High-impact Refactoring based on Architecture Violations

Fabrice Bourquin
Zühlke Engineering AG
Zürich-Schlieren, Switzerland
fbo@zuehlke.com

Rudolf K. Keller*
Zühlke Engineering AG
Zürich-Schlieren, Switzerland
ruk@zuehlke.com

Abstract

Software refactoring has been identified as a key technique for the maintenance and evolution of object-oriented systems. Most interesting are high-impact refactorings, that is, refactorings that have a strong impact on the quality of the system's architecture. "Bad smells" and code metrics have been suggested as means for identifying refactoring needs. According to our experience these techniques are useful, yet, in order to spot opportunities for high-impact refactorings, they should be complemented with the analysis of architectural violations. The subject of this report is a mid-sized Java enterprise application from the telecommunications domain whose functionality had to be radically extended. We show how we combined several tools and techniques to identify opportunities for high-impact refactorings, and discuss the resulting architecture, the refactoring process, tool support as well as related experiences.

1. Introduction

As more and more emphasis is being put on software quality and maintainability, the tasks devoted to software development teams are evolving, and themes like software testing or refactoring are gradually gaining importance. Consequently, developers are increasingly expected to work on existing applications rather than on creating new ones.

In this context, new challenges arise. Existing applications are to meet new business expectations, and new developments must be integrated into existing applications without destabilizing their base functionality. Most importantly, the software architecture also has to adapt to these new expectations.

Refactorings have typically been used as a means for improving detailed design and code quality [1]. Yet, to effectively address the above challenges, refactorings at the architectural level are required. To this end, Roock and Lippert for instance describe "large refactorings", that is, composites of small refactorings [3]. We suggest the more general notion of *high-impact refactorings,* meaning that the refactorings have a strong impact on the quality of the system's architecture.

An obvious, yet crucial prerequisite for carrying out high-impact refactorings is the identification of opportunities for applying the refactorings. Roock and Lippert suggest that "architecture smells" [3] are a good indicator for these opportunities. According to our experience, we consider architecture violations as the single most important architecture smell.

To substantiate our hypothesis that architecture violations point at high-impact refactorings opportunities and that these refactorings may lead to important quality gains, we decided to carry out a case study. The subject of the case study is a mid-sized Java enterprise application from the telecommunications domain whose functionality had to be radically extended, and which underwent intense development activities.

This report presents the experience gained from identifying and applying refactorings on that application. Note that this is an experience report and not a research paper. The data at hand was gathered from one single project and not from a series of projects. It is not the result of a controlled experiment, but was produced in a kind of post mortem project analysis. However, since the application at hand is a fairly typical industrial-strength system, our conclusions may be general enough to apply to a wide range of applications.

Below, we first give an overview of our case study. Then, approaches for identifying refactoring opportunities are discussed. Afterwards, we describe

[1] The second author carried out part of this work as Adjunct Professor at Université de Montréal, Canada.

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

the target architecture of the case study, and the implementation of the refactorings. In the following section, the results of the refactorings are assessed. Then, we discuss our approach from the perspectives architecture, process, tool support, and business impact. Finally, we wrap up with a conclusion.

2. The Case Study

The subject of our case study is a Java enterprise application developed by a leading Swiss telecommunications company. Put into production as early as 2002, it has never ceased to be extended and refined until the writing of this report. Over time the application grew in importance, and in summer 2005, the application was transferred, as part of an overall reorganization, to a new development team.

Table 1 shows the most relevant characteristics of the application (as of December 2005, unless otherwise stated).

Table 1. Some characteristics of the application

Development started	June 2001
Java Source Lines in 2002	ca. 50000
Java Source Lines in 2005	140000
Java Classes in 2002	ca. 300
Java Classes in 2005	850
Frameworks used	Java EE, Struts
Application Server	Tomcat, Custom
Development Environment	Eclipse, Ant, CVS
Size of Development Team	6 Persons (full time)

When we joined the project in late 2005, the project team had to tackle an impressive list of change requests and had the task to carry through several extension subprojects. Clearly, the team's primary goal was to integrate new functionality into the application. However, before long it became evident that substantial extensions could only be done after previously correcting various significant defects and limitations of the application. Most importantly, to support ongoing development tasks and guarantee maintainability, the application's architecture would have to be improved radically.

The project team was challenged by low development resources, which were mostly assigned to concrete, business-oriented needs, and by a lack of know-how due to team change. In this context, only refactorings with the highest relevance to the architecture's quality were to be applied. Thus, high-impact refactorings became a focus of our activities.

3. Identifying Refactoring Opportunities

A widespread method for identifying refactoring opportunities is looking for so-called "bad smells". A bad smell is an intuitive sign that can point out a need for refactoring. Fowler lists several bad smells [1], and Kerievsky extends the list [2]. While these bad smells mostly apply to code and design, Roock und Lippert more recently introduced specific bad smells for architectural aspects which they call architecture smells [3]. More formal approaches for identifying refactoring opportunities are based on the use of code metrics [5].

Below, we first discuss the drawbacks of bad smells and code metrics, respectively. Then, we introduce, as an alternative, the analysis of architecture violations. It is this latter approach that allowed us in our project to locate and eventually implement high-impact refactorings.

3.1. Difficulties with Bad Smells

Several difficulties may be encountered when trying to identify bad smells in an application like ours, including:

- Finding bad smells works best when having a prior knowledge of the software. In our case study, the team only got to know the application few months before starting refactoring.
- Most bad smells focus on a small group of related classes [1] [2]. Higher-level structural defects repeated throughout the application are beyond the scope of these bad smells.
- Bad smells cannot be quantified easily, and therefore are hard to prioritize. Moreover, the relevance of a particular bad smell in respect to the overall application is often difficult to assess.

Because of their intuitive nature, bad smells come often in very handy, especially when the focus is well defined (for example, when a few classes or a subsystem has to be extended or corrected). However, in our setup, we had to question our intuition since we did not have a thorough understanding of the application at the outset.

In fact, some bad smells were very easy to identify in our application. After only a few hours of analysis, we ended up with a quite long list which we could use later on to successfully fix a large number of minor defects (i.e. small-scale defects limited to a few classes). Our difficulties started when trying to work out a smaller list of defects to be fixed first: we were

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

unable to identify which ones were most relevant for the overall quality of the application. Worse, by exclusively relying on bad smells, we would not have been able to identify the sources of the major defects of the application (i.e. large-scale defects extending through many components of the application).

The architecture smells described in [3] can be considered as bad smells at the architectural level. Their usage requires an intimate knowledge of the application; furthermore, it is very hard to quantify them. The one notable exception constitutes architecture violations, an architecture smell whose detection can largely be automated, and which has proven to be key to high-impact refactorings, as will be shown below.

3.2. Drawbacks of Code Metrics

A major alternative to looking for bad smells is using code metrics [5]. This approach indeed avoids several problems encountered with the previous approach. In fact, code metrics are quantified, and do not require any prior knowledge of the application. However, they also have serious drawbacks, including:

- The multiplicity of possible code metrics makes it difficult to know which ones are the most relevant for a specific refactoring goal.
- Code metrics typically do not take into account design and architecture aspects, which makes their relevance in regard to architectural refactorings questionable.

Of course, code metrics could easily be computed for our application; however, the collected data did not point out architecture problems in any meaningful way (they were still useful, though, at assessing and improving code-level quality).

Higher-level metrics, such as object-oriented design metrics, have been known for quite some time [6]. However, these metrics have not yet been assembled into composite, architecture-relevant measurements applicable to our industrial context. Therefore, we did not pursue this approach any further.

3.3. Analysis of Architecture Violations

As both bad smells and code metrics exhibit significant drawbacks when being used at the architectural level, we further investigated the concepts set out by Roock and Lippert [3]. After several iterations, we came up with a four-step procedure adapted to our context and objectives:

1. Create a formal model of the application's architecture (for example, a layer model).
2. Assign each existing package of the application to a component of the architecture model.
3. Check the actual references between the packages of the application against the architecture model. Any reference that is not permitted by the architecture model gives rise to a so-called *architecture violation*.
4. Focus on the packages that cause the most architecture violations, and apply the refactorings that are most likely to correct them.

Whereas this method helps identifying opportunities for refactorings with the highest possible impact on the application's architecture, it must be noted that individual refactoring steps are still identified and performed according to the known techniques described in [1], [2] and [3], including looking for bad smells when solving problems of a finer granularity. Thus, the analysis of architecture violations does not replace existing techniques, but rather complements them. Its main added value is the focus on architecture and its quantitative nature and potential for automation. It also integrates very naturally into an iterative process, which we will discuss in 7.2.

In today's context, where most development teams (including ourselves) have to cope with limited resources and numerous requests from stakeholders, refactoring must be performed efficiently. We consider as most efficient refactorings that improve the overall quality of the application's architecture. Therefore, in our case study we put much emphasis on the analysis of architecture violations in order to achieve high-impact refactorings – as a method of choice for optimizing stakeholder benefits from refactoring.

4. Defining the Architecture

According to the procedure described in the previous section, the starting point for refactoring our application was to create a formal model of its architecture. To this end, we leveraged information from various sources, including:

1. Our freshly-gained knowledge of the application's internals.
2. Insights carried over from the previous development team.
3. Analysis of package references using the Sotograph [8] tool.

In line with best practices in software engineering, we opted for a layer model, as shown in figure 1. In this model, references between components (i.e.

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

method call or access to attribute) are subject to the following rules:

- A component may reference components from lateral or lower adjacent layers.
- A component must not reference components from higher layers.
- Unless otherwise stated, a component must not reference other components of its own layer.

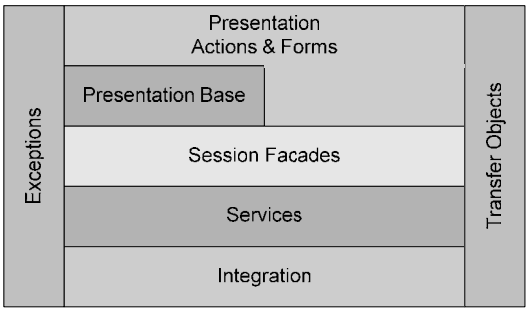

Figure 1. The layer model of our application

At this point, we were in need of a powerful software analysis tool, and we settled for Sotograph with which we were already familiar from previous projects (as a consequence, we were limited to statical code analysis; we discuss this point in 7.3). We entered our layer model into the tool, and mapped each existing Java package to one architectural layer. The result is shown in figure 2 (i.r.t. figure 1, figure 2 is rotated by 90^0 counter clockwise; the layers Presentation Actions & Forms and Services are each shown as two large piles of packages, whereas the layers Exceptions and Transfer Objects are not shown altogether). The Sotograph tool was then used to produce for each package the number and details of the incoming and outgoing architecture violations, of which an excerpt is shown in table 2. In carrying out the last step of our procedure, we focused on the packages with the most architecture violations as targets for refactorings. This step is detailed in the following section.

Table 2. Architecture violations for each package

Package	InboundArchViol
server	1330
presentation.package1	980
presentation.package2	968
presentation.package3	724
presentation.package3	678
presentation.package4	450
presentation.package5	440
presentation.package6	333
…	…

5. Performing the Refactorings

At the outset of the refactoring step it became clear that many architecture violations originated from classes that were simply placed in the wrong packages. A trivial, but important first action was to move these classes to their proper package, changing the package structure when necessary.

As a second action, still focusing on the packages with the most architecture violations, several known bad smells from [1] and [2] were identified. Interestingly enough, many of them had been overseen (or occasionally completely forgotten) in our prior, non-automated analysis. There were various reasons for this, the most relevant ones being:

1. The team spotted bad smells more efficiently in the parts of the application on which it had recently worked, treating other lesser known (but sometimes crucial) parts of the application more casually.
2. The team was often focusing on "unpretty" parts of the application that were indeed hurting good taste but had little overall significance or impact.
3. The bad smells identified by team members with excellent communication skills were at times perceived as more important than the ones suggested by team members who explained theirs less eloquently.

Although some of these bad smells could easily be eliminated via simple refactorings, most of them were found to bear the potential for high-impact refactorings. Table 3 contains a list of some of the most relevant refactorings performed in our case study. As the entries in the column "Time span" suggest, some of the refactorings were carried out during an extended period of time, which is mainly due to the high dynamics of the project. Note that many refactorings were applied several times – the refactoring "Introduce Dynamic Proxies", for instance, more than 200 times. Typically, the high-impact refactorings performed were inspired from the literature, yet had to be adapted to varying extents to the specific context of our application.

Table 3. High-impact refactorings performed

Refactoring	Time span
Extract Profile Object	12/2005
Introduce Session Facades	12/2005-07/2006
Fix Action Inheritance Misuse	01/2006
Introduce Dynamic Proxies	05/2006
Create Workflow Blackboard	12/2005-07/2006
…	…

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

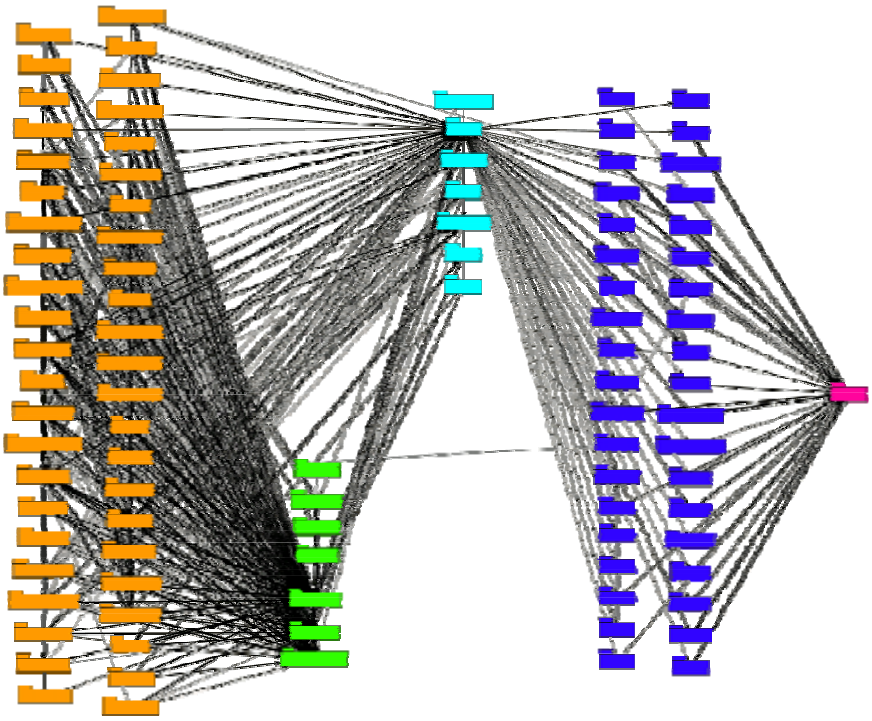

Figure 2. Our application's packages arranged by layer (as of July 2006)

We suggest that similar problems and refactorings solutions are likely to occur in other Java enterprise applications whose target size and scope has been initially underestimated, and where measures for extending the architecture have not been taken early enough. This assumption is based on our experience with similar projects [13], yet still requires further investigation.

In the following two sections, we describe two of the high-impact refactorings we consider as particularly interesting, and which we will refer to later on in this report.

5.1. Creating a Workflow Blackboard

Enterprise applications often have to deal with much user input. In this case, it is common practice to split complex input dialogs into several simpler steps, logically linked via a small workflow. The advantages of such a solution include:

1. The user interface gets more straightforward.
2. The user may be notified early on in the dialog of incomplete or invalid data.
3. Alternative flows can be defined more easily (including skipping steps depending on data already entered).

In the case of web applications, several well-known frameworks have a built-in "workflow engine", or provide it as a plug-in. However, the Struts framework used in our application does not provide such an engine, and basic workflow functionality has been implemented as part of the application.

At the end of the workflow, the data must be collected for further processing. In simple cases, the straightforward (but generally not recommended) solution of going through all input forms can be used, as shown in figure 3.

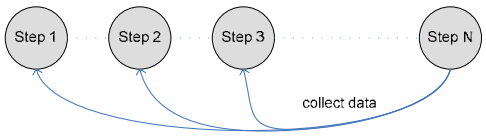

Figure 3. Collecting data at the workflow's end

One of the many drawbacks of this method is its lack of scalability, when the workflow grows or when alternative flows are needed, since more and more logic must be built into the data collection part. If no action is taken, this may result in the maintenance nightmare of managing tightly linked workflow steps

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

and control logic scattered everywhere in the source code, as illustrated in figure 4, and as initially found in our application.

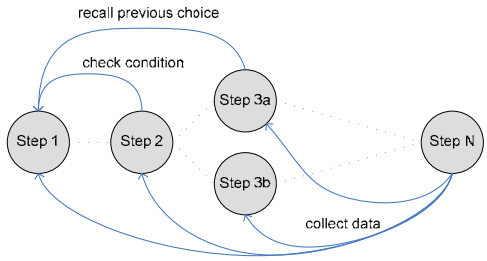

Figure 4. Excessive dependencies in a workflow

We have addressed this problem with the *Workflow Blackboard*, a refactoring we developed for our application that borrows some ideas from the "Case Data" pattern [12]. The general idea behind the refactoring is the introduction of a set of classes designed to hold the data collected throughout the workflow, and which is updated appropriately after each step. The resulting solution is shown in figure 5 and has the following advantages:

1. Single steps do not depend anymore on each other, since they can query previous user input by looking at the data stored in the *Workflow Blackboard*.
2. No complex data collection logic is required anymore at the end of the workflow.

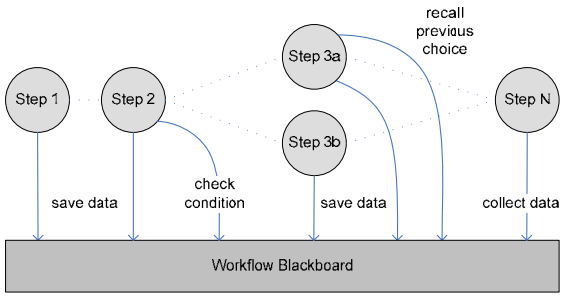

Figure 5. Introducing a Workflow Blackboard

In our application, the *Workflow Blackboard* was directly used as a Transfer Object [10] to communicate with the back-end server, which allowed for reusing existing objects and reducing development effort.

5.2. Introducing Dynamic Proxies

Highly-available applications typically have the requirement to notify system administrators when unexpected errors occur repeatedly. In our case study, the application was required to send a text message to a cellular phone whenever a remotely accessible method on the application's back-end threw a technical exception.

This feature had been implemented by using Proxy classes [9] containing custom notification code. The following code excerpt outlines this solution:

```
public CustomerData getCustomer(String id)
    throws NotFoundException,
           TechnicalException {
  try {
    return delegate.getCustomer(id);
  } catch (NotFoundException nfe) {
    throw nfe;
  } catch (TechnicalException te) {
    notifier.sendAlarm(
      // Trigger of architecture violation!
      MonitoringUI.CUSTOMER, te);
    throw te;
  }
}
```

This solution had been in use for several years. Ultimately, more than 200 remotely accessible methods were supported, with occasionally very long lists of parameters. When we entered the project, the quality of the solution had much degraded, since only about 75% of the methods were correctly implemented, as was revealed by a manual inquiry.

The solution after refactoring makes intensive use of Java dynamic proxies and the recently introduced annotations [7], thus spectacularly replacing more than 200 methods with a single one:

```
// Generic proxy method
public Object invoke(Object proxy,
    Method method, Object[] args)
    throws Throwable {
  try {
    return method.invoke(delegate, args);
  } catch (InvocationTargetException ie) {
    Throwable te = ie.getTargetException();
    UseNotify un = method.getAnnotation(
      UseNotify.class);
    if (te instanceof TechnicalException
        && un != null) {
      notifier.sendAlarm(un.value(), te);
    }
    throw te;
  }
}

// Enabling notifications, the easy way
public interface CustomerService {
  @UseNotify(ServiceGroup.CUSTOMER)
  CustomerData getCustomer(String id)
    throws NotFoundException,
           TechnicalException;
}
```

This solution eliminates many defects in the existing code in an elegant way. Moreover, it curtails the waste of time experienced by the developers who

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

had to manually update the proxy class each time a slight change in a remotely accessible method was required.

6. Assessing the Results of Refactoring

According to Mens and Tourwé [4], an important step in the process of refactoring software is "assessing the effect of the refactorings on quality characteristics of the software". However, little has been said about this topic in the literature, possibly encouraging some erroneous, but widely accepted popular opinions:

- "A refactoring can never go wrong."
- "Feeling like having improved the situation is sufficient to justify a refactoring."

Of course, not having concrete and quantifiable feedback after a refactoring makes it difficult to systematically improve one's practice over time.

In our case study, two formal ways of measuring the results of refactoring were used. Furthermore, anecdotal evidence about the stakeholders' satisfaction with the application was gathered.

6.1. Evolution of Architecture Violations

A possible approach for assessing the results of refactoring is looking at how architecture violations have evolved over time. This may seem trivial in our case, because refactorings were driven by this very measurement. However, as reported in section 5, the concrete refactoring steps were to a large extent performed by looking for bad smells from [1] and [2] and by applying known techniques that are not primarily focused on the architecture. Refactoring may well produce new architecture violations, and nothing guarantees that it will create a smaller number of violations than the amount it helped to eliminate in the first place.

It must also be noted that ongoing development activities, as witnessed in our case study, may produce new architecture violations while refactoring activities are eliminating pre-existing ones, thus hiding the impact of refactoring on the number of architecture violations. In practice, this measurement provides interesting insight on how the application's architecture is evolving, yet for assessing the effect of an individual refactoring, it might be of limited value.

As can be seen in the upper chart of figure 6, architecture violations decreased dramatically in 12/2005 during the first weeks of refactoring, where most "quick-wins" (e.g. moving misplaced classes) were identified and carried through. Additional architecture violations generated by functional extensions to the application were neutralized by ongoing refactoring activities. In fact, after 7 months, the application had grown by more than 50%, as shown in the lower chart of figure 6, yet still comprised less than 50% of the initial amount of architecture violations. The fact that only limited resources had been available for refactoring during that period suggests that the refactorings had been quite effective.

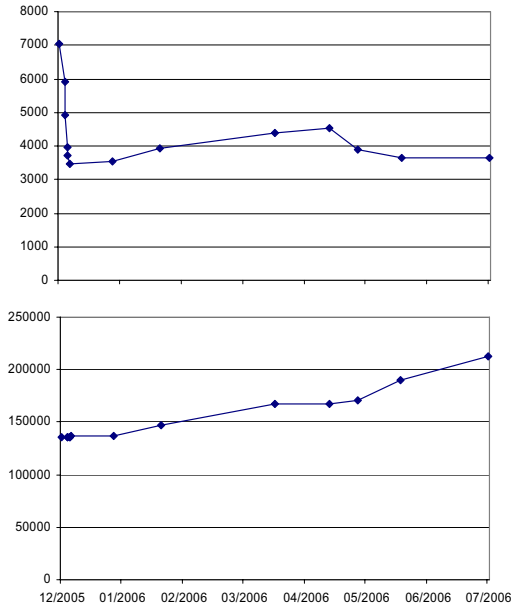

Figure 6. Upper chart: Architecture violations; lower chart: Lines of Java code

Table 4. Architecture violations by layer

Layer	12/2005	01/2006	02/2006	04/2006	05/2006	07/2006
Presentation – Actions & Forms	3308	2323	2700	3000	2997	3042
Presentation – Base	1478	1071	1094	1133	505	477
Session Facades	0	12	0	5	5	5
Services	1718	29	29	47	47	47
Integration	0	0	0	0	0	1
Exceptions	0	0	0	0	0	0
Transfer Objects	516	89	93	292	340	76

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

Furthermore, table 4 depicts for each architectural layer the evolution of architecture violations over time. The effect of two of the refactorings can be observed easily:

- Moving classes to the proper packages, between 12/2005 and 01/2006.
- Introducing Dynamic Proxies, between 04/2006 and 05/2006.

Other refactorings, such as Create Workflow Blackboard, were performed step by step, sometimes over several months and in concert with other refactorings, and therefore cannot be identified clearly.

6.2. Evolution of Code Metrics

Another approach to measuring the results of refactoring activities is the use of code metrics [5]. Some simple metrics for our application are shown in table 5. The following can be noticed for the seven months period of the case study:

- The average number of lines of code per class decreased by about 10%.
- The number of classes taking part in cyclic references was cut in half.
- JavaDoc comments grew faster than the rest of the application.

In short, most code metrics either improved or remained stable during the refactorings. At first sight it might not be evident that eliminating architecture violations has the side effect of increasing code quality. Yet, there are several plausible explanations:

- Most well-known refactorings ultimately tend to improve code quality.
- Developers working on high-impact refactorings might fix code-level problems on the sidelines.
- New developments, performed in our case study at the same time as refactorings, probably have a better quality than existing ones.

A notable exception to the above observations is the number of duplicated code blocks which increased dramatically. This increase could be tracked down to some of our refactoring activities. This measurement showed us the need for additional "code cleanup" iterations which we would have probably forgotten, if such code metrics had not been readily available.

6.3. Feedback from Stakeholders

During the period of refactoring, the application was put twice into production, each time with success. From the beginning, refactoring was tightly interlaced with numerous functional extensions required by ongoing subprojects.

The feedback gathered from the field after those two productive releases has been very positive. The existing functionality has been running with very few new defects; most of them were found to be unrelated to the refactorings.

7. Discussion

Our experience with high-impact refactorings has been quite positive, and the quality of our application has improved significantly. The following sections detail this assessment, from an architecture, a process, a tool, and a business perspective, respectively. Finally, related work will be discussed.

7.1. State of the Architecture

Many improvements have found their way to our application's architecture, and its quality is doubtlessly much higher today than it was one year ago. Several cases of misuse of Java EE functionality have been identified, either based on architecture violations or by the more mundane method of digging through code. Refactorings have been aimed at the problems most relevant to the application's architecture, and many of them were solved. The effectiveness of the refactorings is evidenced by various formal measurements, by the

Table 5. Evolution of code metrics during the refactorings

Metric description	12/2005	01/2006	02/2006	04/2006	05/2006	07/2006
Attributes per Class (Average)	3.21	3.17	3.03	3.17	3.09	3.16
Methods per Class (Average)	9.90	9.92	9.61	9.94	9.62	9.69
Lines of code per Class (Average)	160.1	160.2	158.4	159.0	154.6	150.6
Lines of code per Method (Average)	15.0	15.0	15.2	14.9	14.9	14.3
Classes in a Cycle	99	67	64	71	72	49
Duplicated Code Blocks	230	233	311	360	381	524
JavaDoc Comment Blocks	6503	6583	7104	8679	8921	11326
Non-JavaDoc Comment Blocks	3018	3012	3139	3197	3316	3893
Number of Java Classes	851	855	930	1055	1106	1411
Total lines of Java Code	136267	136954	147339	167767	171002	212549

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

team's positive assessment, as well as by the fact that a lot of new functionality could be added to the application without compromising its architectural quality.

However, some problems seem to be too complicated to be tackled at the moment. An example of this, named "Too Much Data in Session", is being discussed in [11]. Refactoring techniques do exist for this kind of problems, and throwing more skills and money at them would most probably solve them. However, we believe that the most efficient way of tackling these problems would be to fully "reengineer" some components instead of only trying to fix some of their defects.

Not all components in our case study seem to qualify for reengineering though, since a full reengineering would likely lead to higher costs and risks than refactoring. At the moment, it seems to be appropriate mainly for components for which numerous requests for extensions are registered.

To find a compromise between refactoring and reengineering is often difficult in practice, and there seem to be cases where neither one is a good solution. We believe that a good testing strategy [14] could reduce the risks of reengineering and would probably be one of the winning moves for ensuring the future of our application.

7.2. The Refactoring Process

Figure 7 shows how the activities introduced in our case study integrate into the process described in [4]. The result is an iterative refactoring process comprising the three main stages identification, refactoring, and assessment. The key idea of relying on architecture violations closes the loop between assessing the results and identifying further refactoring opportunities.

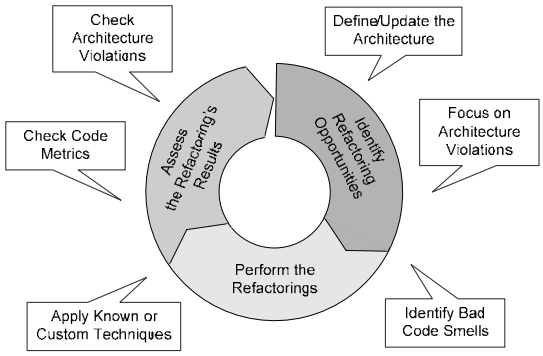

Figure 7. The refactoring process (iterative)

Note that in spite of focusing on architecture violations, leveraging bad smells is an integral part of the process. As noticed in section 6.2, the use of code metrics for assessing the results of refactoring proves useful, because it can highlight possible side effects of architecture improvements at the design and code levels.

7.3. Need for Tool Support

An essential prerequisite for our approach is effective tool support. Various architecture analysis tools are available. We have adopted Sotograph as our tool of choice for two reasons. First, we have had positive experience with it prior to the case study. Second, and most importantly, its functionality for managing architecture violations supports our refactoring process very well. It therefore soon became a key component of our development infrastructure.

However, the Sotograph and similar software analysis tools have their limitations: the intensive use of reflection by Java EE frameworks often makes all efforts based on static code analysis futile. In our case study, most of the user interface of the application, based on Struts, could not be analyzed correctly, which made some metrics such as "Unused Methods" or "Unused Attributes" virtually unusable. Some support for dynamic code analysis would also greatly improve the efficiency of these tools.

Ultimately, tool limitations could bias the process of focusing on high-impact refactorings. Thus, there is definitely a pressing need for extensive Java EE tool support.

7.4. Business Needs and Refactoring

Architecture problems have a big impact on the maintenance and development costs of an application. Solving those problems leads to improvements that ultimately help reducing costs, including:

- Further actions of perfective maintenance are facilitated.
- Extensions to the application are easier to develop and show a lower defect rate.
- Existing components of the application are easier to reuse.

We believe that high-impact refactoring is a cost-effective way of improving an application's architecture, and one of the best ways of reducing an application's maintenance and extension costs. Figure 8 illustrates the expected influence of high-impact refactoring on costs. At the beginning, the cumulated effort (cost) is equally shared between development and maintenance. Subsequent refactoring generates a

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE

temporary peak of the cumulated effort. Yet eventually, refactoring definitely helps harnessing the cumulated effort of development and maintenance.

Figure 8. Expected influence of high-impact refactoring on costs

7.5. Related Work

Murphy et al. suggest a software reflexion model technique [15] as a lightweight and iterative means for comprehending and eventually reengineering complex software systems. This technique integrates well with the two-phase process described in [16] which emphasizes the use of abstract models for triggering refactoring activities. Put together with the complex restructurings discussed in [3], these ideas are at the basis of our work.

8. Conclusion

Many industrial-strength applications do not enforce a well-defined architecture. While revising the architecture might dramatically improve the overall quality and maintainability of the application, the problem is difficult to tackle because classical refactoring techniques are not primarily focused on the architecture.

In this report, we presented the process of aiming refactoring activities at high-level, architecture-relevant problems, and gathering quantitative feedback about how the architecture has improved after the refactorings. We illustrated our matter using concrete data from a real-world Java enterprise application. Finally, we critically assessed our refactoring results, and gave a discussion of our approach.

In conclusion, we have tried to present an interesting and genuine way of making known techniques and tools work together towards a flexible and productive high-impact refactoring environment. We believe that such environments constitute a key success factor in modern day industrial projects.

9. References

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

[2] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.

[3] S. Roock and M. Lippert, *Refactoring in Large Software Projects: Performing complex restructurings successfully*, John Wiley & Sons, 2006.

[4] T. Mens and T. Tourwé, *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126-139, 2004.

[5] N. E. Fenton and S. L. Pfleeger, *Software Metrics. A Rigorous and Practical Approach*, PWS Publishing Company, 1997.

[6] R. Harrison, S. Counsell and R. Nithi, *An Overview of Object-Oriented Design Metrics*, Proceedings of the Eighth International Workshop on Software Technology and Engineering Practice (STEP'97), pp. 230-235, London, England, 1997.

[7] Sun Microsystems, *The Java Platform*, http://java.sun.com.

[8] Software-Tomography GmbH, *Sotograph*, http://www.software-tomography.com.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-wesley, 1995.

[10] D. Alur, D. Malks, J. Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*, Second Edition, Prentice Hall, 2003.

[11] B. Dudney, S. Ashbury, J. Krozak and K. Wittkopf, *J2EE Antipatterns*, Wiley, 2003.

[12] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M .P. van der Aalst, *Workflow Data Patterns*, QUT Technical Report, Queensland University of Technology, Brisbane, Australia, 2004.

[13] R. Weber, T. Helfenberger, R. K. Keller, *Fit for Change: Steps towards Effective Software Maintenance*, Proceedings of the International Conference on Software Maintenance (ICSM'05), pp. 26-33, Budapest, Hungary, 2005.

[14] S. Berner, R. Weber, R. K. Keller, *Observations and Lessons Learned from Automated Testing*, Proceedings of the Twenty-Seventh International Conference on Software Engineering, pp. 571-579, St. Louis, MO, USA, 2005.

[15] G. C. Murphy, D. Notkin, K. J. Sullivan, *Software Reflexion Models: Bridging the Gap between Design and Implementation*, IEEE Transactions on Software Engineering, vol. 27, no. 4, pp. 364-380, 2001.

[16] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, Ch. Verhoef, *A Two-phase Process for Software Architecture Improvement*, Proceedings of the International Conference on Software Maintenance (ICSM'99), pp. 371-380, Oxford, UK, 1999.

Publ. in Proc. of the 11th Europ. Conf. on Software Maintenance and Reengineering, Amsterdam, NL, March 2007, pp. 149-158. IEEE