Straightening Spaghetti-Code with Refactoring?

Markus Pizka[*]
Technische Universität München, Institut für Informatik - I4
Boltzmannstr. 3, Germany - 85748 Garching
pizka@in.tum.de

Abstract *Changes to software systems often entail a loss of quality, especially if they have to be accomplished under pressure of time. Long-term software projects must counter this phenomenon one way or the other to preserve long-term maintainability. This paper presents the results of a case study trying to improve an extensive low-quality code base by object-oriented and tool-supported refactoring. To obtain practically relevant experiences this case study was conducted within an on-going commercial software project. The existing code base was first assessed using metrics as well as subjective judgment and later-on refactored according to the findings of the assessment. By this, we evaluated the practical applicability of several metrics and refactoring tools. The results of this experiment indicate that tool support is immature and the impact of refactoring is limited if the code base has gone astray for a longer period of time.*

Keywords: refactoring, software maintenance, software metrics

1 Code Decay

Though software does not wear out due to heavy use or corrosion like technical products do, the quality of software does often suffer from subsequent changes to the system. The process of gradual deterioration and the effects of accumulated changes are still weakly understood and lead to misunderstandings. While some customers tend to blame developers for their inability to accomplish "small" changes quick but with high quality, some programmers vice versa condemn users for not being able to specify stable requirements.

1.1 Reasons and Effects

As a matter of fact, both points of view must be considered unrealistic. Requirements change because business processes are constantly evolving [8]. Therefore the need for continuous change is inescapable. Frequent changes, on the other hand, reduce the quality of a software system in many cases independently of any negligence. This is because changes usually have to be accomplished under pressure of time and budget, which forces the developer to accept compromises. In addition to this, changes must be made with limited knowledge about the consequences for the overall system. Although program slicing [22] has made progress there is no applicable method to systematically determine the impact of a change on all different aspects [3] of a software system, yet.

As a consequence, the quality of a software system tends to decrease with increasing number of changes. While single changes do usually not have a strong negative impact, accumulated tiny changes together with their interplay may cause serious risks in the long run.

Lehman's work on software evolution [10, 11] provides extensive empirical evidence for the duality of change. Law I states that continuous change is necessary to preserve satisfaction and Law II says that the complexity[1] of the system increases with the number of changes if it is not reduced with additional work.

1.2 How to Deal With It

Although there is a widespread insight in the need of additional (re)work among practicioners surprisingly few research works discuss the important question how to deal with it in detail, i. e. the when, what and how of the additional work.

Techniques like object-orientation, software architecture, and information hiding help to defer the need for additional rework in certain situations but are not capable of eliminating it in general. What is needed is some kind of pro-active consolidation of the software system that is either done periodically or after certain thresholds are reached. Among the goals of this consolidation are restoring structural integrity and eliminating redundancy, obsolete components, and heterogeneity.

[*]Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project ViSEK (Virtual Software Engineering Competence Center).

[1]Note, that the term "complexity" is commonly used without further substantiation to describe the difficulties to understand or perform certain tasks on the system, which has a strong correlation with its quality.

Thus, consolidation is a time-consuming, error-prone and hazardous task. Well-defined processes, special techniques and tools are mandatory for the efficient accomplishment of this complicated task.

1.3 Consolidation by Refactoring

In this paper we present an experiment trying to perform consolidation by tool-supported object-oriented refactoring [15, 6, 7]. Refactoring suggests itself as a suitable technique because its intrinsic motivation is to improve the design of an existing code base.

While the motivation of refactoring corresponds well with our goal to consolidate[2], its intended use differs significantly. Refactoring is mostly used in agile development processes [5, 2, 1] where it is applied frequently whenever needed as part of the evolutionary development process. Here, we want to use refactoring to improve a code base that has gone astray for several man-years *without* any noticeable rework in between! A priori it is unclear whether refactoring is effective under these circumstances.

Therefore our case study investigates the following practically relevant questions in a realistic setting:

1. Are sequences of small refactoring steps suitable to perform coarse granular changes or what else is needed?
2. Is tool supported refactoring mature enough?

Outline Section 2 relates this case study with research work in the broader scope of software evolution and maintenance before we detail the goals, setup, and context of our case study in section 3. Section 4 presents the results of the initial analysis of the code base before the refactorings, presented in 5, were applied. Section 6 interprets the observations made before we summarize the results obtained in section 7.

2 Related Work

The motivation of the work presented in this paper stems less from agile methods or refactoring but from a software evolution [10] and software maintenance [17, 14] point of view. Our goals follow the vision of the US Department of Defense stated in the SRAH report [21] “preserve, extend and leverage . . . investements in systems through reengineering.”

Parnas’ article on “software aging” [16] describes two distinct sources of software decay. First, the failure to adapt software to changing needs and second, the result of changes made. Deterioration should be slowed down by introducing and recreating structure after changes as well as retroactive incremental modularisation and restructuring. The correctness of these statements about code decay are backed by empirical evidence such as in [4].

Though several studies further investigate the importance and driving forces of software maintenance, e. g. [12], there is not much work on how to practically counteract. In [23] several authors from industry and academia propose their attitude towards decay, its reasons, and the required sanctions ranging from improved quality of the development process over company organisation structure to programming languages and modularisation. [20] adds software restoration on three distinct different levels: code, software (including all documents), and system. The latter also regards the environment the software is embedded in.

Hence, the field of code decay and the possibilities to counteract is diverse and broad. Our case study is one small tessera of it, based on the shared beliefs that a) succesfull software will grow old, b) must be changed during its lifetime, and c) must survive the changes made. Here, we try to use refactoring [15, 6] to lengthen the expectancy of life of an unhealthy software system.

3 Case Study REX

The object of study is a medium scale and on-going commercial project which we will call REX[3]. REX is a web-based information-system consisting of Java-Native-GUI-Clients, a Bea weblogic server and an IBM DB/2 data base (see figure 1). Its core code base consists of 200kloc with

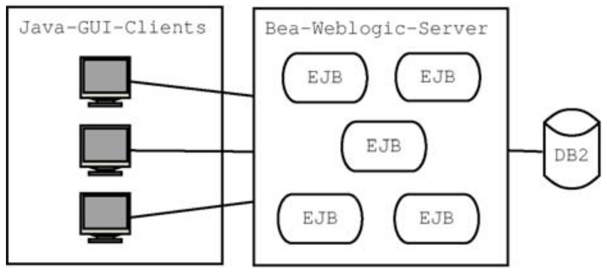

Figure 1. REX architecture

1553 classes resp. interfaces and 20.538 methods. Additional classes for the object-relational mapping are generated by the tool TopLink[4].

3.1 Project Pathology

The project was started 3 years prior to this case study and it consumed approximately 12 man-years so far. First estimates suggested that the required functionality could be

[2]at least at the source code level

[3]acronym for refactoring example; real project can not be disclosed.

[4]http://www.webgain.com/products/toplink

provided within one year. The project quickly got in trouble. It became evident, that a) the schedule could not be realized and b) some components were not working correctly. The reasons for this were diverse:

- incomplete and unstable requirements
- frequent changes und pressure of time
- underestimation of the complexity
- lack of coding and documentation rules
- late – if any – testing
- drop out of project leader
- unexperienced developers

Most of the troubles got under control but the code base itself is still in an unsatisfactory state. The development team regards the code as patchy and hard to comprehend. Integrating new features would take too much time because of unneeded redundancy. In short, the code lacks structure and it has evolved into "(Hyper-)Spaghetti-Code".

3.2 Refactoring Task

The project manager decided to carry out the experiment to improve the existing code base by tool-supported refactoring. A former part time member of the team was given the task to

1. identify weaknesses within the code,
2. select and perform suitable refactorings, and
3. assess the improvement achieved.

The goal of this effort was twofold. First, improvement of the code base and second, answers to the questions whether a) refactoring was practically applicable and b) refactoring was sufficient for the restructuring task.

The familiarity of the experimenter with the code accelerated the process and allowed us to complete this study within 5 months.

3.3 Tools

The experimenter decided to use the tools listed in table 1 for collecting code metrics and performing refactorings. Each tools was used in its most current version as of May 2003.

[5]http://www.thesmallworlds.com/
[6]http://www.togethersoft.com/products/controlcenter/index.jsp
[7]http://www.scitools.com/uj.html
[8]http://www.eclipse.org/
[9]http://www.intellij.com/idea/

Tool	metrics	refactor.
Small Worlds[5]	X	
Together Control Center[6]	X	X
Understand for Java[7]	X	
Eclipse[8]		X
IDEA[9]		X

Table 1. Selected Tools

4 Code Analysis

The first step of the expirement was to collect further information about the shortcomings of the code base. Statements such as "unstructured", "complex", or "unmaintainable" about a software system usually reflect the subjective opinion of an indivdual. We therefore tried to further substantiate the feelings of the development team about the deficits of the code by objective code metrics.

4.1 Metrics

The experimenter collected the following metrics [24] from REX:

- LOC(c): lines of code of class c
- NOIS(f): no. of import statements in file f
- NOA(c): no. of attributes in class c

  $$NOA(c) = NIV(c) + NCV(c)$$

  NIV: instance vars; NCV: class vars
- NOM(c): number of methods in class c
- CC(m): cyclomatic complexity [13] in meth. m
- RFC(c): response for a class

  $$RFC(c) = |M| + \sum_{i=1}^{NOM(c)} |R_i|$$

  M: set of methods m_i of c
  R_i: methods called by m_i
- CBO(c): coupling between objects
- LCOM(c): lack of cohesion of class c [18]

4.2 Tool Experience

The technical difficulties to collect these common metrics were greater than expected.

For example, the experimenter complained that Small Worlds required the user to explicitly deselect all other packages and classes within the same package before one class could be analyzed in isolation. The integrated editor did not provide an adequate search tool. The results of analyzes are not saved completely so that parts of the analysis must be repeated each time the project becomes reopened consuming up to 5 minutes for REX.

Together ControlCenter supported all metrics listed above. The possibility to set limits for values and have them highlighted proved particularly useful. But, collecting the metrics was also highly time-consuming. A complete analysis of the moderately sized code base of REX took several hours.

Similarly, Understand for Java needed hours to parse the code base and collect initial metrics like LOC. It took 5 more hours to generate reports, although the status bar showed 99% completion after just 20 minutes. A big part of the problem was the inability to restrict the analysis to certain classes.

Without any doubt, all tools named here provide advanced techniques far beyond conventional integrated development environments. But, our experiment also showed their limitations from a practical view point.

Although REX is not a very large project, the experimenter found it rather impractical to perform these analyses. Considering the low significance of the values obtained and the high effort needed, collecting metrics for a project like REX was viewed as too costly and will not be repeated without improved tool support.

4.3 REX Facts

Table 2 shows a summary of the metrics collected for REX. With the exception of the high LCOM value and CC

metric	avg	max
LOC	131	3287
NOIS	6	76
NOA	4	213
NOM	13	310
CC	22	521
RFC	98	907
CBO	8	145
LCOM	399	49.892

Table 2. REX metrics

exceeding a NASA recommendation by 3 units, there are no noticeable problems in the average values. The maximum values, on the other hand, exposed some possible locations for refactoring.

4.4 Manual Inspection

The measurements of the code base alone proved insufficient to identify suitable candidates for refactoring and had to be complemented with signficant manual inspections of the code base.

First, false positive indications were frequent. Some striking values related to classes that served special purposes, such as value ojects within the object-relational mapping. One tool, furthermore, identified 12134 methods as unused although they are indeed used. The tool was simply not aware of Java reflection.

Second, false negative indications are frequent, too. Classes might have an arbitrarily weak structure without any noticeable effect on metrics. The main drivers behind complexity in maintenance, redundancy and inconsistency, are hardly covered by the metrics listed above anyway because the tools focus on textual and structural aspects.

Because of this, the experimenter and the development team jointly determined classes in need of restructuring. The decision was based on their experience with the code without strictly abiding to the results of the measurements.

5 Refactoring REX

The Refactoring Home Page[10] currently lists 93 refactorings, ranging from renaming a variable to wrapping entities with sessions. Each refactoring changes the code while preserving its functional behavior.

Besides the presumed advantages concerning maintainability, refactoring might have side-effects on other aspects of the system that need to be regarded, too. For example, `Extract Class`, has an impact on the performance of system, too. Refactoring furtheron entails a loss of familiarity with the code for the developers. In addition to this, unit tests must be changed accordingly if additional classes and methods are created or deleted and documentation might have to be partially rewritten.

Clearly, the complete restructuring process can be time-consuming and risky and should be rewarded with a significant improvement of the code base.

5.1 Tool Support

The usefulness of refactoring depends on the availability of tool support because automation of small transformation steps is its distinguishing feature.

5.1.1 Missing Refactorings

Few of the 93 refactorings currently listed were supported by the tools and some refactorings, such as `Substitute Algorithm`, can not be automated at all. In average, the tools tested in this study supported approximately a dozen out of the 93 refactorings. Many of the easy to state refactorings are indeed hard to implement and can hardly be

[10]http://www.refactoring.com

expected to be fully automated. For example, moving a field from one class to another seems simple at first sight. In detail it requires cascading substitutions within classes and adjustment of import declarations. Eclipse and IntelliJ, for example, move the selected field without adjusting import declarations requiring additional manual work from the user. Thus, many important refactorings were not applied because of the absence of tool support.

5.1.2 Further Obstacles

Undo functions are particular useful for refactoring but hard to implement. For example, one tool silently omitted removing the package directory that was created by a previous `Move Class` operation.

Other refactoring steps furtheron consume extensive resources. Some single refactoring steps took up to several minutes on the dedicated Pentium III, 500MHz/500MB PC of the experimenter. In addition to this, the experimenter often had to touch the code manually after refactoring to complete the transformation step appropriately.

5.1.3 Method Extraction

To illustrate the practical limitations of refactoring we use a condensed `Method Extraction` example.

```
        int i = 0, j = 0;
        while (true) {
  ->        System.out.println(s);
  ->        i++;
        }
```

Figure 2. method extraction

Extracting the body of the loop in figure 2 yields the correct result shown in figure 3.

```
        private int foo(String s, int i) {
            System.out.println(s);
            i++;
            return i;
        }

        while (true) {
           i = foo(s, i);
        }
```

Figure 3. extracted method

If the loop body writes two instead of only one local variable, as shown in figure 4, the locals `i` and `j` can no longer be extracted by using an `int` return type for `foo`.

Conceptually, the type of `i` and `j` had to be converted into a mutable parameter type (such as `int []`) to be able to use call by reference. In addition the tool then would also have to change all operations performed on `i` and `j`

```
        while (true) {
  ->        System.out.println(s);
  ->        i++;
  ->        j++;
        }
```

Figure 4. two locals

into semantically equivalent operations on the new parameter type. Of course, the tool can not know the semantics in general. As a consequence, Eclipse simply raises an error message. The refactoring can not be performed. The attempt to accomplish the same extraction with a sequence of different refactorings, such as extracting the first two lines then changing the signature of foo, and so on leads too similar troubles. The extraction is better performed manually.

5.1.4 Summary

Based on these experiences we come to the conclusion that the existing tool support for refactoring is useful in various situations but not strong enough to adequately support a large-scale restructuring effort as envisaged with REX. Only part of this problem is up to tool vendors. Another problem lies in the refactoring concept itself. Some refactorings are inherently hard or even impossible to automate.

5.2 Selected Changes

Albeit these technical troubles we continued to perform refactorings on REX's code base, some with (partial) tool support others completely manual. Here, we present details of selected examples.

5.2.1 Move Class

The goal of the `Move Class` refactoring is to support structuring of classes into packages according to use or intentional criteria. The manual inspection of REX revealed a weak package structure. The placement of some classes into packages appeared virtually arbitrary.

The restructuring of classes into packages was completed rapidly with the help of tool supported `Move Class` refactoring steps. Although the benefit can not be demonstrated with metrics the team was highly satisfied with the result, judged the code as clearer, and saw positive effects on the reusability of the code.

5.2.2 Decompose Conditional

This refactoring allows to simplify long, complicated if-then-else statements by extracting fields and methods from the condition, then part, and else parts.

A high cyclomatic complexity (CC) of 190 (far above the recommended 15) directed the experimenter to a REX class containing several copies of the conditional expression shown in figure 5 in various methods.

```
if((zN.getET().intValue() ==
    NodeTypeSanction.SANCTION_TYP) &&
   (this.state != MType.READ_ONLY)) ...
                 ⇓

if(isVariant(zN) && isWritable()) ...
```

Figure 5. decomposed conditional

Unfortunately, there was no tool support for this refactoring making it a time-consuming and cumbersome task. It clearly provided better maintainability by means of reduced redundancy and increased readability. While these effects seem obvious it is interesting to note that metrics besides CC do not reflect any improvement, see table 3. All values are higher than before, leaving the impression that the code has degraded. LCOM increases significantly because the three methods added to the class hardly access any of its fields. Similar observations repeated throughout this case study. This experience shows that the benefit of refactoring can not necessarily be measured. The assessment of the outcome has to rely on personal judgement.

	before	**after**
LCOM	1614	1811
NOM	63	66
RFC	766	769

Table 3. metrics before and after

5.2.3 Further Refactorings

Further refactorings applied within the REX case study are briefly summarized as follows:

- `Introduce Parameter Object`, no tool support, increased metrics; useful in some cases but performance penalty of up to 290%
- `Replace Type Code with Strategy`, no tool support, increased metrics; only recommendable if it enables polymorphism
- `Replace Conditional with Polymorphism`, no tool support, increased metrics; only useful if identical conditionals are scattered across code
- `Extract Superclass`, no tool support, decreased metrics; improves clarity of the code but time-consuming
- `Pull Up Method`, tool support, increased metrics; rarely applicape
- `Extract Class`, tool support, decreased metrics
- `Move Method`, partial tool support, increased metrics

6 Results

After 5 months of code analysis and refactoring some shortcomings of the code base could be dispelled. But, the development team, project management, and the experimenter agreed, that REX's overall weak code base could not be improved significantly. An additional effort with a different approach will be needed to effectively improve REX's code base.

The experimenter summarized her experiences concerning retrospective refactoring as follows:

- *refactoring is far harder to apply then expected*
- *it is both time-consuming and error-prone*
- *the usefulness of refactoring for restructuring purposes without concrete need is doubtful because it is unclear whether the increased beauty will simplify or aggravate future changes*

6.1 Semantics Versus Syntax

Only a small part of our experiences is due to lack of tool support. By analysing the problems experienced we identified a core property of refactoring as the main obstacle for effective restructuring of larger code bases, that is syntactical instead of semantical transformation.

It should be obvious that virtually all 93 refactoring are predominantly syntactical operations. They are mostly concerned with moving text fragments from one place to the other with little change to the content. By this, they are obviously not able to restructure weak algorithms. This can be illustrated with the multiplication example shown in figure 6. The loop can trivially be simplified into a single assignment using basic arithmetics. It is also evident that refactoring does not support transformations of this kind because the connection between addition and multiplication is a semantical relation and not syntax.

```
a = 0;
while (c > 0) {
    a = a + b;     ⇒   a = b * c;
    c--;
}
```

Figure 6. transformation of multiplication

The trouble is, that many of the shortcomings in a decayed code base are of this nature. Maintainers' nightmares, like while loops with hundreds of statements and conditionals can typically not be fixed by moving the statements withing the loop into a series of methods. They rather indicate algorithmic shortcomings that need to be fixed by renovating the underlying algorithmic idea including data types.

Therefore, it seems that refactoring is inherently limited to the context of agile software development, where it is used frequently to accomplish concrete changes. In contrast to this, REX showed that it is of little help for retrospective restructuring of larger code bases for quality reasons.

7 Conclusion

This paper presented the results of a 5 months case study trying to imrove the quality of a commercial, medium size code base by refactoring. The code was analyzed with both metrics and manual inspection to detect suitable spots for refactoring and the refactorings were performed with tool-support where possible or manually otherwise. The result on the code base was nonsatisfying. Extensive manual rework will be needed to actually achieve the desired improvement of the code base. Besides this, refactoring proved to be much more difficult and time-consuming than expected. One reason for this is lacking tool support, another part lies in the inherent limitations of refactoring which are syntactical in nature. Refactoring is definitely useful in many situations but of minor help for a large scale consolidation effort.

From the lessons learned in the REX case study we recently started another consolidation experiment. Following the idea of an evolutionary software process modell [19] that alternates between bottom-up growth and top-down consolidation phases, this experiment focuses on semantic consolidation. Here, the experimenter uses a process for manual inspection tailored to detect redundancies, inconsistencies and algorithmic aspects. The results of this follow-up study collected so far are promising and will be published separately after completion of the study.

Surely, “a fool with a tool is still a fool” and we do not claim that we were able to use refactoring in an optimal way. Nevertheless, this case study exposed the practical limitations of using refactoring for retroactive restructuring in a realistic commercial setting.

Acknowledgment It is pleasure to thank Elena Kolodizki and Martion Ober for supporting this work. Mrs. Kolodizki carried out the extensive and time-consuming experiments described in this paper as part of her master thesis [9].

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] D. Cohen, M. Lindvall, and P. Costa. Agile software development. Technical report, Fraunhofer Center for Experimental Software Engineering Maryland and The University of Maryland, 2003.

[3] X. Corporation. Aspect-oriented programming home page. http://www.parc.xerox.com/aop/, 2000.

[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Technical Report 81, National Institute of Statistical Sciences, Mar. 1998.

[5] K. B. et al. Manifesto for agile software development. www, 2001. http://agilemanifesto.org/.

[6] M. Fowler. Refactoring: Improving the design of existing code. *Lecture Notes in Computer Science*, 2418:256, 2002.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1 edition, 1999. jun.

[8] W. Kadir and P. Loucopoulos. Relating evolving business rules to software design. In *Int. Conf. on Software Engineering and Practice (SERP)*, volume 1, pages 129–134, Las Vegas, NV, June 2003. CSREA Press.

[9] E. Kolodizki. Einsatz von refactoring in langzeitprojekten – fallstudie. Master’s thesis, Technische Universität München, May 2003.

[10] M. Lehman. The programming process. Technical Report RC2722, IBM Research Centre, Yorktown Heights, NY, Sept. 1969.

[11] M. Lehman. Software evolution threat and challenge. Professorial and Jubilee Lecture, Oct. 2003. 9th international Stevens Award, hosted by ICSM 2003.

[12] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maitenance Management*. Addison Wesley, Reading, 1980.

[13] T. McCabe. A complexity measure. *Int. Conf. on Software Engineering*, 2(4):308 – 320, 1976.

[14] F. Niessink. Perspectives on improving software maintenance. In *Int. Conf. on Software Maintenance*, Florence, Italy, Nov. 2001.

[15] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign , IL , USA, 1992.

[16] D. L. Parnas. Software aging. In *Int. Conf. on Software Engineering*, pages 279–87, Sorrento, Italy, 16–21 May 1994. IEEE.

[17] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[18] L. H. Rosenberg and L. Hyatt. Applying and interperting object oriented metrics. In *Software Technology Conference*, Utah, Apr. 1998.

[19] T. Seifert and M. Pizka. Supporting software-evolution at the process level. In *Net.ObjectDays 2003*, Erfurt, Germany, Sept. 2003. tranSIT GmbH.

[20] H. Sneed. *Software-Sanierung*, chapter 1, pages 123–141. B. G. Teubner, 1988.

[21] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

[22] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[23] B. Wix and H. Balzert, editors. *Softwarewartung*. Angewandte Informatik. BI Wissenschaftsverlag, 1988.

[24] X. S. Zikouli. Object-oriented metrics a survey.