring tools to eliminate old constructions and idioms from source code. They propose the idea that refactoring has to be an essential part of language design and refactoring tools has to be an essential part of language implementation along with compilers.

This document will describe problems caused by language evolution. In the second section we examine language evolution problems and refactoring solutions concerned with it. In the third section we take a look to the program components evolution. In fourth part we consider some refactoring tools and number of researches made to improve accuracy and reduce effort to maintain software. And finally, conclusions are in section five.

## 2 Language refactoring

Programming languages constantly evolve. Programmers can concentrate only on a new features and ignore outdated parts of the code. But the problem arise when the programmer is forced to maintain other’s code. He needs to be aware of both “old” and “new” ways of accomplish the same task. Usually authors just do not know that he using a construct for which a better alternative exists.

Backwards compatibility means that language could be extended but nothing could be removed. Therefore language becomes extremely complex overtime.

Authors in [15] propose refactoring tools to become as essential part of language implementation as compilers are now. When language evolve and new constructs and idioms are introduced, it should be accompanied by automated refactoring tools that replace all outdated constructs. This would allow programming languages to completely eliminate outdated features and idioms. Also old constructs can be eliminated from the compiler,
static analysis tools, IDEs, etc. It benefits the end user of the language not to learn outdated dialects (and possibly helping experienced programmers to un-learn them).

How language evolve?

Overbey and Johnson in [15] examine language evolution on the example of language changes between Java 1 to Java 6. In some of the cases user need to indicate which pattern to match, in others refactoring couldn’t be automatically applied. Examples of language evolution are below:

- In J2SE 5.0 were introduced generics (parametric polymorphism). Very good research in refactoring has been done by Kiezun et al. in [1]. They presented a type constraint-based algorithm for converting non-generic libraries to add type parameters.

- Enumeration. Programmers tend to use integer constants to mimic a C enum, useful refactoring would need to allow the user to convert these to Java 5 enumerations. This is often used in tandem with refactorings like Replace Conditional with Polymorphism[7]. The choice between classes, constants, and enumerations is a design decision.

- Autoboxing. In Java Language Specification[10] defined a context in which primitive values are boxed and unboxed automatically. Since there are only two ways to box each primitive (for int values, these are the Integer class constructor and Integer#valueOf(int)) and one to unbox (Integer#intValue()), the refactoring involves matching these patterns and determining whether or not they occur in a context where the primitive would be auto-boxed/unboxed.

- Static import. Java 5’s import static allows static members of another class to be accessed without qualifiers. A canonical use is to reference constants like Math.PI or methods like Math.cos() as simply PI and cos(). Sun recommends that this feature be used “very sparingly,” when the programmer would otherwise “be tempted to declare local copies of constants, or to abuse inheritance”. An automated refactoring to remove static qualifiers would similarly need to be applied sparingly.

- Force strict floating point computations. J2SE 1.2 changed the semantics of floating point computation and the strictfp keyword was introduced to force the JVM to use old floating point behavior. Completely preserving the semantics of a Java 1.1 program on a J2SE 1.2 JVM would require an abundance of strictfp modifiers in the code.

- Assertions. There is no single idiom that assertions intend to replace. Prior to their introduction, each system had a proprietary idiom for making assertions. Thus, this is the case where syntactic find-and-replace is the best option.
• Annotations. There is no widespread idioms that annotations were intended to replace, since it is fundamentally new addition to the Java language, that provide the facility that was not previously available.

Some languages, such as Fortran and Python do not support backwards compatibility. Python developers elaborated the basic policy of a backwards compatibility [16]:

• Unless it is going through the deprecation process below, the behavior of an API must not change between any two consecutive releases.

• Similarly a feature cannot be removed without notice between any two consecutive releases.

• Addition of a feature which breaks 3rd party libraries or applications should have a large benefit to breakage ratio, and/or the incompatibility should be trivial to fix in broken code.

But actually only Python 3.0 in 2008 was the first ever intentionally backwards incompatible Python release. Fortunately, there is a tool 2to3[9] that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library lib2to3 is, however, a flexible and generic library, so it is possible to write own fixers for 2to3.

In Python Enhancement Proposal [16] author propose the following algorithm in case of making any incompatible changes:

• Every change should be discussed in advance within the community.

• Add a warning. Old usage should rise a warning. DeprecationWarning is the usual warning category to use, but PendingDeprecationWarning may be used in special cases were the old and new versions of the API will coexist for many releases.

• Wait for the new release and see if there is any feedback.

• The behavior change or feature removal may now be made default or permanent in the next release. Remove the old version and warning.

3 Refactoring libraries

Despite that traditionally refactoring has been applied to application, also refactoring can deal with libraries, frameworks and components. Usually maintaining the system means to update it to use the latest version of it’s components. Developers like to reuse components, because it save time and effort. Ideally the interface of the components should be immutable, but in practice it do not. Some studies are devoted to the question why it changes. For example, in [12] authors proposed 12 named types of software evolution that
recognized based on objective evidence. Also authors identified classification of activities for practitioners and their managers. But in our context we more interested in question how the software changes.

Dig and Johnson in [5] made a study where they looked at five case studies: one proprietary, three open source frameworks and one library. All five case studies are JAVA systems. They studied the differences between two major releases. They distinguish between two types of component changes: breaking and non-breaking. Breaking changes are not backwards compatible, while breaking changes are backwards compatible. Such a change could be an enhancement of functionality, like adding a new component. Authors found that about 80% of breaking changes could be expressed as structural, behavior-preserving transformations (refactoring). Most API changes occur as responsibility is shifted between classes (e.g., methods or fields moved around) and collaboration protocol changes (e.g., renaming or changing method signature). Authors state that refactoring plays important role in the evolution of components. Almost in all of those cases it is possible to provide script that would be able automatically migrate code to the newer version of the library. Authors promised to propose automated migration tool, which is based on the refactoring.

The last 20% of cases should be carried by application developers. But sometimes it is possible to provide library specific refactoring. For example, in [4] authors implemented refactoring tool, Concurrencer, that migrate Java applications to use thread-safe collection classes. This tool automates three refactorings: converts Integer field to AtomicInteger, converts HashMap to ConcurrentHashMap and parallelize divide-and-conquer algorithm. Authors are planning to extend their tool to support other features such as conversion to other thread-safe Atomic* and scalable Collection classes.

Almost at the same time Tansey and Tilevich [20] are propose refactoring that migrates codebases from the JUnit 3 unit testing framework to JUnit 4. Also this paper makes some important novel contributions. For instance, authors propose the notion of Annotation refactoring. Annotation refactoring is a new class of refactoring that replaces the type and naming requirements of an old framework version with the annotation requirements of a target framework.

4 Refactoring tools

Refactoring tools can improve accuracy and speed of creating and maintaining software. Refactoring tools automate refactoring that developer would perform with the editor. Many popular IDEs for variety of languages include refactoring tools, such as Eclipse, Microsoft Visual Studio, Xcode etc. Currently researchers are making a great effort to evaluate, improve and support refactoring tools. For example, Murphy-Hill and Black in [11] distinguish between floss and root canal refactoring. Floss refactoring authors compare with daily teeth cleaning, it is refactoring that developer perform all the time during the coding. But root canal refactoring is the infrequent, protracted period compared to the dentist visiting. Developer perform floss refactoring to maintain healthy code and perform
root canal refactoring to correct unhealthy code. Authors state that modern refactoring tools are mostly suitable for floss refactoring. Also they developed five principles that determine that tool enables floss refactoring, such as choose the desired refactoring quickly, navigate program code while using the tool etc. Those principles should help developers to choose a refactoring tool that suits their daily programming tasks.

Many effort was done to compare different refactoring tools. In [18] authors applied the Taxonomy of Software Evolution, developed by Mens et. al [19]. This taxonomy is based on the mechanisms of change and the factors that impact upon these mechanisms. The goal of this taxonomy is to position concrete tools and techniques within the domain of software evolution, so that it becomes easier to compare and combine them. Authors apply this taxonomy to four tools that provide explicit support for refactoring. Considered tools are the following: Smalltalk VisualWorks 7.0, Eclipse 2.0, Guru (for SELF 4.0) and the Together ControlCenter 6.0. After a detailed discussion and comparison of these tools, authors analyze the strengths, weaknesses and limitations of the evolution taxonomy that was used.

Another research on comparison and usability was done in [6]. Authors completed usability study for software refactoring tools. They reviewed 11 collections of usability guidelines using ISO 9241 - 11 standard, then composed single list of 38 guidelines. From this list they developed 81 usability requirements for refactoring tools. The refactoring tools studied are: Eclipse 3.2, Condenser 1.05, RefactorIT 2.5.1, and Eclipse 3.2 with the Simian UI 2.2.12. Authors evaluate that tools by refactoring a Java program that implements a library system. The system has 12 initial classes with 319 LOC, which is refactored into 15 classes with 250 lines of code. The results were summarized using Sheridan's levels of automation[17]. Authors conclude that non of four refactoring tools provided ideal level of automation. The overall purpose of this study was to identify areas in which tools usually do not fulfill usability requirements. Authors are planning to propose some tool, such as Eclipse plugin that will meet 21 requirements, which authors considered most valuable.

**Conclusion**

Accomplished review shows that refactoring is a viable technical strategy for language evolution. But why those tools are not widely used so far? Probably the reason is the number of technical challenges in such tool implementation and still those tools are quite buggy[2]. Moreover the problem could be also cultural: many programmers still have difficulty to trust the tool to rewrite significant part of their code. For refactoring to become a vehicle for language evolution, refactoring tools will need to be equally robust and trustworthy that compilers currently are[15].

Refactoring engines should guarantee that structural changes they perform will not brake the applications. Migration tool based on refactoring engine should be able to perform the biggest part of tedious job updating to a new version, only about 20% of changes should remains for developers. Some changes could require a human expertise.
Also the availability of powerful migration tools would change things for component users and designers as well. Designers won’t fear that their changes would break the client. With this new freedom they won’t have to carry bad design decisions made in the past. They will produce the design to be easier to understand and reuse.

References


