Refactorings in Large Software Projects

How to Successfully Execute Complex Restructurings

Martin Lippert: lippert@acm.org
Stefan Roock: stefan@stefanroock.de

Date: 11/01/2004

1 Introduction

Big Upfront Design

Once software developers believed it was possible to create the technical software design for a comprehensive system completely, correctly and free of contradictions right at the beginning of a project. Many projects proved though that this ideal approach can hardly be realized. More often it causes significant problems.

A typical example of this fact are requirements that were either unknown or not taken into consideration at the beginning of a project and thus were not integrated in the original system design. Later on, integration of these disregarded requirements into the project will be much more difficult. If the developers are lucky, the requirements will seamlessly fit into the existing system. However, this is rarely the case. So-called 'work-arounds' are needed. These enable developers to meet the requirements within the system, even though the actual software design is not suitable for such a method.

Loss of Structure

One problem of these work-arounds is that they cause a gradual degeneration of the system design that leads to a *loss of structure*. The more work-arounds are built into the system, the more difficult it becomes to recognize and apply the original software design. Often developers describe such a system as 'historically grown.'

Today, many development methods have a different approach to software design. Especially agile development methods – most prominently *extreme programming* – no longer treat software design as a clearly and rigidly defined constant that is defined at the beginning of a development project. Instead they assume that a software design emerges step by step during the development process. If it is continuously adapted and improved to meet present requirements, it is called *emergent design*. Design improvements become established as an important and independent activity during development and evolve into an integral part of this process. This activity is called *refactoring*.

Refactoring

First of all, refactoring means changing the internal structure of a software to make it easier to read and modify without altering its observable behavior. Besides acknowledging this rather technical definition, many developers also associate a process-related aspect and a certain attitude with the refactoring term. In the context of extreme programming, refactoring means first and foremost an ongoing and repeated reflection about the software's structure and improving it in small increments.

Refactoring Catalogues

In his book on refactoring (see [Fowler 99]), Martin Fowler gives much advice on how refactorings can be accomplished. In this book he refers to very basic modifications of an object-oriented system, like, for example, 'Rename Method' or 'Encapsulate Field.' For each of these very small refactorings he describes – besides other aspects – also the 'mechanics' of a refactoring. The mechanics of a refactoring describe very small steps neccessary to perform the refactoring. Small increments ensure that the system remains operable at any given time. This procedure reduces the risk of introducing errors, created by unwanted side-effects, into the software during refactoring. In addition to the book, Martin Fowler's refactoring homepage provides a comprehensive list of refactorings.

Based on the refactorings depicted by Martin Fowler, Joshua Kerievsky identified further refactorings focusing on design patterns. These show how design patterns can be introduced to an existing system (or separated and removed from it), e.g. 'Introduce Observer' or 'Replace Constructor with Factory.' Kerievsky provides depictions of 'mechanics' similar to those of Fowler.

The descriptions of concrete refactorings, such as 'Rename Method' or 'Introduce Observer' are very valuable for developers, because they demonstrate when and how such a refactoring can be accomplished. Today, many development environments support developers quite efficiently during small refactorings.

1.1 Architecture Smells

Refactorings are often executed as a response to code smells. A certain portion of the source code 'smells like problems.' This is, for example, the case if the same code section occurs more than once in the system.

Besides smells on the code level, smells can also be identified on the architecture level, e.g. if the defined interface of a subsystem has been circumvented. We will provide a catalogue of *architecture smells*; some of which call for larger restructuring measures.

1.2 Large Refactorings

Theoretically it is thinkable to continuously provide an optimal system structure via small refactorings, but in practice, when dealing with complex projects, this is not realistic. Even projects involving skilled developers with a lot of know-how occasionally require larger restructuring measures of the system – large refactorings. Ron Jeffries' experiences confirm this observation:

"Our feeling is that if we could stick to our XP rules, we wouldn't need special taxes or special times to clean things up. But realistically, can you play our best game day in and day out?"[1]

In his book, Fowler also explains the necessity of large refactorings, called *big refactorings* by him and Kent Beck. Various examples for such big refactorings can be found in his book as well as on his refactoring website (http://www.refactoring.com/rejectedExample.pdf, Chapter 15: A Lon-ger Example). In many object-oriented development projects it poses a big challenge to handle these large refactorings.

Large refactorings often take longer than a day and change significant parts of a system. These properties of large refactorings create a number of problems that the developers will have to deal with. Among others, we observed the following problems:

- Developers 'lose track' of large refactorings, because they are created over a long period, and this process is frequently interrupted. They remain incomplete. As a consequence, the software's structure is in worse shape than before the refactoring.
- If a refactoring influences large parts of the system, a high demand for merges is often the result. This situation occurs when a refactoring is supported by an IDE and many parts of the system are altered at once; or when a big refactoring is not broken down into smaller increments. Such high demand for merges quickly discourages the developers' use of large refactorings. Thus much-needed design modifications will not be made.
- In many cases it is very difficult to foresee the consequences of single steps of large refactorings. Frequently during the execution of a large refactoring developers realize that the separate increments cannot be carried out according as planned. There is still no easy-to-handle means for dealing with such necessary changes of procedure.
- Because of the previously described difficulties, often large refactorings will not take place parallel to the normal system develop-

1. http://c2.com/cgi/wiki?TechnicalDebt

ment. Instead, the team puts the system's development process on hold for a certain period to solely focus on the large refactoring. This method of handling large refactorings does actually have more in common with reengineering than with refactoring. Also, many projects do not allow for temporary interruptions of development processes.

These are the problems we wish to discuss in the next chapters. The following questions will be at the center of our attention:

- How can large refactorings be broken down into smaller increments? Can large refactorings be assembled from small refactorings?
- How can large refactorings be planned? How can existing refactoring plans be adapted when it becomes clear that they cannot be realized as planned? How can one obtain undo-functionality for large refactorings, also during the actual refactoring process?
- How long can/may large refactorings take? How can I proceed to further develop (to add functionalities to) the system during the execution of a large refactoring? How can one make sure that the development process does not counteract the refactorings?
- How can plans for large refactorings be integrated in the development process? What type of development process is suitable here? Which prerequisites must the development process meet? How can/must/ should I document/communicate the present stage of a large refactoring?

1.3 Refactoring and Databases

Today, there is hardly an application system in existence that works without a (most common: relational) database which stores the objects of an application. If the storage structure of a class or the interaction of classes within the system is changed, this often means the database structures as well as data present in the database need to be accommodated too. Modifications of the database structure and the stored data have the reputation of being a complex and tedious task.

Many small or large refactorings can lead to frequent modifications of the system's classes. Since we do not expect the design to be established at the beginning of development, the database schema cannot be laid out at the project start. On one hand this means that refactorings of the program code can affect the database structures. The structures need to be refactored together with the code. On the other hand it may be necessary to additionally enhance the database schemata themselves and thus refactor them.

In this book we will show how refactorings affect a system's connection with a database.

1.4 Refactoring and Published-APIs

Refactorings do not alter the observable behavior of a software. The software is always treated as a whole. If, for example, we rename a method in a Java system, all occurrences of the original name in that system must be changed too.

Normally it is no problem to identify all references to a method name in a system and to adjust them accordingly. Many development environments will do this automatically. The simple renaming of a method will become difficult though if a system cannot be considered as a whole. Typically this is the case when a system provides an API that is also used by other systems. Such an API is also called a 'published API' as opposed to a 'public API.'

If a method, which is externally visible outside of a system's published API, is renamed, the IDE or the developer cannot adapt all existing references for this method, because a number of these references will lie within those systems that build on the published API.

As we can see, published APIs constitute a problem for an aggressive refactoring approach. In many cases this means that a modification of published APIs will be completely prohibited (or only be allowed to a very limited extent). As a result, not all refactorings of a system can be carried out, since some of them would alter the published API.

In this book we will address these problems and describe methods that will allow developers to integrate published APIs in their refactorings. At the same time we will aim at permitting *merciless refactoring*, even if this affects published APIs.

1.5 Reading Tips

Chapter 2 provides a brief introduction to the refactoring topic. Those readers who already have some practical experience with refactorings can skip this chapter.

Chapters 3 and 4 should be considered and read as a unit. They constitute the book's core.

Chapters 5, 6 and 7 can be read independently from each other. Developers who have experience working on large refactorings in projects will understand chapters 5 and 6 without having read chapters 2 to 4.

It is recommended that you read chapter 3 before you start reading chapter 7.

1.6 For Whom Was this Book Written?

This book primarily targets developers who have had some first experiences with refactorings and are familiar with the concepts Martin Fowler presents in his book. For all others there is a brief introduction to the topic at the opening of the book.

1.7 The Background of this Book

The book conveys experiences with specific refactoring situations and offers readers a variety of tips as well as assistance for how to use these refactorings in their own development projects.

The book is in part based on our own development project experiences, but also to a large extent on discussions with other developers, which took place on mailing lists, but also at conferences or workshops

1.8 Acknowledgements

Repeatedly we discussed our problems and insights with other people and tested them in projects. Therefore we would like to thank all those who supported us, who participated in discussions and provided valuable ideas and suggestions. Our special thanks go to:

- The employees and partners of its-wps GmbH for their committed collaboration on a number of projects.
- Walter Bischofberger and Henning Wolf, whose work with the *Sotograph* generated important input for the chapter on architecture smells. They also read early texts for this book and gave us much appreciated feedback.
- Marcel Bennicke has analyzed *Eclipse* with the Sotograph and allowed us to publish the results. You will find them in the chapter on architecture smells.
- The participants of the workshop on Large Refactorings at the OT 2003 conference. During our discussion, they relayed important and very interesting experiences, which further motivated us to research this topic.

A number of authors have contributed their own articles to this book: Walter Bischofberger, Berrin Ileri, Dierk König, Klaus Marquardt, Jens Uwe Pipka, Markus Völter and Henning Wolf.

We would like to thank the following persons (in alphabetical order) for their input regarding earlier drafts of this book as well as for their constructive criticism: Walter Bischofberger, Christoph Kögl, Claus Lewerentz, Klaus Marquardt, Torsten Mumme, Jens Uwe Pipka, Joachim Sauer, Bruno Schaeffer, Schmolitzky, Kurt Schneider, Marco Schulz and Robert Wenner.

1.9 References

[Brant & Roberts] John Brant & Don Roberts: *Smalltalk Refactoring Browser.* http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser.

The first tool for the support of refactorings. It enabled developers to realize many automated refactorings in Smalltalk and served as a blueprint for many integrated development environments where refactoring-support was pivotal.

[Fowler 99] Martin Fowler: Refactoring – Improving the Design of Existing Code, Addison-Wesley, 1999.

The standard work in refactoring. It covers the fundamental refactoring methods and is a standard tool for every developer.

[Kerievsky 04] Joshua Kerievsky: *Refactoring to Patterns*, Addison-Wesley Signature Series, 2004.

In this book Joshua Kerievsky addresses the question in how many steps patterns can be inserted in an OO system. The book is a consequent continuation of [Fowler 99].

[Opdyke 92] William F. Opdyke: *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

The first comprehensive work dealing with refactoring. It focuses on refactoring to push the development of frameworks.

[Refactoring 04] http://www.refactoring.com, 2004.

A site created by Martin Fowler that offers a collection of refactorings. Here, you will also find the refactorings from [Fowler 99].

[Roberts 99] Donald Bradley Roberts: *Practical Analysis for Refactoring*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1999.

This work is about the practical application of refactorings and analyzes how refactorings can be automated through the use of appropriate development tools. The implementation of the Smalltalk Refactoring Browser constitutes the basis of this work.

[Wake 03] William C. Wake: *Refactoring Workbook*, Addison-Wesley, 2003.

This book contains many practical tips how refactorings can be handled. It can also be used as a workbook for simple refactoring.

2 Refactoring - An Overview

This chapter provides an overview of the refactoring topic. To this end, we will first address the basic idea behind agile development methods, the idea that software is designed in a stepwise process (*Emergent Design*). This view is in opposition to the classic demand to create the entire software design prior to programming (*Big Design Upfront*).

Refactoring is the main instrument used in a step-by-step design process. A brief introduction of the basics will deal with the questions of when and how refactorings should be carried out. Then we will proceed to look at the relationship of refactorings and tests and discuss how modern refactoring tools are changing the present refactoring practice.

2.1 Emergent Design

The classic approach to software design is to come up with a complete design which will then be implemented. However, in the past years it became clear that this proceeding is rarely feasible. The design of a software must be repeatedly adapted and improved during development. Otherwise the software system will age, making it increasingly difficult to realize modifications. At some point, no developer will dare change the running system.

But if developers adapt the system design to meet current software requirements, the ageing process can be stopped and even reversed. In time, the software design can be improved.

2.1.1 Developing Software Is a Learning Process

For modern, evolutionary and iterative development processes, developers assume that software development is a learning process.

Whereas research results in this field strongly emphasize that it is a learning process for *all* those involved in a project, we will focus on the system's developers here.

The longer a project progresses, the more developers will learn about its requirements and the suitable software design. While some design choices which have been made in the course of the project will prove beneficial and correct, others will turn out to be wrong or awkward, the reason being that there is no such thing as a universal or best design for a software.

During the past few years, new approaches in the context of object-orientation were researched as well, and new findings were made regarding how certain design problems can be solved elegantly. At the same time, a software design is always created for a specific application type; depending on both the context in which the application is set and on the tasks it shall fulfill. If these factors change in the course of a project (and for evolutionary and iterative development processes it is assumed they do), the design must inevitably be adapted.

The opinion regarding software design and design modifications have changed due to these findings: Design changes are no longer considered a necessary evil or proof of mistakes; they merely document that software is able to meet the demands of changed prerequisites and will do so.

2.1.2 No Design, Simple Design, Emergent Design

If you consequently follow this train of thought, this implies that the developers don't need to present a precise idea of the design for the whole application at the beginning of a development project. Instead they should draft a rough design for the entire system, and a detailed design for the portion of the system which is currently in development. They should always make design adjustments and thus improve it. The application's design will thus evolve gradually.

One important prerequisite for an emerging design is that it is continuously adapted to the changing conditions. Developers should not ignore recognized weaknesses in the system's design, i.e. code smells. The rule is: the longer a smell exists in the system, the more difficult it will eventually become to eliminate. In a worst-case scenario this could mean that the developers do not refactor at all during development, but execute a redesign of the system at the end of a release cycle instead. With the ongoing refactoring of the source code, we choose to take the opposite route: refactoring and design will become parts of the daily development work. This does not mean that less designing takes place. The efforts are merely distributed more evenly over the whole period of the development process.

2.2 What Does Refactoring Mean?

Agile Methods and Refactoring

Today refactoring is an integral part of agile development projects. It is one of the tools a developer uses, just like a suitable programming language or an integrated development environment.

Refactoring means improving the technological design of a software without altering its observable behavior. The developers do not add any new features during a refactoring, i.e. they don't do any bug fixes or change anything about the software which would be detected by the software user. Instead, only the internal structure – the technological design of the software – is changed.

Creating a software design[1] is a challenging task. Besides comprehensive experience in software systems design, the developers first of all need to know precisely the respective software system's tasks and requirements to create a good design. Often it is not feasible to determine all requirements in advance, because:

- too much time passes before programming begins and the software can be utilized.
- the requirements are changing in the course of the project.
- misunderstandings arise, which will only be recognized and eliminated after the first couple of implementations.

In agile development processes the project participants even act on the premise that the project requirements will change during each iteration, that new ones will emerge and old ones might be eradicated altogether. The developers are forced to adapt the software design again and again – through refactorings. This is the only way to keep the software modifiable – 'soft' indeed. One might say that the software's aging is thus prevented.

2.2.1 An Example

An example[2] shall illustrate the underlying idea of the refactoring term. We developed a class *Movie* for a video store's rental system:

1. Design refers here exclusively to the software-technological design of a software, that is, its inner structure. It does not refer to the visual design of the user interface.
2. This example is taken from [Westphal 04].

```
public class Movie {
  static final double BASE_PRICE = 2.00; // Euro
  static final double PRICE_PER_DAY = 1.75; // Euro
  static final int DAYS_DISCOUNTED = 2;

  public static double getCharge(int daysRented) {
    double result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      result += (daysRented - DAYS_DISCOUNTED) *
                PRICE_PER_DAY;
    }
    return result;
  }
}
```

Because there are various places in the system dealing with amounts, these shall be calculated via a class of their own, labeled *Euro*, from now on. We introduce this new class and replace the constants of the class *Movie*.

```
public class Movie {
  static final Euro BASE_PRICE = new Euro(2.00);
  static final Euro PRICE_PER_DAY = new Euro(1.75);
  static final int DAYS_DISCOUNTED = 2;

  public static double getCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(
                 PRICE_PER_DAY.times(additionalDays));
    }
    return result.getAmount();
  }
}
```

Therefore, in the beginning, the new class *Euro* will only be used in the internal implementation of the class *Movie*. Consequently *Movie* shall not give out the amount as *double*, but directly as *Euro*:

```
public class Movie {
  static final Euro BASE_PRICE = new Euro(2.00);
  static final Euro PRICE_PER_DAY = new Euro(1.75);
  static final int DAYS_DISCOUNTED = 2;

  public static Euro getCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(
                 PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Unfortunately this modification leads to compile errors, because the clients of the *Movie* class for *getCharge* will continue to expect the return type *double*. Within a large system, this can create of hundreds or even thousands of compile errors at once. In order to make the refactoring process as pain- and risk-free as possible, it should be realized in small increments

Thus we will make sure that our changes of the class *Movie* will be carried out without rendering all client classes invalid. An often used method to accomplish this is the duplication of methods:

```
public class Movie {
  static final Euro BASE_PRICE = new Euro(2.00);
  static final Euro PRICE_PER_DAY = new Euro(1.75);
  static final int DAYS_DISCOUNTED = 2;

  public static Euro getEuroCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(
                 PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }

  /**
   * @deprecated
   */
  public static double getCharge(int daysRented) {
    return getEuroCharge(daysRented).getAmount();
  }
}
```

Now we got two methods with different names and a different return type that almost serve the same purpose. The compiler relays warnings to all clients that use *getCharge*, providing us with a to-do list for the conversion of the clients. Once all clients are using *getEuroCharge*, *getCharge* can be deleted from *Movie*.

If necessary, the method *getEuroCharge* can subsequently be renamed *getCharge*, either via method duplication or – much easier – with the aid of the development environment's refactoring support.

2.2.2 Refactoring Categories

Refactorings can concern various parts of a software system. In his book *Refactorings*, Fowler discriminates the following categories:

1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: *Extract Method*, *Inline Temp* or *Replace Temp with Query*.
2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like *Move Method*, *Extract Class* or *Remove Middle Man* can be found.
3. Organizing Data: These refactorings restructure the data organization. Examples are: *Self Encapsulate Field*, *Replace Type Code with Class* or *Replace Array with Object*.
4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as *Introduce Null Object* or *Decompose Conditional*.
5. Making Method Calls Simpler: These refactorings simplify method calls, such as *Rename Method*, *Add Parameter* or *Replace Error Code with Exception*.
6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as *Pull Up Field*, *Extract Interface* or *Form Template Method*.

For many refactorings, a reverse refactoring exists. For instance, a new method can be extracted if an existing method seems to be too long (*Extract Method*). On the other hand, a method can be dissolved if it has become obsolete (*Inline Method*). A similar strategy exists on the class level (*Extract Class*, *Inline Class*) or inside inheritance hierarchies (*Pull Up Field/Method*, *Push Down Field/Method*).

2.2.3 Observable Behavior

Refactorings Do Not Change a Software's Observable Behavior

If developers carry out a refactoring and thus change the software's structure, the software's observable behavior should not change – one could also say that refactorings do alter a program's syntax, but not its semantics.

When developers carry out a refactoring and thus modify a software's structure, its observable behavior should not change. Opinions regarding interpretation of the term 'observable' vary though. Strictly speaking, each single refactoring influences a system's dynamic behavior, but usually these changes are merely marginal. The difference would be measurable, but normally go unnoticed by the system user. A run-time change in the tenth-of-a-second range would be considered 'not observable' in most applications.

The question what exactly observable behavior is cannot be answered independently from the system and its application context. Pragmatically, one can settle for the definition that observable behavior has changed when the system user notices it.

2.2.4 When Is a Refactoring Carried Out?

Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as *bad smelling code* – so-called *code smells*. Bad smells often emerge when the so-called *Once and Only Once Principle*[3] has been disregarded: each design choice shall be expressed exactly in one place in the system.

A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. For a more comprehensive listing and depiction of common *bad smells*, see [Fowler 99].

Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification.

Most code smells can be cured with the appropriate refactoring. A method that is too long, for instance, can be broken down into many smaller methods with the refactoring *Extract Method*.

When developers detect a code smell, it can be eliminated with the aid of a refactoring at various stages of the project:

- Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software.
- After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range.

In many cases both methods are used, so that the following programming mini-cycle is created:

3. Here is a corresponding joke about that principle: The *Once and Only Once Principle* contradicts itself because the word 'Once' occurs twice.

1. Cleaning up the code based on the new requirements – with refactorings.
2. Implementation of the changes. If this turns out to be a complex task, refactorings will be used during implementation.
3. Cleaning up the new code – of course with refactorings.

2.2.5 How Is a Refactoring Carried Out?

Refactorings will alter an executable software. This always implies a risk, because there is the chance that new errors will find their way into the software. Therefore two axioms, which should be observed for refactorings, have been established:

- Refactorings are always to be broken down into small iterations that constitute complete and testable entities.
- Refactorings must only take place after the required automated unit or acceptance tests have been conducted. With these tests developers check if the software displays the same behavior as it did prior to refactoring.

While we are going to discuss the second axiom in greater detail later in this chapter, we will now deal with the first axiom, which states that refactorings should be broken down into small iterations.

Newcomers to refactorings show a tendency to bundle many small restructurings and implement them in a single, big step. Instead of dissecting only one method at a time, a superclass is created simultaneously, some parameters are complemented, a float value is packed into a value object, and two other classes are combined. Quite frequently developers get lost in the growing jungle of structural changes. The result is a system that will not be executable for a long period and which is difficult to get running again. Often new errors will sneak into the software. Due to the number of parallel introduced modifications, they are easily overlooked.

But if a refactoring is executed step by step, significantly smaller changes of the system can be committed back to the code repository, each of which contributes to a fully functional system. The risk of introducing new errors into the software will clearly be reduced, because the single alterations are straightforward and separately testable. Also, the risk of merge conflicts, because other developers have changed the same classes, is reduced.

Even a seemingly simple refactoring has the potential of influencing substantial parts of a system. If, for instance, the developers rename a method, one of the consequences might be that substantial parts of the program can no longer be compiled. Even an apparently simple refactoring can lead to changes in many parts of a system. If the refactoring is carried out in one big step, the developers will spend a relatively long time finishing it – at least as long as they don't have a tool to support them. Also, the danger of making mistakes increases.

It is not always easy to break down a refactoring into small increments. At first sight, the renaming of a method seems to resist deconstruction. Once the method has been renamed, all references must be changed as well. In his book, Martin Fowler assigns so-called 'mechanics' to each refactoring (see [Fowler99]). They describe what steps are to be taken to execute a refactoring. For example, for renaming a method, the developer could proceed as follows[4]:

1. Create a new method with a new name and copy the implementation from the old method into the new method.
2. Compile.
3. Change the old method's implementation so that it calls the new method.
4. Compile and test.
5. Find all references to the old method and step by step change them into the new method. Compile and test after each modification.
6. Remove the old method.
7. Compile and test the system.

These mechanics show that even the renaming of a method can be carried out in at least four separate steps. After each step, the system can be compiled and tested. Even if the method is used in many places in the system, the developers can always check in modified versions of the source code into the shared repository. The step-by-step procedure as well as the tests guarantee that the system will remain functional at any given time.

Although today the renaming of a method is done automatically by many development environments, and thus is a job that a developer can finish within a few seconds (we will address this issue in a later section of this chapter), this example shows that in principle it is possible and useful to break down refactorings into many small increments.

Martin Fowler's book on refactoring provides the respective mechanics for the refactorings listed in his book. On one hand they can serve as instructions for refactorings, on the other hand they offer ideas for how refactorings can basically be broken down into small increments. Practice has proven that all refactorings can be treated in this way, even if it seems impossible at first sight.

4. This is a slightly simplified version of the mechanics used by [Fowler99].

2.2.6 “Detours”

Breaking down refactorings into small increments is no trivial task. Let’s have another look at the example from the previous section: The old method continues to exist, while the new method has already been implemented. Only when all references to the old method have been replaced by references to the new method, the old method will be removed. In this way the old method serves as kind of detour. The entire system stays functional, although parts of the code have yet not been adapted for the new method.

Such detours are a typical characteristic of mechanics for refactorings. The comparison with road construction is not too far-fetched: Here too, detours will be created to enable traffic to flow in spite of the ongoing construction work.

For the example above this means that the old method does no longer contain implementations of its own, but calls the method with the new name instead.

During a refactoring, detours will temporarily make the system more complex. In the example above two methods for the same task exist simultaneously during the refactoring process. Only after all references to the old method have been modified, the old method will be deleted and the desired structure has been realized. Therefore it is of utmost importance to complete refactorings and conduct only a few refactorings at the same time. If these rules are not observed, the system’s structure will deteriorate due to the many remaining detours.

2.2.7 Refactoring Catalogues

Like for design patterns, for refactorings the attempt was made to find and write down universal descriptions and instructions, which eventually became refactoring catalogues. These catalogues describe a number of essential refactorings, each with a brief explanation of when the respective refactoring should be used, and how it can be realized.

The standard catalogue for refactorings can be found in [Fowler99]. This catalogue describes in detail seventy-two refactorings for the restructuring of object-oriented constructs. Supplementing the book, Martin Fowler has put up an online catalogue with an extended list of refactorings on his refactoring website (http://www.refactoring.com/).

While all the refactorings depicted by Martin Fowler in his book focus on basic object-oriented concepts, Joshua Kerievsky has assembled a catalogue of pattern-based refactorings (see [Kerievsky03]). The refactorings in his catalogue are, for example, for adding an observer pattern (*Replace Hard-Coded Notifications with Observer*) or a composite (*Replace Implicit Tree with Composite*).

2.2.8 Practical Experience and Advice

- Read Martin Fowler's refactoring book completely and keep on using it as a reference. It contains many tips and ideas, a comprehensive refactoring catalogue, and it shows how refactorings can be broken down into small increments.
- Be open to the practice of executing refactorings in small steps. Admonish yourself again and again to follow the small steps.
- Even if it does appear too difficult or not feasible at all to break down each refactoring into small increments: Go ahead and try it!
- If you fail to break down a refactoring, carry out a little review afterwards. After refactoring you will know how you did it, which will quite often make you realize how you could have broken it down.
- Practice proves that one can always come up with small steps. One of the underlying ideas is to build a detour first and then tear up the road. This also implies that in the beginning the system will become a bit more complex. Therefore refactorings should always be completed. Never let refactorings drag on over a long period.

2.3 The Role of Tests

Automated tests play a significant role in refactoring. They serve to check again and again if the entire system works exactly how it did before single steps of a refactoring or a complete refactoring have been executed. This security measure ensures that developers run a much lower risk of introducing new errors into the software.

Of course this only works as long as the refactoring does not alter the interface of a class. As soon as the interface of a class is modified as part of a refactoring, the tests need to be adapted to the modified interface. This raises the question of how the tests can function as a safety net if we have to manipulate them ourselves.

There are two different approaches of how to deal with tests during a refactoring: either the developers conduct the actual refactoring first and then customize the tests (*Code-First Refactoring*), or the tests are modified prior to the actual refactoring process (*Test-First Refactoring*).

2.3.1 Code-First Refactoring

For code-driven refactoring, the developers will carry out the refactoring and use the still unchanged tests as a safety net. In the course of the refactoring the tests are customized to fit the new code structure.

For renaming methods this means: as long as the old method still exists, old tests of this class can be carried out without requiring modifications. During the refactoring process the old test class can be fitted to the new method. This must happen before the old method is deleted from the class.

Detours are beneficial during test procedures: they ensure that one can continue to use the old test classes, but latest when the detours have been removed, the tests must be adapted to match the new structure as well.

2.3.2 Test-First Refactoring

Alternatively the fundamental idea behind test-driven development can also be applied to refactoring.

In test-driven development the developers first write the test, followed by implementation of the class, until the test turns out to be successful. If we apply this idea to refactoring tasks, we will arrive at test-first-refactoring: The developers will first change the tests and carry out the refactoring afterwards. This will be done until the modified tests are running successfully. Here the tests serve as a kind of ‘target’ for the refactoring.

Whereas developers will test a new or altered functionality during ‘normal’ test-driven development, followed by its implementation, they’ll focus on the structure of the code during test-driven refactoring. If, for example, a too long method shall be broken down, the test for the new, extracted method will be implemented first. On this basis, the developers will modify the original method and extract the new method. The already modified test class enables them to immediately test the old as well as the new method.

Test-driven refactoring has the same advantages we can also witness during test-driven programming: the new code structure is designed and implemented with its exemplary use (for testing) in mind, while for the new structure a test is readily available etc.

2.3.3 Practical Application: A Combination of Both Approaches

In practice, both approaches will rarely occur by themselves, i.e. isolated. In most cases, developers will combine the two procedures.

For renaming methods, for instance, first a test for the new method is implemented within the existing test. This is accomplished by copying the test for the old method and changing the used method accordingly. Afterwards, the new method can be added to the code, and one can follow the mechanics described above. Finally the old method is deleted together with the test for the old method.

Even if a refactoring is completely automated through its development environment (e.g. *Rename Method*), both approaches will be combined. The development environment makes sure that both tests as well as tested code are modified simultaneously.

2.3.4 Dependent Classes

In both cases, the test classes of dependent classes function as an additional safety net.

Let's have a look at the example of the renamed method: The class's clients will first call the old method. The clients' test classes indirectly check if the old method is still working. Step by step, all clients are adapted to the new method. Since these modifications only affect the implementation details of the clients, the clients' test classes don't have to be changed. They can also be used as a safeguard for the clients' modified versions. Thus the developers can automatically check if they made a mistake when they manipulated the clients.

This procedure will only work though as long as the dependent classes don't use any Mock, Stub or Dummy objects. In that case, integration tests must be utilized.

2.3.5 Refactoring of Tests

Test classes too need to be refactored from time to time. They are prone to the same code smells that we might 'scent' in the application's normal code. For 'normal' refactorings we used the test classes as safety nets to prevent the introduction of any new errors into the software. What can serve as our safety net though if we are going to refactor the tests themselves? After all, here too, we can make mistakes.

The answer is simple: The class to be tested will serve as our safety net. We proceed on the assumption that the test class did run successfully prior to refactoring. If a test within the test class fails after the test class has been refactored, an error must have been made during refactoring (or we found a new error in the class to be tested).

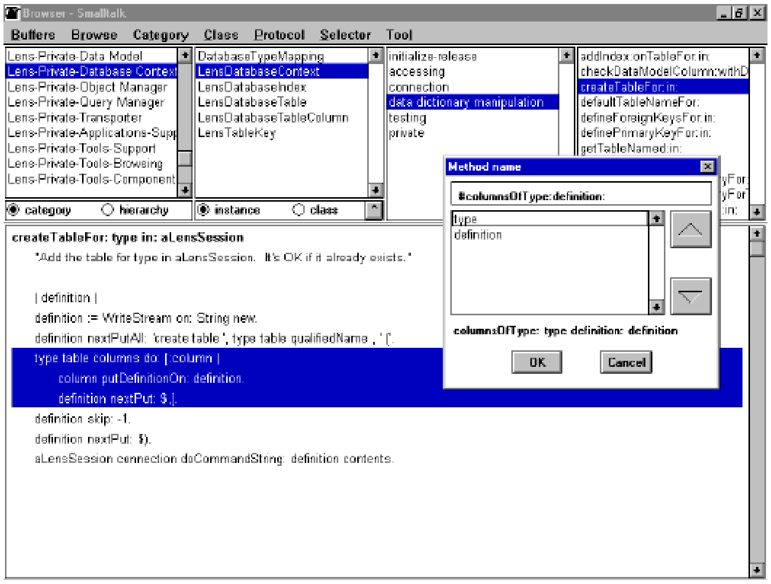
In addition, we can use test coverage tools (e.g. [JCoverage 04], [Clover 04]) to check test coverage before and after refactoring. However, it is only in part possible to let the test coverage tools check the same functionality after refactoring as before, and it requires a lot of tweaking. This is because such testing simply isn't the primary purpose of test coverage tools.

2.4 Tools Support for Refactorings

Refactoring tasks can effectively be supported through the use of suitable refactoring tools. The first tool specifically for refactoring was the *Smalltalk Refactoring Browser*, which was developed by John Brant and Don Roberts at the University of Illinois at Urbana Champaign. With this tool, many fundamental refactorings can be carried out automatically.

The refactoring browser for Smalltalk (see Figure 2-1) offers the option of renaming a class, for example. If the developer assigns a new name to a class with the aid of this function, all references will be automatically updated to match the new class name. The developer no longer needs to manually update clients of the respective class. The same can be done for renaming methods. The refactoring browser also enables the extraction of a method. To achieve this, the developer only needs to highlight the code section that shall be extracted and assign a name to the extracted method. The code will then automatically be copied into a new method with the assigned name and replaced by a call in the original method. Also, the refactoring browser automatically determines which parameters and return values are required by the new method.[5]

5. The Smalltalk refactoring browser also supports a variety of other refactorings. We will only introduce a few of them to illustrate the principal handling of this tool.



```
createTableFor: type in: aLensSession
    "Add the table for type in aLensSession.  It's OK if it already exists."

    | definition |
    definition := WriteStream on: String new.
    definition nextPutAll: 'create table ', type table qualifiedName , ' ('.
    type table columns do: [:column |
        column putDefinitionOn: definition.
        definition nextPut: $,].
    definition skip: -1.
    definition nextPut: $).
    aLensSession connection doCommandString: definition contents.
```

```
#columnsOfType:definition:
columnsOfType: type definition: definition
```

Fig. 2-1
Smalltalk Refactoring Browser

The refactoring browser for Smalltalk has significantly changed the thinking about and work with refactorings. In the meantime, many integrated development environments have started to offer similar functionalities. Especially current Java-IDEs, like IntelliJ IDEA or Eclipse (see Figure 2-2), offer powerful refactoring support. Their implementations have long surpassed the original refactoring browser for Smalltalk.

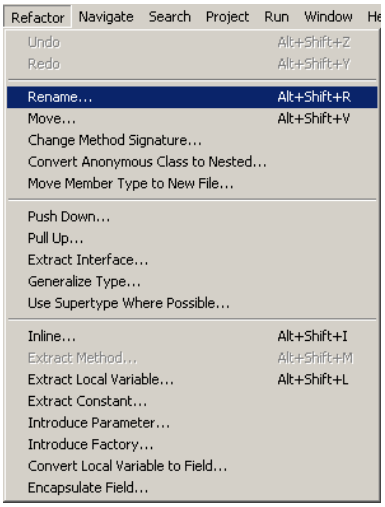

Refactor Navigate Search Project Run Window He
Undo Alt+Shift+Z
Redo Alt+Shift+Y
Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature...
Convert Anonymous Class to Nested...
Move Member Type to New File...
Push Down...
Pull Up...
Extract Interface...
Generalize Type...
Use Supertype Where Possible...
Inline... Alt+Shift+I
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Introduce Parameter...
Introduce Factory...
Convert Local Variable to Field...
Encapsulate Field...

Fig. 2-2
The Refactoring Menu of Eclipse

It is interesting to observe how this tool support has changed the work with refactorings. Renaming a class, an interface, or an operation in the common IDEs is a matter of a few seconds. Only by pressing a couple of keys, the old name will be replaced in the entire system.

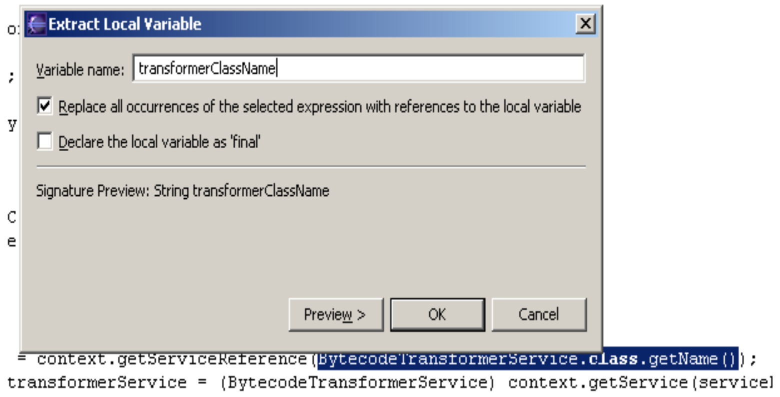
Tool support has advanced to the point of even correcting references to the respective name in source code comments or other files (such as XML files). However, this will only work with files that possess clearly defined semantics which are known to the refactoring tool. For a refactoring tool this is the only way of finding out if, for instance, a certain type is referenced or not. For JSP files (JavaServer Pages), for example, this can be easily done, because the semantics of the embedded source code are clearly defined. For a XML file it will be more difficult: here the IDE can only conduct a text search to find out if a certain type is referenced. If the type is not fully qualified (with complete package identifier), the refactoring tool will soon announce its defeat.

The number of supported refactorings in development environments grows with each new version. The current version of Eclipse allows developers the extraction of interfaces. Here the IDE does not only create the interface and lets the class implement it, moreover, all clients of the class are analyzed, and type references to the class are replaced by the interface where this is feasible.

Present research is one step ahead: researchers are trying to automate design pattern-based refactorings (see [Cinnéide 00]).

This shows that a growing number of and more complex refactoring operations are supported by IDEs, making it easy for developers to execute the desired refactorings. Refactoring is becoming a part of their daily work with source code.

Today it is possible to execute the conversion of an expression into a local variable in the source code with a single keystroke, using a refactoring provided by the IDE (see Figure 2-3).


Extract Local Variable

Variable name: transformerClassName

☑ Replace all occurrences of the selected expression with references to the local variable

☐ Declare the local variable as 'final'

Signature Preview: String transformerClassName

```
= context.getServiceReference(BytecodeTransformerService.class.getName());
transformerService = (BytecodeTransformerService) context.getService(service)
```

Fig. 2-3
Refactoring "Extract Local Variable" in Eclipse

2.4.1 Incremental Refactoring vs. Tools-supported Refactoring

A tool-based automation of refactorings seems to render the previously described mechanics and the step-by-step proceedings during refactoring obsolete. As a matter of fact, it is no longer necessary to rename a method in a series of single steps, because the IDE can accomplish this completely in a few seconds. Nevertheless, the basic idea behind incremental refactoring is not at all outdated.

There will always be refactorings that are either not supported by an IDE or that cannot be supported by an IDE (see next section). In these cases, it is still sensible to carry out refactorings step by step. Here too, the examples in Fowler's book can provide valuable advice how one's own refactoring can be broken down.

2.4.2 Limitations of Tools Support

Unfortunately refactoring tools have their limitations too. They cannot support all possible refactorings. In this section, we will take a brief look at some of these limitations and point to possible solutions:

- Refactoring tools can only provide automated support for such refactorings that can be generically described. This is the case with most refactorings introduced in [Fowler99], but the developer still has to manually combine several refactorings to form a composite refactoring. Refactorings like *Extract Hierarchy* or *Separate Domain from Presentation* cannot be executed automatically by today's software tools because they require too much context information. For example, in the case of the *Separate Domain from Presentation* refactoring, developers must decide which portions of the code belong to the application's domain model and which ones to the presentation-specific part.

- Refactoring tools rely on having the complete source code at their disposal, which will potentially undergo change through the refactoring. Only then the refactoring can be executed *safely*. If, for instance, a method that redefines a method from a library shall be renamed, the redefined library method must be renamed too to guarantee the same behavior. However, all popular refactoring tools will alert developers to such situations instead of blindly modifying the code. This problem does not only emerge when external libraries are used, but also when development takes place in different locations or when the system is developed in subprojects for one reason or another.
- If the application itself possesses a published interface for other systems, this interface can be changed with refactorings, but the interface clients will need to be refactored as well. We will dedicate an entire chapter labeled 'API-Refactorings' to this problem.
- This problem does not only emerge when external libraries are used, but also when development takes place in different locations or when the system is developed in subprojects for one reason or another.
- A regular refactoring can alter an application's source code. Persistent data will usually not be included in such an automated refactoring. Therefore it must be manually adapted to the application's new version. This problem is known for relational database connections as well as for purely object-oriented persistence mechanisms.
- If an object-oriented system uses a relational database, a mapping of object-oriented elements to the relational elements of that database is necessary. If a part of the object-oriented application is refactored, this can affect mapping to the database. This problem will also be discussed in a whole chapter.

2.5 Experiences and Recommendations

- Tests and refactorings constitute an inseparable unit. Only sufficiently automated tests guarantee the safe execution of a refactoring.
- If the interface of a class is changed in the course of a refactoring, the corresponding test class must also be adapted. In this case it is recommended that you first modify the test class and then proceed with the refactoring step (test-first refactoring).

- A design can emerge and grow in the course of a project. A rough outline of the architecture will often suffice in the beginning.
- However, do not make the mistake of working without any idea of what the architecture should look like – especially when you have either little or no experience with refactorings in projects. Otherwise you risk falling back on *hacking*.
- Refactorings are an essential part of software development. Only continuous refactoring will help to change and improve the software's design during development.
- Do not put off refactoring work. You can compare refactoring work to taking out garbage. If you don't regularly take out your garbage, you'll drown in it at some point.
- Use the refactoring options offered by modern development environments.

2.6 References

[Cinnéide 00] Mel Ó Cinnéide: *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. Thesis, Trinity College, Dublin, October 2000.

In his Ph.D. thesis, Mel Ó Cinnéide elaborates on how many of the well-known design patterns can be integrated in the code with refactoring techniques. Other than Kerievsky, Cinnéide is working on a tool-based approach that will enable the automated introduction of design patterns into the code.

[Clover 04] *Clover:* http://www.thecortex.net/clover

Clover is a commercial tool for measuring the test coverage of Java programs.

[FIT 04] *FIT:* http://fit.c2.com

FIT is a tool for the conduction of automated acceptance tests (also function tests). These tests are specified via HTML tables (e.g. using tables with input values and expected output values for certain system functions) executed by a test runner. Using fixtures, the test runner binds the application to be tested to the tables containing the tests. The test result documentation is then delivered in the form of HTML pages.

[Fitnesse 04] *Fitnesse:* http://www.fitnesse.org

Fitnesse is based on [FIT 04] and does not only offer FIT, but also a Wiki web that allows easier test specification and organization.

[Fowler 99] Martin Fowler: *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts, Addison-Wesley, 1999.

Not only does Fowler depict basic refactorings; he also introduces the distinction between public and published interfaces.

[JCoverage 04] *JCoverage:* http://www.jcoverage.com/

JCoverage is a tool for measuring the test coverage of Java programs. It exists in two versions: an open source and a commercial version.

[Kerievsky 03] Joshua Kerievsky: *Refactoring to Patterns*. To be published. http://www.industriallogic.com/xp/refactoring/

In his refactoring-to-pattern catalogue, Joshua Kerievsky consequently continues with Martin Fowler's work and describes how a number of popular design patterns can be treated during refactoring. The catalogue contains instructions for introducing a specific design pattern, but also a complementary refactoring for the respective pattern's removal.

[NoUnit 04] *NoUnit:* http://sourceforge.net/projects/nounit

NoUnit is a tool for finding program sections that are not supposed to be tested.

[Pipka 02] Jens-Uwe Pipka: *Refactoring in a „Test First" World*. XP 2002. http://www.agilealliance.com/articles/articles/JensUwePipka--RefactoringinaTestFirstWorld.pdf

This article addresses the problem that test code is often changed by refactorings, too, and therefore no longer applicable as a safety net for refactorings. We suggest a refactoring procedure similar to the test-first approach, i.e. to adapt the test first and then execute the refactoring.

[Roberts et al. 04] Don Roberts, John Brant, and Ralph Johnson: *A Refactoring Tool for Smalltalk*. Published in Theory and Practice of Object Systems special issue on software reengineering. http://st-www.cs.uiuc.edu/users/droberts/

A description of the Smalltalk refactoring browser.

[Westphal 04] Frank Westphal: *Testgetriebene Entwicklung mit JUnit und FIT*. dpunkt Verlag. Publication scheduled for 2004.

Westphal explains test-driven procedures in software development. Of course he also touches upon the issue of refactoring.

3 Architecture Smells

When experienced developers are looking at a system, they'll very soon develop a feel for its weaknesses. They will say that the system *smells*; it possesses distinct smells, which point to conspicuous states in the system. Whether these states really pose a problem or not must be decided in each individual case. If we follow a smell and actually detect a problem, we will solve it using refactorings.

In his book about refactorings (see [Fowler 99]), Martin Fowler describes smells that can be cured with small refactorings. Examples of causes for these smells are too long methods, long case statements etc.

Besides these code smells, architecture smells can frequently be identified. These will require large refactorings. The following sections will describe occurrences of architecture smells that we repeatedly encountered. As with code smells, an architecture smell does not always inevitably indicate there is a problem, but architecture smells point to places in the system's architecture that should be further analyzed. When we conduct architecture reviews, we refer to architecture smells for guidance.

Architecture smells can be found on various levels:

- **In uses- and inheritance relations between classes:** These smells refer to the elemental relations between single classes.
- **In and between packages:** For many programming languages, concepts for grouping related classes exist, for example the package concept in Java. In and between such packages architecture smells can also occur. We are going to address them here.
- **In and between subsystems:** Packages alone do not constitute a sufficient concept for the structuring of larger systems, which is why packages are often bundled in so-called subsystems. In and between such subsystems, architecture smells can occur.
- **In and between layers:** Besides subsystems, so-called layers are often introduced into larger systems. They also serve to structure

the system. Often these layers will serve to separate the UI model from the domain model. We have identified a number of architecture smells that can emerge *in* as well as *between* layers.

The larger a system is, the more important are analyses of subsystems and layers. In small systems, the interesting aspects of their architecture express themselves in packages and classes, whereas subsystems and layers often don't exist at all. Nevertheless, smaller systems too will become more clearly defined when they are divided into subsystems.

If the system is big enough to consist of a significant number of subsystems, it is more important to ascertain that the relations between the subsystems are clean than to ensure that the subsystems possess an optimal internal structure. If a chaotic structure exists within a subsystem, it will quasi be 'quarantined' by clear structuring of the subsystems – chaos cannot spread to the remaining parts of the system. Later on, the chaotic subsystem can be isolated from the rest of the system and either be revised or completely newly developed.

Of course it is also important to select the correct size for each level. Classes, packages, subsystems and layers should not contain too many, but neither too few elements. Figure 3-1 illustrates the resulting tension between understandability and reusability. The more components are part of a layer, the more of them can be reused by the layers above them. It should not go unmentioned though that the layer will become more difficult to understand as its number of components increases.

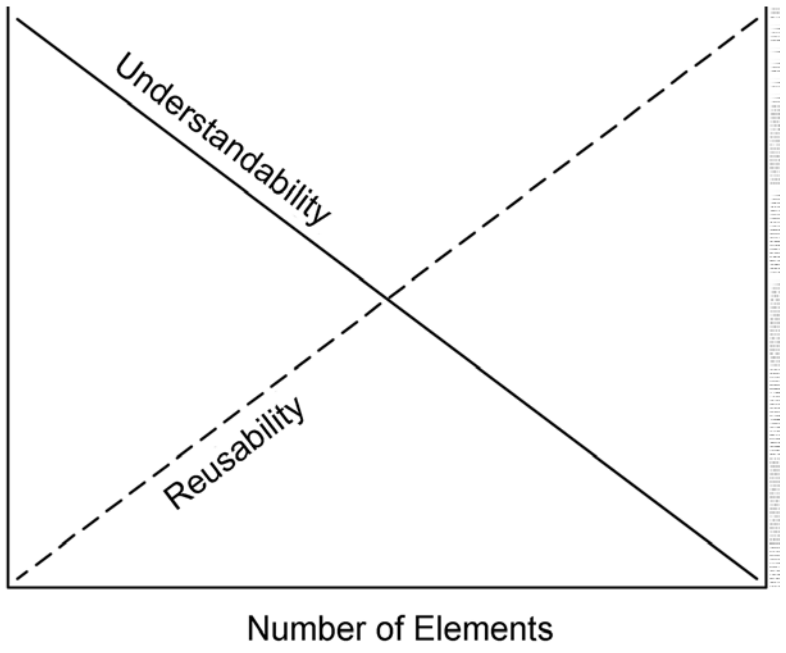


Fig. 3-1
Tension between Elements

Figure 3-1 obviously simplifies the relationship between understandability and reusability. In practice, the optimum between both values does not always meet at an intersection. Instead, there's rather a large 'middle zone'. Moreover, it might happen that reusability deteriorates along with decreasing understandability, because everybody avoids using items that are complicated or even not understandable at all.

It is not possible to provide general numbers, but there is a rule of thumb that can somewhat serve as a guideline: If an element consists of more than 30 subelements, it is highly probable that there is a serious problem:

a) Methods should not have more than an average of 30 code lines (not counting line spaces and comments).
b) A class should contain an average of less than 30 methods, resulting in up to 900 lines of code.
c) A package shouldn't contain more than 30 classes, thus comprising up to 27,000 code lines.
d) Subsystems with more than 30 packages should be avoided. Such a subsystem would count up to 900 classes with up to 810,000 lines of code.
e) A system with 30 subsystems would thus possess 27,000 classes and 24.3 million code lines.

f) If the system is divided in 3 to 10 layers, each layer comprises 3 to 10 subsystems.

In view of these numbers it soon becomes clear that one can carefully approach the upper limit of 30. If this is done consistently for each level though, the cumulative effect will lead to serious problems, as clarity might be impaired. Therefore, in practice the average values should stay visibly below the upper limit of 30.

The development environment Eclipse provides an apt example:[1]

a) Eclipse counts about 1.5 million lines of source code including line spaces and comments; without line spaces and comments it has about 730,000 lines of source code.
b) This source code is distributed over circa 460 packages, circa 12,400 classes and interfaces, plus approximately 89,500 methods.
c) This means that a method has an average of about 8 lines of source code – without comments and line spaces.
d) Classes and interfaces have an average of circa 7.2 methods.
e) Common packages contain about 27 classes and interfaces.
f) The Eclipse plug-ins can be viewed as subsystems. Consequently, Eclipse consists of 48 subsystems. A subsystem comprises an average of 9.6 packages plus 260 interfaces and classes.
g) Eclipse itself does not define layers. If one analyzes the static dependencies of subsystems, a layering consisting of ten layers can be identified (see Figure 3-2, provided by Marcel Bennicke), with approximately five subsystems assigned to one layer. The illustration shows that extreme variances occur: from layers with only one subsystem to layers with sixteen subsystems everything will be assembled here.

1. These numbers refer to Eclipse Version 2.1 with the plug-ins that are part of the software's standard package. We are grateful to Marcel Bennicke, who generously provided them.

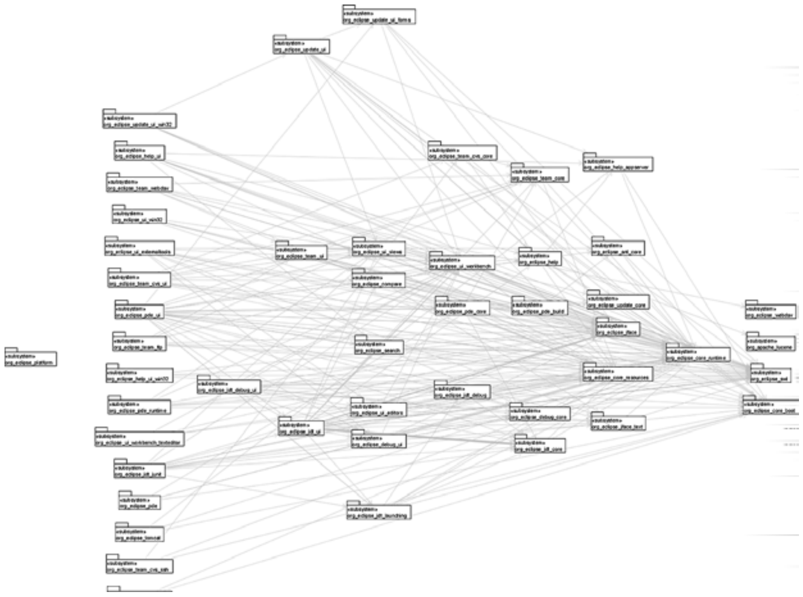


Fig. 3-2
Eclipse Subsystems: Vertical Layers

Figure 3-3 gives an overview of architecture smells in this chapter.

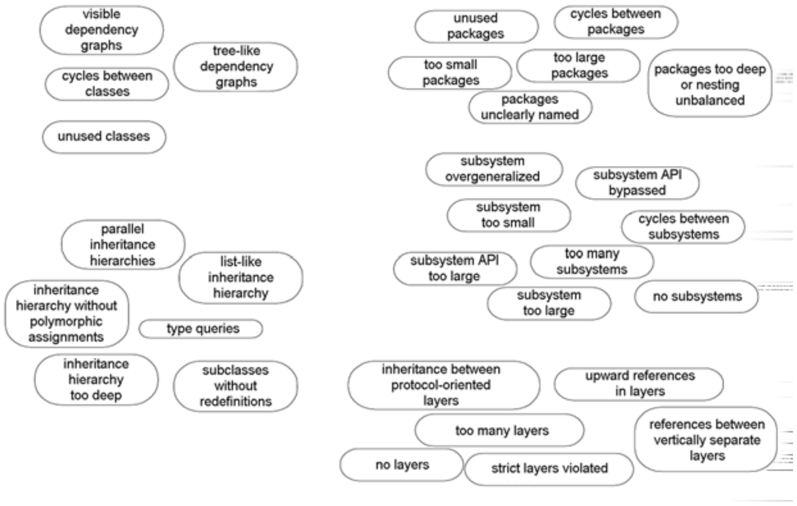

visible dependency graphs
tree-like dependency graphs
cycles between classes
unused classes
parallel inheritance hierarchies
list-like inheritance hierarchy
inheritance hierarchy without polymorphic assignments
type queries
inheritance hierarchy too deep
subclasses without redefinitions
unused packages
cycles between packages
too small packages
too large packages
packages too deep or nesting unbalanced
packages unclearly named
subsystem overgeneralized
subsystem API bypassed
subsystem too small
cycles between subsystems
subsystem API too large
too many subsystems
subsystem too large
no subsystems
inheritance between protocol-oriented layers
upward references in layers
too many layers
references between vertically separate layers
no layers
strict layers violated

Fig. 3-3
Architecture Smells

3.1 Design Principles

Like code smells, architecture smells are caused by a violation of recognized design principles. This is the reason why design principles can provide us with valuable tips for curing architecture smells. If the violated design principle can be identified, it will give us a first idea of how a better system structure might look like. Therefore, we will provide an overview of today's popular design principles in the following table (Figure 3-4).

Fig. 3-4
Design Principles

Principle	**Explanation**
DRY - Don't Repeat Yourself	Do not write the same or similar code more than once. Also called "Once and Only Once" principle.
SCP - Speaking Code Principle	The code should communicate its purpose. Comments in the code could indicate that the code communicates its purpose insufficiently.
OCP - Open Closed Principle	A design unit should be open to adjustments. Such adjustments shall not render existing clients invalid. Inheritance is one of the mechanisms that will let you achieve this goal: the subclass can make adjustments while the clients of the superclass remain valid.
LSP - Liskov Substitution Principle	One instance of a class must be usable for all instances where the type is the superclass. Not only it is required that the compiler translates the source code, but after the modification the system must still function correctly.
DIP - Dependency Inversion Principle	High-level concepts shall not depend on low-level concepts/implementations. The dependency should be vice versa, because high-level concepts are less liable to change than low-level concepts. One can introduce additional interfaces to adhere to the principle.
ISP - Interface Segregation Principle	Interfaces should be small. The methods of single interfaces should possess a high number of couplings.
REP: Reuse/Release Equivalency Principle	The elements that are reused are the elements that will be released.
CRP: Common Reuse Principle	The classes of a package are reused as a whole.
CCP: Common Closure Principle	The classes of a package shall be closed against the same type of changes. If a class must be changed, all classes of the package must be changed as well.
ADP: Acyclic Dependencies Principle	The dependency structure between packages shall be acyclic.
SDP: Stable Dependencies Principle	A package shall only depend on packages that are at least as stable as itself.
SAP: Stable Abstractions Principle	The more stable a package is, the more abstract it should be. Instable packages should be concrete.

Principle	Explanation
TDA: Tell, Don‘t Ask	Don’t ask an object about an object, but tell it what to do. Similar to the “Law of Demeter“: Each object shall only talk to “friends,” i.e. only to objects that it retains as fields or receives as parameters.
SOC: Separation Of Concerns	Do not mix several concerns within one class.

Fig. 3-4
Design Principles

3.2 Smells in Dependency Graphs

Classes can be coupled through use and inheritance. First, we will only deal with use. If we look at the uses relations between the classes of the system, we will see the *static dependency graph*. During system runtime this will result in the *dynamic dependency graph* between objects. In this chapter, we are only interested in the static dependency graph.

3.2.1 Obsolete Classes

Classes that are no longer in use will burden the system with obviously obsolete functionality. Not only single classes can be no longer in use, but also entire class graphs[2] (see Figure 3-5).

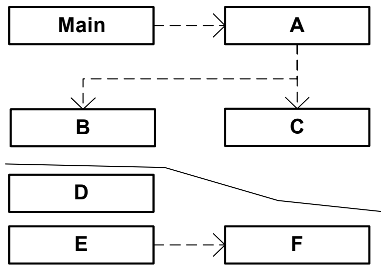


Fig. 3-5
Unused Classes: D, E, F

Unused classes mainly emerge for two reasons:

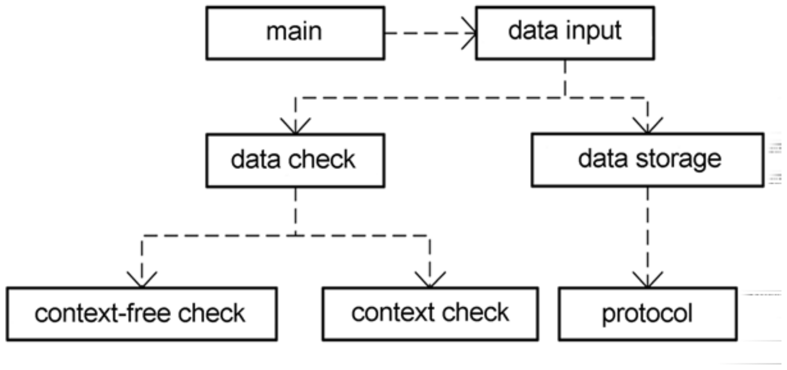
1. *Technology is hoarded as a supply:* A developer speculates that the class might eventually be used, although there is no evidence of a concrete demand for it.
2. *Refactorings:* A formerly required class becomes obsolete due to modifications of the system.

2. In our analyses, we will focus on complete applications. Naturally, obsolete classes can easily emerge in frameworks and libraries if they are only provided to service the client.

3.2.2 Tree-like Dependency Graphs

Tree-like dependency graphs (see Figure 3-6) indicate a functional decomposition of the system. Each class of the tree is used by exactly one other class. Reuse does not happen.

Fig. 3-6
The Tree-like Dependency Graph



3.2.3 Static Cycles in Dependency Graphs

Two classes using each other constitute the simplest imaginable cycle in a dependency graph (see Figure 3-7). Cycles can also include various classes (see Figure 3-8).

Fig. 3-7
A Cycle between Two Classes

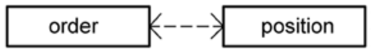
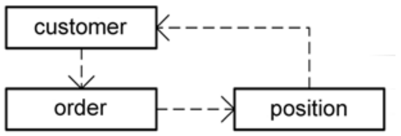


Fig. 3-8
A Cycle Including Various Classes



The presence of many cycles in a system will lead to its *lumping*. Cycles have negative effects on:

a) *Understandability*: The classes cannot be understood 'one after another,' because they presuppose each other to be understood. Instead, one has to alternate between classes to comprehend the graph in its entirety.
b) *Maintainability:* Cyclic dependencies can have severe and unpredictable consequences, thus making it harder to change the systems affected by them.

c) *Planability:* Cycles make it more difficult to anticipate the effects of changes. It will be more difficult to assess the effort required for and the complexity of a change.
d) *Clarity in design*: Often in one cycle each class can either directly or indirectly access any other class in the dependency graph. Therefore, in principle concerns can be arbitrarily distributed among these classes. The danger of placing methods in 'wrong' classes is considerable, which in turn makes it more difficult to comprehend the design.
e) *Reusability*: The class graph can only be (re-)used as a whole. If in a given context only one class from the graph is of interest, this class cannot simply be reused.
f) *Testability*: The classes can only be tested in their totality as a graph. This increases the demand for testing and error-searching. If one wishes to isolate classes during the test, relatively complex test patterns, such as Mock Objects (see References), must be utilized.
g) *Exception-Handling*: Often exceptions will accumulate in cycles. If some method in the cycle throws an exception, this event will potentially affect all other methods in that cycle.

Obviously longer cycles have much stronger smells than short ones. Especially cycles between exactly two classes can be desired – such cycles are even conditional for some design patterns (for example iterator, see References). Besides their length, interaction of the cycles is also of interest. If several cycles share the same classes, the situation will become much more complicated and soon lead to uncontrollable chaos. An impression of this constellation is given in Figure 3-9, although here 'only' the dependencies between packages are illustrated. If one tried to understand Swing in its entirety in order to modify it, no reasonable starting point could be found. Also, modifications at any point in Swing might result in side-effects in any other location in Swing.

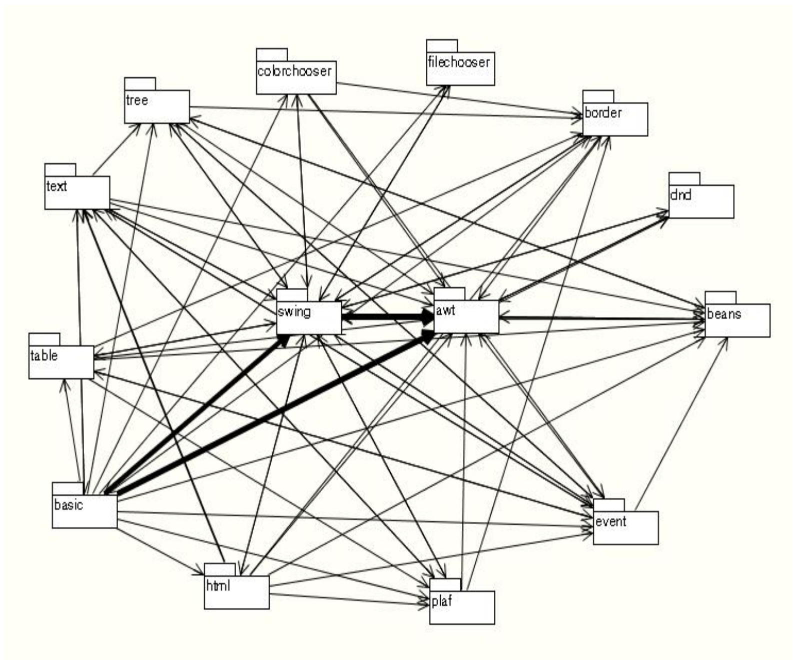


Fig. 3-9
Cycles in Swing-Packages

3.2.4 Visibility of Dependency Graphs

Object-orientation supports the principles of *encapsulation* and of *information hiding*: the internal implementation is hidden behind an interface. Many developers believe that encapsulation and information hiding will emerge solely because fields are declared private. This is not the entire truth though: In many systems it is possible for clients of an object to receive fields from the object via *get methods*. Based on the delivered objects, the client can continue to navigate. As a matter of fact, the dependency graph in the system is public and not at all hidden. A system with a public dependency graph will create more problems if one tries to change it, whereas changes to a private dependency graph will only have local effects.

The *Law of Demeter* (see References) as well as the *Tell, don't ask* principle (see References) are pointing in the right direction: ideally a client tells the used object what it is supposed to do. The client shall not accept another object from the used object, nor work with it.

Let us, for instance, imagine a number of orders in various states. We can especially differentiate between open or closed orders. Open orders are the ones in which the company has invested some money, but payment issues with the customer haven't been settled yet. Thus it is interesting to find out how much the total value of all open orders is.

If we spot a method *calculateValueOpenOrders* somewhere in the following form, the *Tell, don't ask* principle has been ignored:

```
public float calculateValueOpenOrders
    (ListOfOrders orders) {

  float totalValue = 0.0f;
  for (int i=0; i<orders.getNumber(); i++) {
    Order a = orders.getOrder(i);
    if (a.isOpen) {
      totalValue += a.getValue();
    }
  }
  return totalValue;
}
```

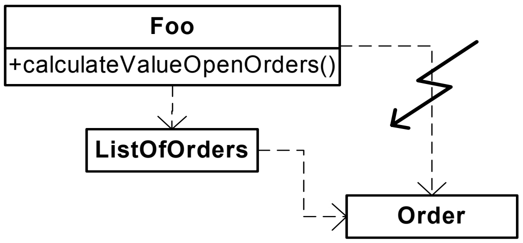


Fig. 3-10
A Violation of "Tell, don't ask"

In our second step, we move the case statement between open and not open orders into the class *Order* and get:

```
public class ListOfOrders {
  public float calculateValueOpenOrders() {
    float totalValue = 0.0f;
    for (int i=0; i<getNumber(); i++) {
      Order a = getOrder(i);
      totalValue += a.getOpenValue();
    }
    return totalValue;
  }
}

public class Order {
  public float getOpenValue() {
    if (isOpen()) {
      return getValue();
```

```
        }
        else {
          return 0;
        }
      }
    }
```

We might be unhappy about the fact that in this example the order returns the open value. If you decided to apply the *Tell, don't ask* principle one more time, you'd supplement the class *Order* with a method *addOpenValue* and remove the method *getOpenValue*. However, this would mean that the class *Order* would know that a certain number of orders exists. In this case, we would violate the *Separation of Concerns* principle.

Not only is this new implementation a bit shorter, it also possesses a number of additional advantages:

- The functionality is where it belongs. It is no coincidence that in the first example the name of the class containing the method *calculateValueOpenOrders* has not been mentioned. In most cases, such methods can be located directly in UI classes (e.g. *OrderEvaluatorDialogue*) or in help classes with bizarre names (e.g. *OpenOrders Calculator*).
- The *Tell, don't ask* principle ensures that types are only used locally, plus they are no longer distributed all over the system. Thus they will simplify the realization of optimizations.

What makes this smell so unpleasant is the fact that it cannot be found by merely taking a close look at the dependency graph. One must read the actual code to determine if many *get methods* exist, and if they are used in an undesirable way.

3.3 Smells in Inheritance Hierarchies

Classes are not only coupled through use, but also inheritance. The coupling via inheritance provides the advantage of polymorphism (the ability to adopt multiple shapes). During runtime, objects of different types can stand behind one identifier. This is made possible through polymorphic assignments. If an object is bound to a variable, this object must not necessarily possess this particular variable type. It is sufficient if the object's type is a subtype of the variable type.

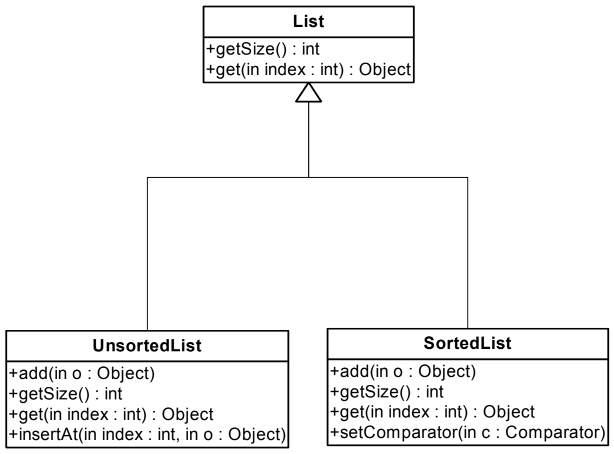
Inheritance also results in a closer coupling than use. Latest since the discussion about design patterns we know that in case of doubt use is preferable over inheritance: the classes will be coupled less closely, and the resulting structures can be used more flexibly.

This is why inheritance hierarchy problems are quite severe: due to the close coupling of the classes in the hierarchy, problems will be passed on from superclasses to their subclasses.

3.3.1 Type Queries

Each type query in the system (*instanceof*) constitutes a violation of the *Once and Only Once* principle:[3] The inheritance relation expresses itself not only in the classes of the inheritance hierarchy, but in the clients too. If alterations in the inheritance hierarchy are required, the type queries must be adapted as well (see Figure 3-11).

Fig. 3-11
Inheritance Hierarchy and Type


List
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

The following code snippet demonstrates how client code for this inheritance hierarchy could look like:

```
public void doSomething (List l)
{
  Customer k = new Customer();
  if (l instanceof UnsortedList)
  {
    UnsortedList ul = (UnsortedList) l;
    ul.insertAt(0, k);
```

3. Each design decision should be expressed exactly in one place, and one place only.

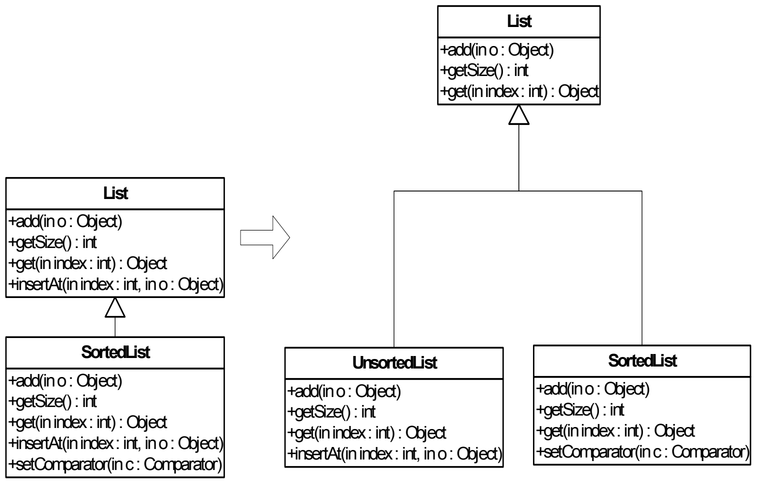
```
    }
    else
    {
      SortedList sl = (SortedList) l;
      sl.add(k);
    }
  }
```

Actually, the smell is quite helpful here. The type queries are almost begging for the developer to take a closer look at the inheritance hierarchy. Indeed the problem can be solved quite easily when the method *add* has already been implemented in both *List* and *UnsortedList* with the behavior depicted here.

A large system can hardly be realized without type queries. If a high number of type queries can be found in a system though, this indicates errors in an inheritance hierarchy. Maybe a subclass has been derived from another class, because the latter showed some similarity to it. As a matter of fact, a new superclass, from which both classes will inherit, should have been extracted first.

Figure 3-12 shows a popular example of a flawed inheritance hierarchy. It is plausible that a sorted list *is* a list and therefore permits the formation of subclasses, but the method *insertAt* has sneaked into the list class, and this method does not make sense in the subclass *SortedList*. Secretly it has changed the list to an unsorted list, and it is common knowledge that a sorted list cannot be derived from an unsorted list. Therefore a new superclass *List* must be created, which will combine the characteristics shared by sorted and unsorted lists alike.

Fig. 3-12
Wrong Inheritance Hierarchy



List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)
List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

3.3.2 List-like Inheritance Hierarchy

In a list-like inheritance hierarchy (see Figure 3-13) each class possesses a maximum number of one subclass. Such inheritance hierarchies either point to speculative generalizations or to too big classes.

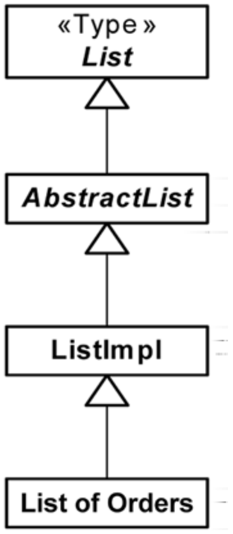


Fig. 3-13
List-like Inheritance Hierarchy

Speculative generalization means that superclasses were implemented for a definitely required class in hopes that the created abstraction might come in handy later on. This situation occurs quite often when the class hierarchy only consists of two classes.

Unfortunately, we cannot foretell the future and don't know for sure which abstractions will be needed later on. Experience has shown that speculative abstractions are not a good solution if an abstraction is actually needed. Frequently one will find that, for instance, wrong fields or methods were put in the superclass, or that the generalization is required in an altogether different place, or that the design problem can be solved much more elegantly with aggregation instead of inheritance. Usually, in such case the speculative structures need to be rearranged[4].

List-like inheritance hierarchies occasionally also emerge when classes become too long. Reducing the class's size through subclass formation is especially seductive for newcomers to object-oriented pro-

4. The fact that our observations are always based on complete applications also applies to this section, whereas in frameworks you will likely find superclasses and interfaces with possibly only one single implementation. This will particularly be the case if the framework uses the respective class or class hierarchy to realize a scheduled extension.

gramming: some methods will stay in the original class, while other methods will be put in a newly created subclass. This procedure is so tempting because hardly anything can go awry. Besides, it doesn't require too much thought.

The close coupling of sub- and superclass will indeed reduce the size of the superclass, but the subclass will actually stay too big. Its size is not only determined by its own methods, but by those inherited from the superclass too.

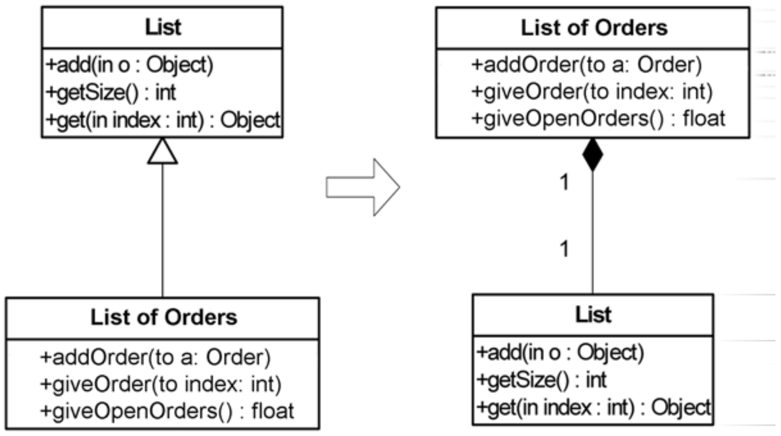
A subclass formation that is implemented due to the aforementioned motives will seriously impair the system's structure: the concept of inheritance is applied in an 'alien' context, which can seriously impede the understandability of the system.

One indication of too long classes is the absence of redefined methods inside the subclass (see next section).

3.3.3 Subclasses Do Not Redefine Methods

If subclasses don't redefine methods of their superclass, this can indicate that no abstraction is expressed through inheritance – we are facing pure implementation inheritance. Often a uses relation between classes will turn out to be more effective (see Figure 3-14).

Fig. 3-14
Implementation Inheritance: No Redefinition of Methods



List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
List of Orders
+addOrder(to a: Order)
+giveOrder(to index: int)
+giveOpenOrders() : float
List of Orders
+addOrder(to a: Order)
+giveOrder(to index: int)
+giveOpenOrders() : float
1
1
List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object

3.3.4 Inheritance Hierarchies Without Polymorphic Assignments

Similar to the previously mentioned smell, inheritance hierarchies without their respective polymorphic assignments point to the presence of unnecessary generalizations. The most significant advantage of inheritance as opposed to use is its flexibility, which is achieved through polymorphism. If no polymorphic assignments exist, this flexibility will not be used, and inheritance can be replaced by uses relations.

This smell is difficult to detect, because it only emerges when two situations are combined (inheritance hierarchy and assignments). A first indication of its presence is when too few assignments exist for the superclass type and too many for the subclass type, or when the superclass type is not much used in the system.

3.3.5 Parallel Inheritance Hierarchies

You can find parallel inheritance hierarchies in many systems because they are so beautifully symmetrical. For example, Figure 3-15 illustrates an existing domain-specific inheritance hierarchy between the business objects *Partner*, *Customer* and *Supplier*. Partners, customers and suppliers should be displayed on the UI level in list form. Thus, one view class exists for each of the three business object classes. These view classes inherit from each other according to the business object classes' hierarchy.

Last but not least, parallel inheritance hierarchies necessitate that one and the same design choice (namely that of the abstractions) must be expressed in two places. If a revision of this design choice needs to be made, all parallel inheritance hierarchies must be adapted.

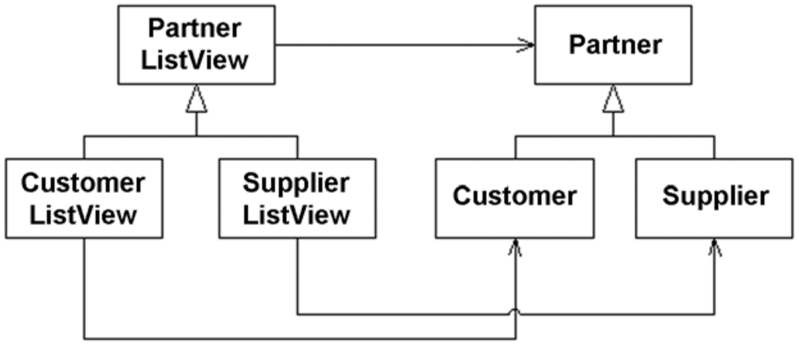


Fig. 3-15
Parallel Inheritance Hierarchies

In many cases, parallel inheritance hierarchies can be resolved in such a manner that only one inheritance hierarchy is left, while the classes of other inheritance hierarchies are integrated through use.

Figure 3-16 shows a modified version of the system from Figure 3-15. The views for customers and suppliers are using the view for partners now, which no longer inherits from them.

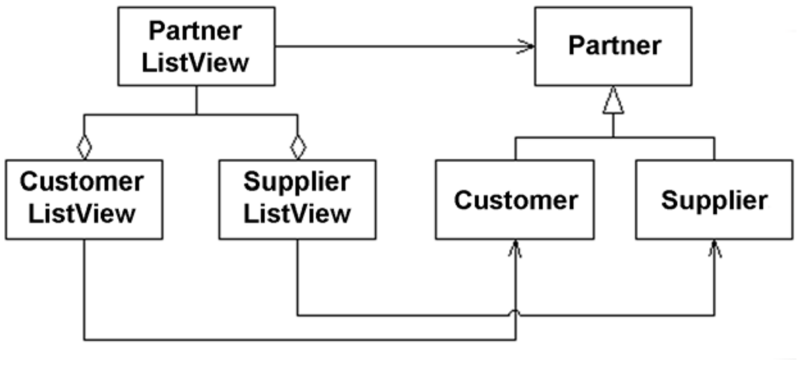


Fig. 3-16
Parallel Inheritance Hierarchies Have Been Removed

This smell is also mentioned in Martin Fowler's refactoring book.

3.3.6 Too Deep Inheritance Hierarchy

Especially in the pioneer days of object-oriented programming very deep inheritance hierarchies could be found in systems, because if the concept of inheritance is sound, more inheritance must be better than less inheritance.

In fact deep inheritance hierarchies can result in extremely flexible systems. Unfortunately, at the same time the system's understandability and the adaptability of its inheritance hierarchies suffers. If inheritance takes place across 10 levels, it is almost impossible to determine which implementation of a method is called by reading the code.

If the superclass needs to be changed, this does not only affect many subclasses. It is also difficult to project how this change will affect the classes at the bottom of the inheritance hierarchy (see Figure 3-17).

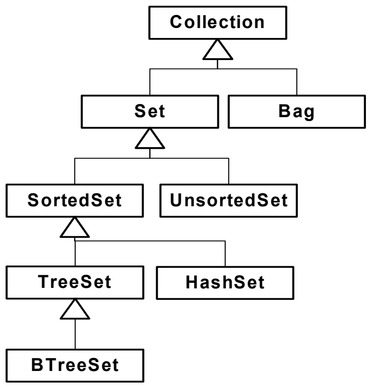


Fig. 3-17
Deep Inheritance Hierarchy

Inheritance hierarchies always demand careful planning. Deep inheritance hierarchies require a lot of attention to detail. This attention to detail will not always be delivered in a project's daily business routine. This is the reason why speculative and unnecessary generalizations are often found in deep inheritance hierarchies.

Easier to handle are shallow inheritance hierarchies, which tend to be broader or have been united in the formerly separate classes (see Figure 3-18).

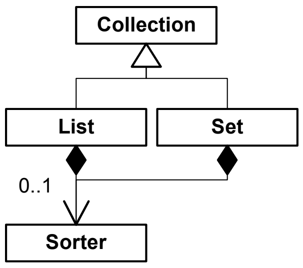


Fig. 3-18
Shallow Inheritance Hierarchy

3.4 Smells in Packages

Java and other programming languages offer concepts for the grouping of classes. We will call these groupings *packages* in keeping with Java terminology. A package can contain a number of classes.[5] The complete class name consists of the package name and the class name.

In Java, packages can be nested *syntactically*. For instance, the packages *java.util* and *java.io* are located in the package *java*, or respectively in packages called *java*. While specific visibility rules must be observed for classes within a package, this does not apply to nested packages. If one decided to rename the package *java.io* in *jio* and thus move it to the root level, this would not affect the classes – merely the imports would need to be adapted.

In programming languages without a package concept, usually file system directories will assume the role of packages. Naturally, in this case specific package visibility is no longer provided.

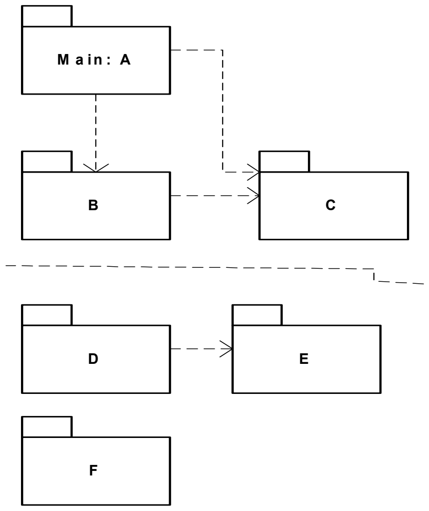
A few of the smells introduced here can also be found in [Marquardt 01]. In addition, this article also provides solutions for some package smells discussed in this chapter.

5. We will summarize interfaces in Java or comparable constructs in other programming languages here under the term 'class,' because this is simpler.

3.4.1 Unused Packages

Packages that are not in use burden the system with clearly obsolete functionality (see Figure 3-19).

Fig. 3-19
Unused Packages



Unused packages, like unused classes, are created primarily for three reasons:

1. *Technology hoarded as a supply:* A developer speculated that the package will be required later on, although there is obviously no need for it.
2. *Refactorings:* Modifications of the system rendered a formerly required package obsolete.
3. *Changed Requirements:* The package contains functionalities that are obsolete due to new requirements.

3.4.2 Cycles between Packages

Cycles between packages can be created through use, inheritance or through a combination of use and inheritance (see Figure 3-20).

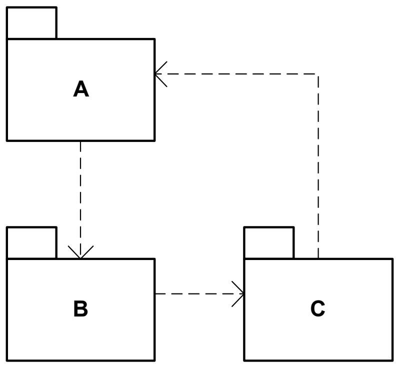


Fig. 3-20
Cycles between Packages

Cycles between packages have a less severe impact than cycles between classes:

a) *Understandability*: One cannot gain an understanding of packages through looking at them 'one by one,' because they presuppose each other to ensure understandability. Instead, one must skip between packages and perceive the package graph as a whole.
b) *Clarity in design*: The dependency structure of the packages results in first restrictions for permissible dependencies between classes. If packages are cyclically dependent, the permissible relations between classes can no longer effectively be restricted. Also, the assignment of classes to packages becomes less compelling. If each package can be accessed by any other package, it would in principle be feasible to place classes in any package, but this in turn would impede the design's understandability.
c) *Reusability*: As a rule, the package graph can only be (re-)used as a whole. If actually only one package from the graph is of interest in a given context, this package cannot be simply reused.
d) *Testability*: Packages can only be tested as a complete set. This leads to a higher demand for testing and error-searching. If one wishes to isolate packages during testing, relatively complex test patterns such as Mock Objects (see References) must be utilized.

Other than in cyclical relations between classes, *exception handling* is not impaired by cycles between packages.

Often cycles between packages point to poorly arranged packages. In most cases, this problem can be solved through simple restructuring, for example by merging all packages participating in a cycle into one package, which will then be arranged based on better criteria.

Cycles between packages will frequently lead to cycles between subsystems (see below).

3.4.3 Too Small Packages

Packages with one or two classes are often not worth the effort of introducing them: the complexity created by the package is not offset by its additional structuring.

Such too small packages can easily be removed through relocation of their classes to other packages. However, one must make sure that in this process no new cycles between packages are created.

3.4.4 Too Large Packages

Packages with a high number of classes can be handled much easier if they are broken down into several subpackages. This will especially lead to their better understandability.

Sometimes too large packages indicate missing subsystems. The creation of a subsystem from a too large package can solve this problem – for instance, if one splits the initially too large package into an interface package and one or more implementation packages.

3.4.5 Package Hierarchies Unbalanced

In most cases, the Java-JDK requires two-level packages (e.g. *java.util*), in some rare cases even three-level ones. Nevertheless, the JDK is able to somewhat usefully organize some thousands of classes.

Similarly to inheritance hierarchies, shallow package hierarchies are easier understandable than deep ones. To be fair one must concede that two-level package hierarchies in projects cannot be created if the convention for package naming is observed. According to this convention, the first three layers are taken by country ID, company name and project name, for example: *us.mycompany.myproject*. Generally, two to three layers below the first three should suffice for a system structuring.

If the package structure is unbalanced, understandability is also impaired. Should all of the application's business objects be located under *us.mycompany.myproject.bo*, the fact that not all UI classes can be found in a different place under *us.mycompany.myproject.ui*, but only packages containing subpackages instead, will cause confusion.

3.4.6 Packages Not Clearly Named

Especially packages containing classes that are not domain-oriented are often named ambiguously, and assigning of identical names occurs. If various packages with names like *util*, *base*, *framework* and *toolkit* can be found side by side in the same system and on the same level, it will be hard for developers to find the package containing the desired class right away.

Developers will face even greater difficulties when a new, not domain-oriented class shall be created. Its placement doesn't seem to matter. This uncertainty might lead to the idea of introducing another package – one that's equally vaguely named.

Ambiguously named packages frequently indicate that the developers had no real understanding of what's inside the packages, so it will come as no surprise if such packages contain classes with workarounds or were simply miscreated.

3.5 Smells in Subsystems

Similar to packages, subsystems summarize classes. They differentiate between internal realization and public interface. The internal realization is invisible for other subsystems. The public interface is comprised of a subset of the subsystem's classes.

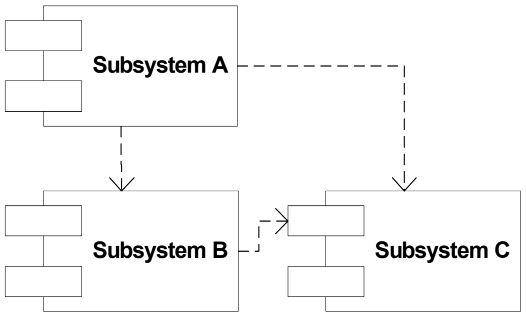
Packages also distinguish between public and private classes and methods. However, usually a single package will not suffice to define an entire subsystem. This requires a number of packages.

A large system should be divided in subsystems. This division will contribute to the system's learnability, maintainability, multi-project development and deployment.

a) *Learnability*: A first, superficial understanding of the system can be acquired if one looks at the subsystems and how they relate to each other.
b) *Maintainability*: Changes of a subsystem can be carried out in relative isolation from other subsystems. If a subsystem possesses a poor internal structure, this will affect the entire system.
c) *Multi-project development*: The development of single subsystems can take place in teams specifically assigned to that subsystem. Therefore, parallel programming is possible.
d) *Deployment*: If the system is not needed as a complete entity, single subsystems can be delivered.

e) *Testability:* Subsystems can be tested as isolated units. This also includes the option of defining and executing comprehensive and isolated test scenarios.

Fig. 3-21
Subsystems



In very large systems, the subsystem principle can be applied recursively, which will lead to a subsystem consisting of subsystems.

Unfortunately, the popular programming languages do not offer any options for the definition of subsystems. Often suitable runtime environments are applied to define and use subsystems. A mechanism based on the language Java can be found as part of the Eclipse platform's plug-in concept (see [Eclipse 03]). Similar runtime environments are, e.g., the DLL concepts, COM components or .NET assemblies.

If such a mechanism is not available, one must fall back on conventions, for example by using the root packages as a public interfaces of the subsystems and interpreting all subpackages as internal realization.

Some of the smells surrounding subsystems are caused by missing subsystem concepts in programming languages. This is, for instance, the case for the 'Subsystem-API Bypassed' smell.

Depending on the used terminology, subsystems are also called *components* or *plug-ins*.

3.5.1 No Subsystems

From a certain size on, a system's structure – if it is exclusively defined on the package level – will become increasingly incomprehensible. If the system consists of more than 100 packages, for example, it is extremely difficult to recognize and define the structure between the packages and to maintain it consistently.

3.5.2 Subsystem Too Large

The phenomenon that no subsystems are defined is a special case that occurs in too large subsystems. From the subsystems' perspective one could say that the entire system constitutes a single (too large) subsystem.

Like missing subsystems, too large subsystems run the danger of becoming incomprehensible and containing too many concerns. In many cases, the occurrence of very large subsystems is accompanied by a loss of clarity: the subsystem is no longer responsible for a single task, but it also takes on concerns in other areas.

3.5.3 Subsystem Too Small

Too small subsystems shift complexity from subsystems into the dependencies among the subsystems themselves (see Figure 3-22). In the most extreme case, each class represents its own subsystem. Obviously this will not lead to a reduction of complexity, instead developers are confronted with an impracticable tangle of dependencies between subsystems (see also: *Too Many Subsystems*).

Usually it is possible to merge too small subsystems into larger subsystems with little effort. However, developers must make sure that no cycles are created between these new subsystems.

3.5.4 Too Many Subsystems

If a system consists of many more than 30 subsystems without further grouping, the understandability of the system will be seriously impaired. This many subsystems and their interrelations can no longer be handled (see Figure 3-22).

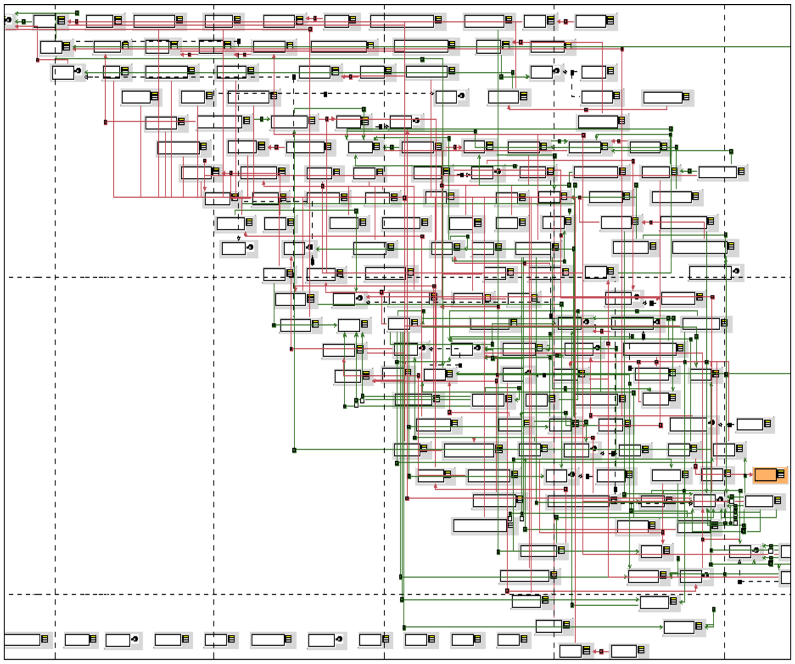


Fig. 3-22
Too Many Subsystems

In such a case, further subsystems that encapsulate the existing subsystems, should be defined.

Sometimes the subsystems were only created too small (see previous section). Here, merging the existing subsystems will solve the problem.

3.5.5 Subsystem-API Bypassed

Since the popular programming languages do not offer generic mechanisms for the definition of subsystems, projects must fall back on conventions. Consequently the subsystem's public interface – the API – will be defined through conventions.

Experience shows that such conventions are bypassed under pressure, e.g. lack of project time – either by mistake or on purpose. Bypassing the subsystem-API and directly accessing the internal implementation of the component is a practice that is not only common, but also potentially fatal (see Figure 3-23). The clients actually unauthorizedly expand the subsystem-API. The originally exclusively internal interface, which is now used by a client, becomes involved in the dependency relationship between subsystems. The result is the 'Subsystem-API Too Large' smell that we are going to describe in the next section.

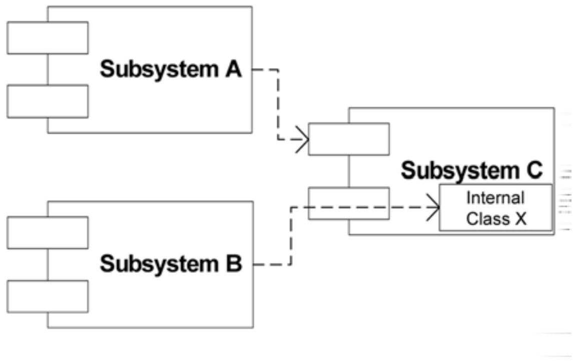


Fig. 3-23
Subsystem B Bypasses the API of Subsystem C

This scenario will have even more negative implications if the subsystem developers don't notice the API's expansion. Should they wish to alter or exchange the internal realization of the subsystem, this will have serious consequences for the clients that bypassed the subsystem-API.

Suitable Runtime Environments

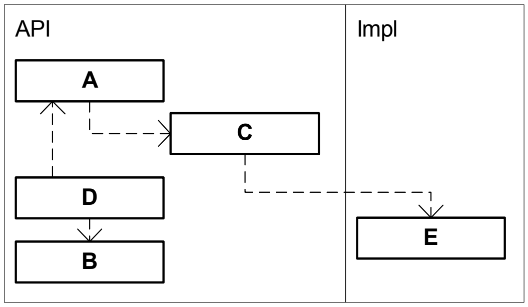
Such violations can be easily detected or even prevented with the aid of a suitable runtime environment. For instance, Eclipse Plug-in Runtime will let you declare the visible packages (public API) of a plug-in (subsystems). The runtime ensures that other plug-ins (subsystems) may exclusively use classes of those packages that have been defined as visible[6].

3.5.6 Subsystem-API Too Large

When the API of subsystem becomes too large in relation to the implementation, the main purpose of the subsystems is not served. A major part of the system will be visible to all other subsystems. Therefore, no significant complexity reduction has been achieved (see Figure 3-24).

6. In the case of a mistake, the corresponding *ClassNotFoundException* will be automatically released because a plug-in can only 'see' such classes via the classloading mechanism that have been declared public.

Fig. 3-24
Too Large Subsystem-API



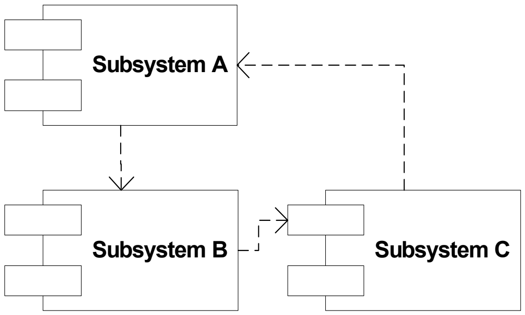
This smell can be detected by a simple means: one only needs to count the number of classes in the API and then compare the result to the total number of classes in the subsystem.

What kind of relation between API and implementation size precisely is useful heavily depends on the context, so that we can not provide any rule of thumb values here. Library-like subsystems, for example, will offer a very versatile API (e.g. a container library).

3.5.7 Cycles between Subsystems

Cycles between subsystems can be created via use, inheritance or through a combination of use and inheritance (see Figure 3-25).

Fig. 3-25
Cycles between Subsystems



At first sight, cycles between subsystems have less serious implications than cycles between classes:

a) *Understandability*: Subsystems cannot be understood by looking at them in a sequential order, because they presuppose each other to ensure understandability. Instead, one must skip between subsystems and perceive the subsystem graph as a whole.

b) *Clarity of design:* Often cycles between subsystems hint at unclear concerns of the subsystems. In many cases it is impossible to resolve immediately in which subsystem the wanted class is located, or where a new class could be sensibly placed.
c) *Reusability*: The subsystem graph can only be (re-)used as a whole. If, in a given context, only a single subsystem from the graph is of interest, this subsystem cannot be reused as a standalone, i.e. isolated from the other subsystems.
d) *Testability:* A subsystem cannot be tested isolated from the othersubsystems.
e) *Parallel development:* Cyclic dependencies between subsystems make the parallel development of subsystems by different teams and/or as part of different projects more difficult.

Exception handling is not affected by cycles between subsystems. Although cycles between subsystems create less problems quantity-wise than cycles between classes, they are much more problematic in practice. The relations between subsystems are an important aspect of software architecture and – contrary to cycles between classes – they cannot be cured locally. To achieve that, the system's architecture must be modified. If we are dealing with a large system, the APIs between the subsystems must be changed. However, sometimes the subsystems are maintained by different teams. In that case, the teams must coordinate their efforts.

Often cycles between subsystems point to unfavorably arranged subsystems. The problem can be solved, for example, by merging all subsystems participating in a cycle into a single subsystem, which then can be broken down based on better criteria.

3.5.8 Overgeneralization

In order to assure that subsystems provide the greatest extent of reusability, they must be flexibly applicable. This generalization can be overdone though, which will result in the subsystem's overgeneralization. It will become more flexible than it actually needs to be. Not only does this lead to additional subsystem development work; it also makes using the subsystem more difficult. Overgeneralization occurs when the clients – in relation to the size of the used subsystems – require a large amount of code.

Another indicator of overgeneralization is violation of the *Once and Only Once* principle. All clients of the subsystem write very similar code to parameterize the subsystem for its purposes.

This problem can be tackled by moving the always identical client code into the subsystem. Afterwards, the subsystem can be refactored internally, so that overgeneralization will not become an issue.

Of course the problem of overgeneralization can also be found on the methods, classes and packages level.

3.6 Smells in Layers

Besides the breaking down of a large systems into subsystems, the ordering of subsystems in layers has proven to be efficient. Each layer is assigned a specific aspect of the system. One of the most popular generic layer models is the 3-tier model (3-tier architecture, see Figure 3-26). It emphasizes that UI layer, domain model, and persistence should be kept separate.

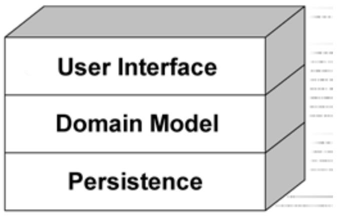


Fig. 3-26
3-tier Model

A more detailed layer model is that by Bäumer (see Figure 3-27). It distinguishes between three domain-independent layers: *system base*, *technology* and *handling & presentation*. The three domain model layers *business domain*, *business section* and *application context* build on the generic layers. In the business section layer different products are located, which may not depend on each other (see dashed line in the illustration). The stretched angles of the *technology* and *handling & presentation* layers indicate that the layering is not strict: all three domain model layers may use these two technical layers. More details regarding this layer architecture can be found in [Bäumer 98].

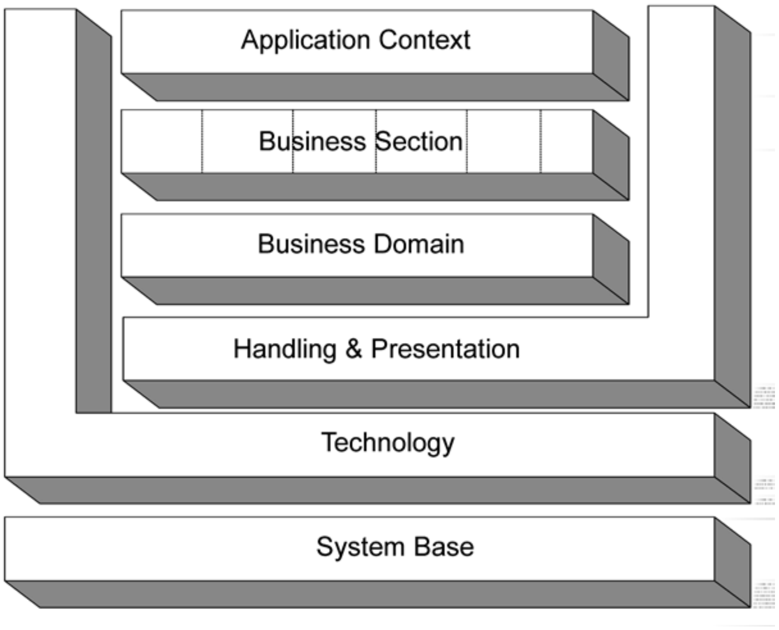


Fig. 3-27
Bäumer's Layer Model

Domain-specific layer models are the ISO-OSI model[7] for distributed systems (see Figure 3-28) or the layering of plug-ins in Eclipse[8] (see Figure 3-29).

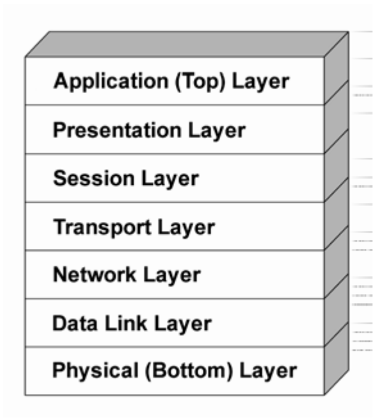
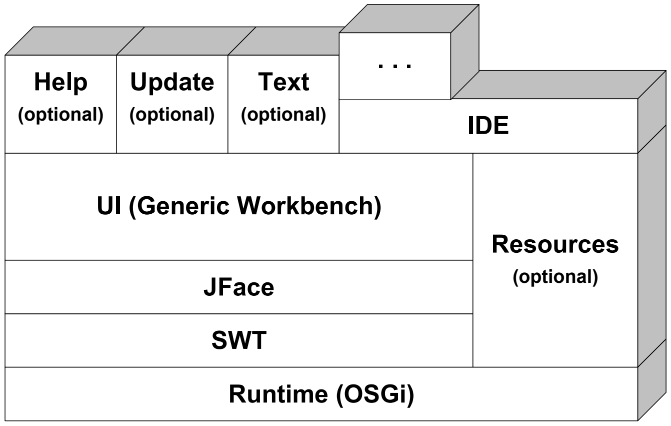


Fig. 3-28
ISO-OSI-Layer Model

7. Details can be found in [Kerner 89].
8. Although the documentation for the Eclipse platform does not mention any explicit layer architecture, the layering displayed inside the Eclipse platform and the projects building on it are easily recognizable.

Fig. 3-29
The Layers of Eclipse



The layer models assume different perspectives of the term *layer*. Consequently, the layers of the 3-tier model as well as those of the ISO-OSI model are *strict*: each layer may only access the layer directly below it. Layers that are located farther down cannot be used. Thus the UI layer of the 3-tier model is not entitled to directly accessing the persistence layer. Strict layers always apply the principle of information hiding: each layer conceals all layers below it.

If a layer is allowed to access other layers, accessing it through the layer directly below, it is called *non-strict*. Strict and non-strict layers can both be utilized within the same layer model: for instance, in the WAM[9] layer model the handling & presentation layer is non-strict, whereas the technology layer is strict and hides the system basis layer from other layers.

A second distinguishing criterion is the interface between the layers. For *protocol-oriented* layers, such as the ISO-OSI model, the interfaces between the layers are provided by functions. At the interface, no classes for building subclasses are offered. The APIs of *object-oriented* layers (e.g. in the Eclipse layer model) primarily include interfaces and abstract classes that either cannot or should not be implemented. In principle though, protocol-oriented and object-oriented layers can be mixed.

The major advantage of object-oriented layers is their flexibility, which is achieved through subclass-forming. Protocol-oriented layers, on the other hand, offer more flexibility in layer implementation and allow the use of non-object-oriented technologies for the realization of layers. This can be a huge advantage if relevant portions of the layer

9. The German acronym WAM stands for *Werkzeug, Automat & Material*, which translates into 'tools, machine & material.'

implementation already exist in a non-object-oriented programming language like Cobol or are built from purchased systems that do not possess an object-oriented interface. Thus, protocol-oriented layers allow a much simpler exchange of complete layer implementations.

In very large systems some layers are also separated vertically in addition to their horizontal separation, to define a so-called *product line*. One example of this practice is the business section layer in the layer model according to Bäumer. The single products in the business section layer are not allowed to depend on each other. Typically for such product line architectures, the separation of products is not applied to all layers: the lower layers are used by all products. Otherwise one would simply have completely separate systems.

3.6.1 No Layers

Demand for change can occur in different areas. Often layers are formed based on the large areas in which modification requirements emerge. For example, the 3-tier model uses the areas *user interface*, *domain model* and *persistence*.

This type of layering enables easy identification of those areas of the system that will potentially be affected by a change. For instance will a change of only the domain model not affect persistence.

If no layers exist, this kind of orientation aid is missing. Should the system consist of a large number of subsystems, it will be extremely difficult to identify potentially affected subsystems without layers.

For most systems, developers can name a layering that was intended. However, in many systems this intentional layering is violated so gravely that in the end no layers can be identified anymore.

3.6.2 Upward References between Layers (Cycles between Layers)

If a layer uses a higher located layer, the basic principle of layering has been ignored. Modifications of one layer cannot only have consequences for the higher layers, but also for those that are located further below (see Figure 3-30).

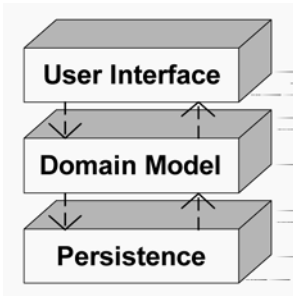


Fig. 3-30
Upward References between Layers

Simultaneously, upward references also create cycles between layers. They have similar effects as those created by cycles between subsystems (see *Cycles between Subsystems* smell) and might lead to the emergence of cycles on the subsystem level. Other than subsystems, layers make it comparatively easy to identify which relation is the one that is not permitted: namely always the one from bottom to top.

3.6.3 Strict Layers Violated

Since the common programming languages do not provide concepts for the definition of layers, layers must be built based on conventions. In this scenario, one cannot reliably prevent that strict layers are violated. It can always happen that a layer skips the one directly beneath it and accesses a layer further below instead, be it accidentally or on purpose (see Figure 3-31)

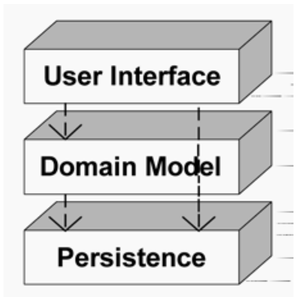


Fig. 3-31
Strict Layers Violated

If layers that are basically strict are violated, their alterability is affected. The number of a layer's potential clients will increase, and the dependency between layers will grow.

3.6.4 Inheritance between Protocol-oriented Layers

Inheritance between protocol-oriented layers is not allowed. Otherwise a stricter than desirable coupling would occur. In particular it would become impossible to re-implement the layer that inherited in a non-object-oriented programming language later on. Moreover, inheritance generally restricts the alterability of the lower layer, because changes to the superclasses can only to a certain extent be hidden from subclasses.

Recognizing Violations of Layers

Additional techniques allow the recognition of such layer violations at the time of development. AspectJ, an aspect-oriented language extension for Java, offers mechanisms for controlling method calls between layers. One example of such an aspect can be found in [Bodkin et al. 04].

3.6.5 Too Many Layers

Strict, protocol-oriented layers cause indirections: The supervisory data flow begins in the uppermost layer and proceeds downwards layer by layer. If a call results in a return value, the return value will be handed upward, following the same route.

The existence of many layers can create unnecessary indirections: one indication of unnecessary indirections are dumb delegations: one method simply invokes another method without implementing any functionality of its own. Whereas the single occurrence of a delegation is not necessarily to be considered bad, extensive use of delegations between many layers can point to problems. If many delegations exist, most likely a number of negative effects will follow in their wake:

- A lot of effort must be invested in the programming of methods without implementing any functionality.
- Program understandability will suffer.
- The ensuing modifications will require a lot of effort. Specifically modifications of parameter lists in a lower layer will impact all higher layers.

3.6.6 References between Vertically Separated Layers

We already discussed that layers cannot only be arranged horizontally, but also vertically. This is often done to structure separate products or business sections. For example, a product line is a set of software systems that share a common basis. Besides using the same basis, no further references between these systems are allowed.

References between vertically separated layers create dependencies between layers (see Figure 3-32). Thus the purpose of product lines can no longer be served:

- *Delivery:* vertical layers shall be deliverable and applicable independently from each other.
- *Parallel development:* for each single vertical layer one team shall be responsible, which does not have to confer with other layer teams regarding changes.

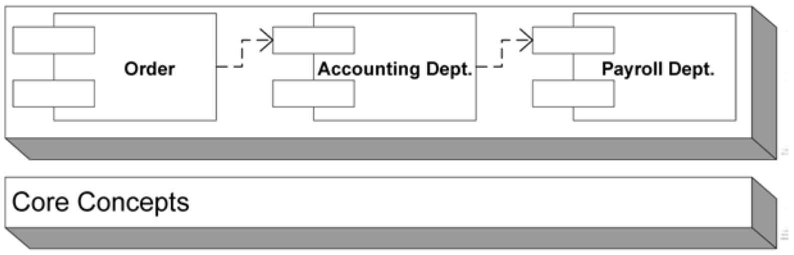


Fig. 3-32 *References between Vertical Layers*

If fundamental relationships between different products in the system exist, the described vertical separation between layers cannot be made. In this case, these basic relationships (the stable domain layer model) will be located in the core concept layer as represented in Figure 3-32, on which the vertically separated layers are founded.

3.7 Locating Smells

Reading the Code Is Not Enough

Architecture smells can hardly be found through simple code reading. After all, they usually emerge not from a single class, but from the interaction of many classes. Code reviews offer a good framework for the detection of smells, but even for code reviews a suitable tool support is mandatory to visualize the system. Whereas simple UML tools for code reviews will at least visualize the system on the package level, more specialized tools are required for the detection of numerous architecture smells.

Modern development environments (for instance Eclipse) offer powerful semantic search functions. Thus we can easily determine which classes inherit a specific class, or how many references to a method exist. In this way, hypotheses about smells can be verified. For example, if we suspect that only one subclass of class A exists, we can easily check this: we simply ask for a display of A's type hierarchy.

Unfortunately, these display options are not sufficient for the detection of architecture smells. If you don't know yet which classes are involved in a smell, you have no venturing point from which to start searching.

Besides development environments, a number of tools exist that can help detect common smells in software systems.

More Specific Tools

There is a number of tools that can help to find common smells in software systems. A brief overview of these tools is listed in the following sections (the URLs for these tools are listed in chapter 3-9).

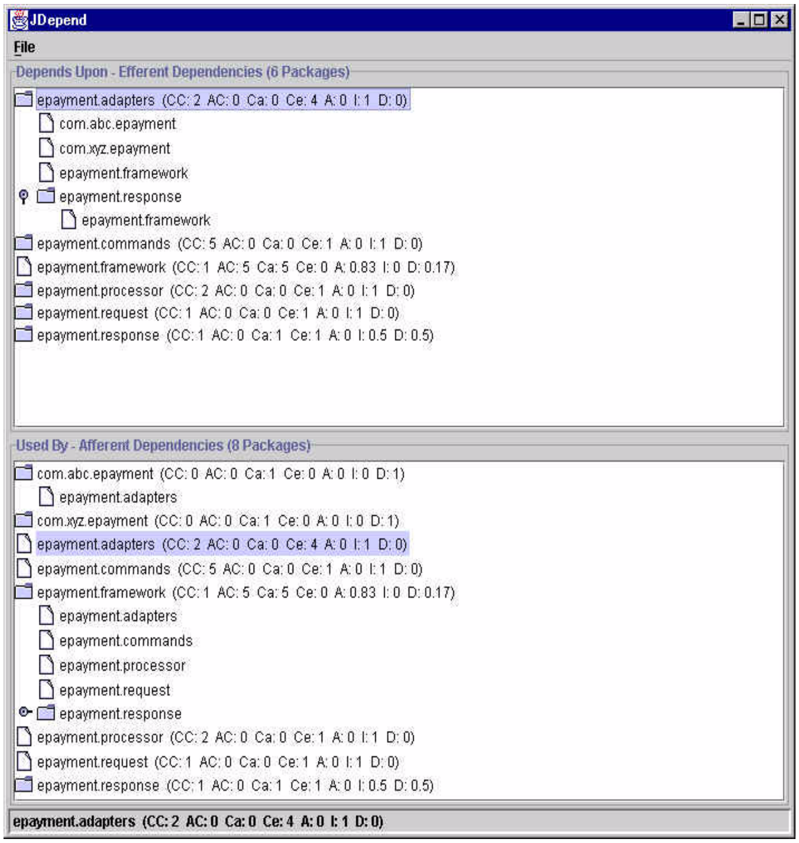
3.7.1 PMD

For Java systems, the open source tool *PMD* (see [PMD 03]) alerts developers to code smells such as empty catch blocks or unused methods. However, PMD analyses are restricted to only one class at a time and do not consider relations between classes. Thus PMD analyses are not sufficient for the identification of architecture smells. PMD is available as a plug-in for the popular Java development environments.

3.7.2 JDepend

JDepend (see [JDepend 04]) analyzes the dependencies between packages and classes and calculates Robert Martin's metrics (see [Martin 97]). JDepend possesses an interface for the display of dependencies and couplings (see Figure 3-33), but it also offers a programming interface. The latter enables, e.g., the formulating of JUnit tests that make sure that no unwanted dependencies are introduced into the system.

Fig. 3-33
JDepend: Dependencies of Packages



```
JDepend
File
Depends Upon - Efferent Dependencies (6 Packages)
epayment.adapters (CC: 2 AC: 0 Ca: 0 Ce: 4 A: 0 I: 1 D: 0)
    com.abc.epayment
    com.xyz.epayment
    epayment.framework
    epayment.response
        epayment.framework
epayment.commands (CC: 5 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.framework (CC: 1 AC: 5 Ca: 5 Ce: 0 A: 0.83 I: 0 D: 0.17)
epayment.processor (CC: 2 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.request (CC: 1 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.response (CC: 1 AC: 0 Ca: 1 Ce: 1 A: 0 I: 0.5 D: 0.5)

Used By - Afferent Dependencies (8 Packages)
com.abc.epayment (CC: 0 AC: 0 Ca: 1 Ce: 0 A: 0 I: 0 D: 1)
    epayment.adapters
com.xyz.epayment (CC: 0 AC: 0 Ca: 1 Ce: 0 A: 0 I: 0 D: 1)
epayment.adapters (CC: 2 AC: 0 Ca: 0 Ce: 4 A: 0 I: 1 D: 0)
epayment.commands (CC: 5 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.framework (CC: 1 AC: 5 Ca: 5 Ce: 0 A: 0.83 I: 0 D: 0.17)
    epayment.adapters
    epayment.commands
    epayment.processor
    epayment.request
    epayment.response
epayment.processor (CC: 2 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.request (CC: 1 AC: 0 Ca: 0 Ce: 1 A: 0 I: 1 D: 0)
epayment.response (CC: 1 AC: 0 Ca: 1 Ce: 1 A: 0 I: 0.5 D: 0.5)

epayment.adapters (CC: 2 AC: 0 Ca: 0 Ce: 4 A: 0 I: 1 D: 0)
```

Example

The following Java source code shows how you can test package dependencies in JUnit tests (source code borrowed from [JDepend 04]):

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class ConstraintTest extends TestCase {
    private JDepend jdepend;

    public ConstraintTest(String name) {
        super(name);
    }

    protected void setUp() {
        jdepend = new JDepend();
```

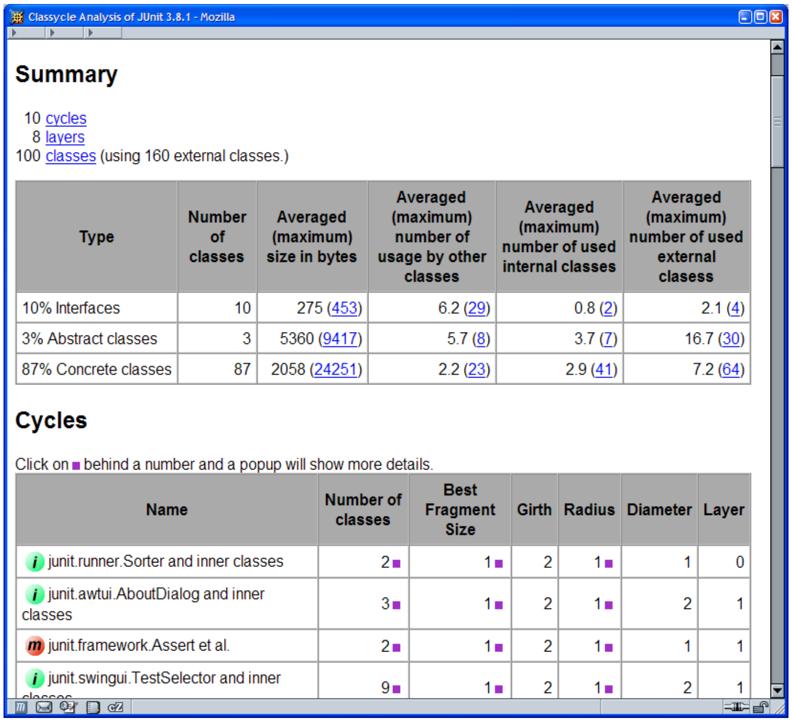
```
          try {
           jdepend.addDirectory
             ("/projects/util/classes");
           jdepend.addDirectory
             ("/projects/ejb/classes");
           jdepend.addDirectory
             ("/projects/web/classes");
          } catch(IOException ioe)  {
              fail(ioe.getMessage());
          }
      }

      protected void tearDown() {
          jdepend = null;
      }

      /**
        * Tests that the package dependency con-
straint
       * is met for the analyzed packages.
       */
      public void testDependencyConstraint() {
          DependencyConstraint constraint =
           new DependencyConstraint();

          JavaPackage ejb =
           constraint.addPackage("com.xyz.ejb");
          JavaPackage web =
           constraint.addPackage("com.xyz.web");
          JavaPackage util =
           constraint.addPackage("com.xyz.util");

          ejb.dependsUpon(util);
          web.dependsUpon(util);

          jdepend.analyze();

          assertEquals("Dependency mismatch",
          true,      jdepend.dependencyMatch(con-
straint));
      }

      public static void main(String args[]) {
```

```
                junit.textui.TestRunner.
                  run(ConstraintTest.class);
          }
    }
```

An Eclipse plug-in for JDpend is available (see [JDepend4Eclipse 04]).

3.7.3 ClassCycle

ClassCycle (see [ClassCycle 04]) is an open source tool for the detection of cycles between classes. The detected cycles are displayed either in a XML or an HTML report (see Figure 3-34). Based on the dependencies between the classes, layers are generated and classes assigned to layers.



Classycle Analysis of JUnit 3.8.1 - Mozilla

Summary

10 cycles
8 layers
100 classes (using 160 external classes.)

Type	Number of classes	Averaged (maximum) size in bytes	Averaged (maximum) number of usage by other classes	Averaged (maximum) number of used internal classes	Averaged (maximum) number of used external classes
10% Interfaces	10	275 (453)	6.2 (29)	0.8 (2)	2.1 (4)
3% Abstract classes	3	5360 (9417)	5.7 (8)	3.7 (7)	16.7 (30)
87% Concrete classes	87	2058 (24251)	2.2 (23)	2.9 (41)	7.2 (64)

Cycles

Click on ■ behind a number and a popup will show more details.

Name	Number of classes	Best Fragment Size	Girth	Radius	Diameter	Layer
i junit.runner.Sorter and inner classes	2 ■	1 ■	2	1 ■	1	0
i junit.awtui.AboutDialog and inner classes	3 ■	1 ■	2	1 ■	2	1
m junit.framework.Assert et al.	2 ■	1 ■	2	1 ■	1	1
i junit.swingui.TestSelector and inner	9 ■	1 ■	2	1 ■	2	1

Fig. 3-34 *ClassCycle: An Example of a Display Generated by JUnit*

3.7.4 Eclipse Metrics Plug-in

The *Eclipse Metrics plug-in* (see [Eclipse Metrics Plug-in 04]) is an open source metrics tool that has been realized as a plug-in for the Eclipse development environment. In a first step it supplies reference values that enable an evaluation of a software system's quality. One must keep in mind though that these values often refer to the code level (e.g. average method length) and therefore offer relatively little support for an examination of the architecture. Moreover, it often remains unclear *where* the problems are stemming from and what must be done to solve them.

At least the Eclipse Metrics plug-in is able to graphically display the relations between packages, so that one can detect one or another architecture smell, given a bit of patience and some knowledge about the targeted architecture (see Figure 3-35).

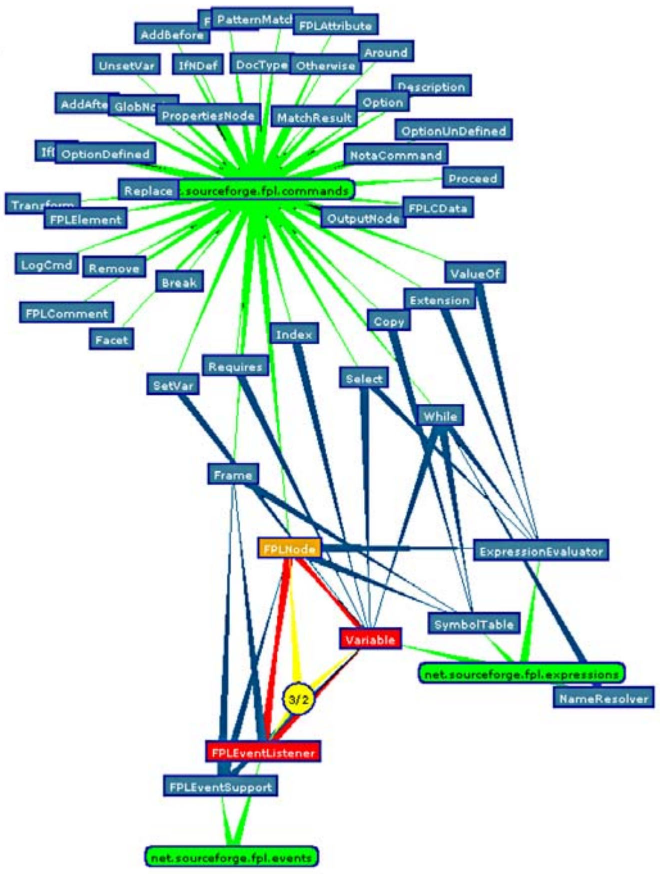


Fig. 3-35
Eclipse Metrics Plug-in: Package Overview

3.7.5 RefactorIT

RefactorIT (see [RefactorIT 03]) is a commercial refactoring tool that does not only support refactorings, but also the preceding step of detecting smells. To this end it provides the common metrics as well as some dependency analyses.

3.7.6 Dr. Freud

Dr. Freud (see [Dr. Freud 04]) visualizes dependencies between packages and classes (see Figure 3-36) and calculates Robert Martin's metrics (see [Martin 97]). Currently, Dr. Freud is still being developed, but it worked quite decently in our tests.

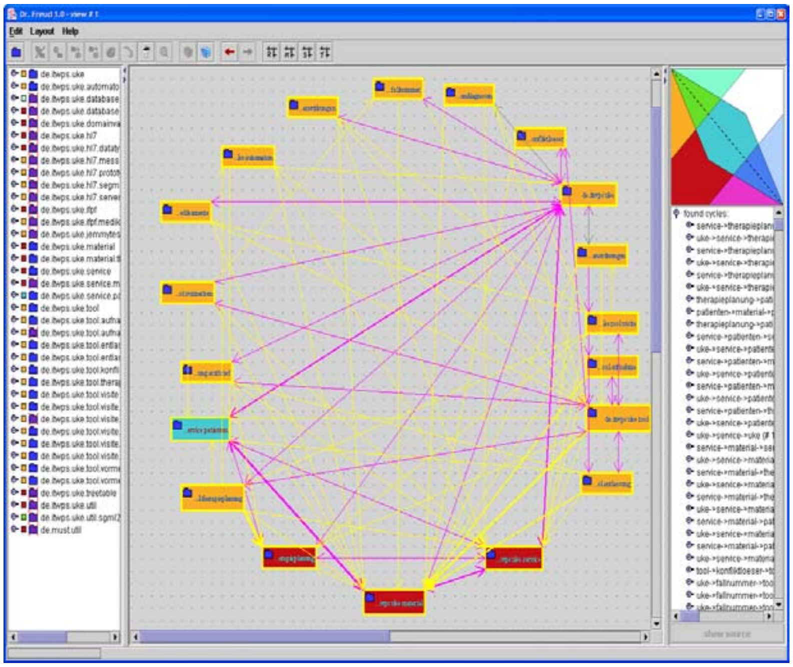


Fig. 3-36
Dr. Freud: Package Overview

3.7.7 SA4J: Structural Analysis for Java

SA4J (Structural Analysis for Java, [SA4J 04]) is a promising IBM technology preview. This tool visualizes the dependencies between packages in different ways. Particularly interesting is its highlighting of packages which are difficult to change as well as its tracking of direct and indirect dependencies, starting with a class. These functions allow developers to anticipate the consequences of changes to a class.

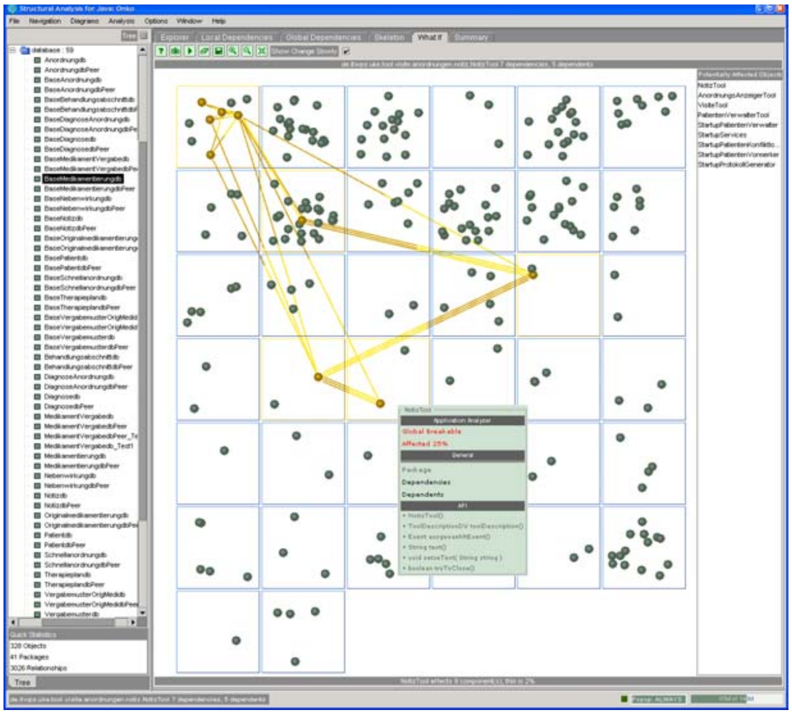


Fig. 3-37
SA4J: Dependencies

In addition, *SA4J* possesses an *Auto Explore* function that will run a movie clip showing dependencies between packages (see Figure 3-38). This feature provides an amazingly concise insight into the system's structure and quickly detects problematic dependencies.

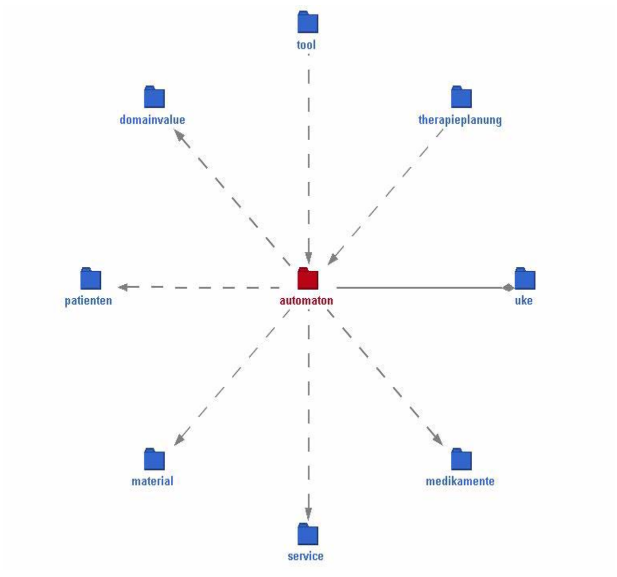


Fig. 3-38
SA4J: Dependencies of Each Single Package

3.7.8 Sotograph

The commercially available *Sotograph*[10] (see [Sotograph 03]) was developed specifically for the detection of architecture smells. It identifies the smells depicted here with little effort and is thus an important aid in controlling the architecture of a large software system. As far as we know, Sotograph is the only tool that can analyze relations between subsystems and layers (see Figure 3-39).

Moreover, it is an interesting fact that Sotograph manages all information obtained from a system analysis in a relational database, thus making it easy to create individual queries and evaluations in Sotograph. Due to its own database storage, Sotograph also allows the efficient analysis of extremely large systems (several million lines of code). Sotograph is described in detail in chapter 7.

10. The name is a compound of software and tomograph. In effect, the Sotograph is a tomograph capable of visually displaying the internal structure of a software system.

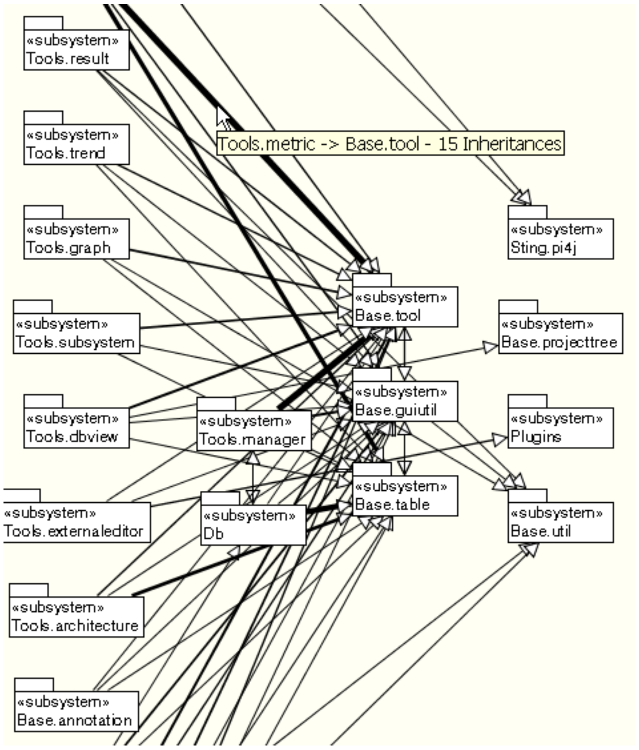


Fig. 3-39
Sotograph: A Part of a Subsystem Graph

3.8 Preventing Smells

In theory, all smells can be prevented through disciplined programming. In practice things don't quite work this way. The motto here is: The difference between theory and practice is smaller in theory than in practice.[11]

The reasons for the occurrence of smells are manifold:

- *IDEs:* Modern development environments automatically insert the imports for required classes and packages. Developers are no longer forced to encode the imports manually and must not reflect about whether the import is permitted, or if it might lead to a cyclical dependency between subsystems.
- *Pressure of time:* When the upcoming release of a system shall be delivered as soon as possible, developers are frequently pressured into violating the architecture. It is the only way of meeting the deadline. Due to time pressure, these violations of

11. We found this quote on the Internet.

architecture are often not documented, and often they will not be removed after release.

- *Misunderstandings:* Sometimes developers do not fully understand the scope and premise behind a system's architecture. They conform with what they did understand and unintentionally violate the architecture. This phenomenon occurs almost always during the training of new employees or project members.
- *Changes in architecture:* Projects that run over a longer period usually require repeated adaptations of the software architecture. These adaptations are not always done incrementally, so that existing code violates the new architecture.
- *Technological Changes:* The replacing of a technology component with a new version or with a totally different component can cause a whole series of *deprecated* warnings to occur at once.

In spite of these problems, at least cyclical references between subsystems can be constructively prevented: in the development environment, each subsystem is created as a project of its own, and the permissible dependencies between subsystems are defined as project dependencies in the development environment. In Java this will lead to the creation of one JAR archive for each project. With its dependencies between projects in Eclipse and the Eclipse plug-in model, Eclipse offers excellent support.[12]

12. For business applications, the Eclipse plug-in model can also be used without IDE. It offers a very powerful component model for Java.

Excursion: You Have to Live Architectures

A contribution by Markus Völter (voelter@acm.org)

In the course of the development of an enterprise system (J2EE server, rich client) with about 20 developers, soon a classic 'dying' of the architecture set in. With 'dying architecture' I mean that the architecture smells have such a severe impact that the quality level targeted by the architecture can no longer be reached.

Alas, everything started out so well! The concepts were clear-cut. The technical prototype was a success, the customer was thrilled and Gartner Group decided that the architecture was flawless. Then real life began to take its toll: The number of developers went up, the average qualification sank, the architect always had other things on his plate – and time pressure increased.

The consequence was that the architecture concepts were executed less and less consistently. Dependencies were in a tangle, performance dropped (too many client/server hops and too many single database queries), and originally small modifications turned into huge catastrophes. To a certain extent the architecture concepts were circumvented on purpose. For example, classes were instantiated via Reflection because the class was not accessible at compile-time.

One problem of architectures is the fact that traditional development methods do not allow an automated checking of many architectural specifications (with model-driven development and AOP (aspect-oriented programming) some betterment can be expected). The purpose of many specifications remains in the dark anyway as long as developers can't see the whole picture. Due to typical project-related constraints, developers often have no chance to familiarize themselves well enough with the architecture.

So What Is the Morale of this Story?

- Architecture concepts are very well, indeed they are very important. Just as important is the training and coaching of the developers to ensure a correct implementation of the architecture.
- Regular reviews of the code are essential to discover and eradicate unintentional or wilful violations of architectural specifications as early as possible.
- It is common knowledge that the correction of a mistake will become the more expensive, the longer you wait with it in the process. Since architectural concepts mostly define fundamental issues, it is particularly important in this context to heed this principle.

3.9 References

[Bäumer 98] Dirk Bäumer: *Software-Architekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme*. Ph.D. Thesis. University of Hamburg, Dept. of Informatics, Software Engineering Group. http://www.sub.uni-hamburg.de/disse/12/Beleg.pdf. 1998.

Bäumer describes the architecture principles of large software systems and presents a model architecture. We derived the distinction between protocol-oriented and object-oriented as well as that between strict and non-strict layers which we used in this chapter from his book.

[Bäumer et al. 97] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, Heinz Züllighoven: *Framework Development for Large Systems*. Communications of the ACM, October 1997. Vol. 40, No. 10, 1997.

The authors describe a tier architecture for large, object-oriented application systems. The tier architecture introduced here clearly ventures beyond the scope of common 3-tier models.

[Bodkin et al. 04] Ron Bodkin et al.: Enterprise Aspect-Oriented Programming with AspectJ, presentation material for the tutorial, http://www.newaspects.com/

This tutorial about Enterprise Aspect-Oriented Programming with AspectJ teaches, among other topics, how AspectJ's capabilities as *a language can be utilized to determine at compile-time whether there are method calls that illegally bypass layers.*

[ClassCycle 04] ClassCycle: *http://classycle.sourceforge.net/index.html*

ClassCycle is an open source tool for the detection of cycles between classes. It generates reports about class cycles in XML or HTML .

[Code-Smells 03] Code-Smells: http://c2.com/cgi/wiki?CodeSmell

This page of the C2-Wiki is about code smells and contains a list of often-occurring code smells. Besides code smells, one can also find references to a couple of architecture smells.

[Daum 03] Berthold Daum: *Java-Entwicklung mit Eclipse 2*. dpunkt Verlag. 2003.

This book does not only elaborate on the Eclipse development environment but also explains the plug-in architecture that is suitable as an application flow environment for business applications.

[Dr. Freud 04] Dr. Freud: *http://www.freiheit.com/technologies/download*

Dr. Freud visualizes dependencies between packages and classes and calculates Robert Martin's metrics (see [Martin 97]).

[Eclipse 03] Eclipse: http://www.eclipse.org

Eclipse is an open source development environment with refactoring support for Java. It is sponsored by IBM. Its plug-in architecture allows for an easy expansion of its functionalities, so that today a great variety of open source-plug ins as well as commercial plug-ins for Eclipse exists.

[Eclipse Metrics Plug-in 04] Eclipse Metrics Plug-in: http://sourceforge.net/projects/metrics

This is an open source plug-in for Eclipse which provides common metrics for object-oriented systems, e.g. the average method length. The resulting values let developers – where this makes sense – directly navigate towards the source of a smell, e.g. exceptionally long methods.

[Fowler 99] Martin Fowler: *Refactoring. Improving the Design of Existing Code*. Addison-Wesley. 1999.

The standard work about refactorings. Besides refactorings, this book contains a list of code smells – that is, the little sisters and brothers of the architecture smells discussed here. The comprehensive code examples refer to Java, but they can relatively easily be applied to other object-oriented programming languages.

[Gamma et al. 97] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1997.

This standard work on design patterns also contains patterns which lead to some of the architecture smells addressed in this chapter. This is, for instance, the case for the iterator pattern, which results in cyclical relations between the container and the iterator. The cycle could be eliminated, but this will lead to some loss of clarity. This cycle is acceptable though, since only two classes are involved in it. These are closely coupled anyway and will be put in the same package.

[Hunt & Thomas 98] Andrew Hunt, David Thomas: *Tell, Don't Ask*. http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html

A depiction of the Tell, don't Ask *principle.*

[JDepend 04] JDepend: *http://www.clarkware.com/software/JDepend.html*

JDepend analyzes the dependencies between packages and classes and calculates Robert Martin's metrics (see [Martin 97]).

[JDepend4Eclipse 04] JDepend4Eclipse: *http://andrei.gmxhome.de/jdepend4eclipse*

JDepend as an Eclipse plug-in.

[Kerner 89] H. Kerner (Hrsg.): *Rechnernetze nach ISO-OSI, CCITT.* 1989.

Describes the ISO-OSI layer model.

[Lakos 96] John Lakos: *Large-Scale C++ Software Design*. Addison-Wesley. 1996.

This book introduces important architecture principles of comprehensive software systems that are relevant beyond the C++ context.

[LawOfDemeter 03] Law of Demeter: http://c2.com/cgi/wiki?LawOfDemeter

This page of the C2-Wiki gives a description of the Law of Demeter, according to which an object is supposed to communicate only with its direct "friends." Technically this means that an object shall not invoke a method on an object that it received from another object: method calls on objects which have been the results of functions are not permitted.

[Lieberherr & Holland 89] Karl Lieberherr, Ian Holland: *Assuring Good Style for Object-Oriented Programs*. IEEE Software. September 1989. Pp. 38-48.

The article depicts the Law of Demeter.

[Mackinnon et al. 00] Tim Mackinnon, Steve Freeman, Philip Craig: *Endo-Testing: Unit Testing with Mock Objects*. XP 2000 Conference. 2000. The online version of this resource can be found at: http://www.connextra.com/aboutUs/mockobjects.pdf

The original article about Mock Objects was introduced at the XP 2000 conference.

[Marquardt 01] Klaus Marquardt: *Dependency Structures. Architectural Diagnoses and Therapies*. In: Proceedings of the Sixth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 2001). UVK 2001

In this article, a number of bad smells are discussed in the form of diagnoses and therapies. The collection of diagnoses primarily focuses on architectural aspects and offers a variety of possible therapies for each smell that will help remove it.

[Martin 97] Robert C. Martin: *Stability.* C++ Report, 1997

Although this article is several years old, the content has neither collected dust, nor is it C++-specific. Martin explains important architecture principles that can also be found in this chapter's architecture smells.

[Mock 03] Mock Object: http://c2.com/cgi/wiki?MockObject

This page in the C2-Wiki depicts the Mock Object test pattern which allows isolated testing of interdependent parts of the system. The use of Mock Objects furthers the Law of Demeter *and the* Tell, don't Ask *principle.*

[PMD 03] PMD: http://pmd.sourceforge.net

PMD is an open source tool for the detection of code smells in Java systems, such as empty catch blocks. It can also be used to check *portions of program conventions. As a plug-in, PMD can be integrated in various development environments.*

[Produktlinien 03] Produktlinien: http://www.sei.cmu.edu/plp/product_line_overview.html

This source explains the concept of product lines. According to the definition given here, a product line is a set of software systems that share a common basis.

[Refactoring-Web 03] Refactoring-Web-Site: http://www.refactoring.com.

On the refactoring website operated by Fowler, among other material, an online catalogue of refactorings, which has long exceeded the scope of Fowler's refactoring book, can be found. One can also find links to other websites dealing with refactoring.

[RefactorIT 03] RefactorIT: http://www.refactorit.com

RefactorIT is a commercial refactoring tool, which – as a plug-in – can be integrated in various development environments. Not only can RefactorIT execute refactorings, but it is also able to create a number of metrics for Java systems.

[SA4J 04] SA4J: http://www.alphaworks.ibm.com/tech/sa4j

SA4J (Structural Analysis for Java) is a promising IBM Technologievorschau. This tool visualizes the dependencies between packages in various way.

[Sotograph 03] Sotograph: http://www.sotograph.com

The sotograph supports the quality assurance of large systems on the software architecture level. Besides the system to be checked, the Sotograph also reads a description of its architecture, against which it checks the system. Thus architecture smells are easily identified.

[Szyperski 97] C. Szyperski: *Component Software*. Harlow, England, Addison-Wesley. 1997.

The subsystems mentioned in this chapter can also be referred to as components.

[TellDontAsk 03] Tell, Don't Ask: http://c2.com/cgi/wiki?TellDontAsk

This page of the C2-Wiki gives an explanation of the Tell, don't ask *principle, which can also be understood as an clarification of the* Law of Demeter*: objects shall not be asked for information, which*

will make the client act on it. Instead, the client shall tell the object what it is supposed to do. Thus is ensured that knowledge about dependency graphs will not spread over the whole system.

4 Large Refactorings

In this chapter, we are going to address refactorings that are not covered in Fowler's work, i.e. those other than basic refactorings.

To this end, we will introduce the term *large refactorings* to clearly distinguish them from Fowler's basic refactorings.

Two exemplary collections of samples form the core of this chapter. They reflect our experiences with and best practices for large refactorings. We differentiate between two types of samples: on one hand, we address organizational problems and solutions as well as those that are part of the development process. This approach is gaining more and more relevance, especially for large refactorings. The solutions we offer can be applied to your own projects and help you find adequate ways of dealing with large refactorings. On the other hand, we analyze recurring fragments that you can use as modules for your own large refactorings.

4.1 Introduction

During our participation in numerous projects we recurrently observed that – besides fundamental refactorings – larger restructurings are required. If, for example, a pivotal inheritance hierarchy in the system must be rearranged, the impact of such a change can significantly affect the system. It might become necessary to adapt considerable portions of the code. We will call such restructurings *large refactorings*.

Reasons for Large Refactorings

Large refactorings may be needed for various reasons. These are the most common ones:

- Developers put off small refactorings too long. If a software's design isn't continuously improved, small design weaknesses will accumulate, and a more comprehensive rearrangement might eventually be required.

- Architecture smells emerge – unnoticed first – over time. If one tries to cure them, the respective refactoring can very soon expand beyond the scope of a small and basic refactoring.
- New features or altered software requirements can necessitate large refactorings. While some features will either integrate seamlessly into a software or after a couple of minor refactorings, others call for a more elaborate restructuring.

Therefore, many development projects avoid executing large refactorings while a project is underway. As a result, the outdated structures will often be left in the system, or they will be tackled with a large redesign after release. We pursue the goal of integrating large refactorings into an ordinary, evolutionary development process.

While many developers possess an intuitive understanding of what constitutes a large refactoring, it is difficult to come up with a *precise definition*. Intuitively, the following characteristics are assigned to large refactorings:

1. *Duration*: Large refactorings last longer than one day.
2. *Team*: Large refactorings affect the entire project team.
3. *Unsafety*: Large refactorings cannot be completely replaced by basic (safe) refactorings. Additional (unsafe) modifications are required.

Unfortunately, these characteristics prevent a totally clear distinction between large and basic refactorings. The manual renaming of a central method in a big system will take more than a day and concern the entire team, but it can be fully realized through applying the basic refactoring *Rename Method*. If the development environment supports the renaming of methods, the refactoring will be done in a few minutes, so that at least the first characteristic of large refactorings listed here no longer applies.

In this book, we will content ourselves with this loose definition because we believe it does not impair the comprehensibility and usefulness of this chapter. The intuitive understanding based on the three characteristics mentioned above offers a sufficiently clear framework.

Even while dealing with basic refactorings we learned that these are no trivial matter. We observed the same for large refactorings. Often coming up with small steps that are self-contained (i.e. compilable and testable) appears to be particularly complicated.

One of the reasons, among others, is that a large refactoring will affect significantly more code in the system than a small one. Not all effects that a refactoring has on the system will immediately be evident. The previously mentioned change impact analysis might be of some help here. Moreover, the sections on mechanics in Martin Fowler's book will provide valuable advice on how refactorings should be broken down.

4.1.1 Important Terms

A number of terms will repeatedly come up in the following sections. We wish to explain these briefly. First, we differentiate between basic and non-basic refactorings:

Basic Refactorings

- *Basic refactorings* are those refactorings that are described in [Fowler 99] and mostly refer to basic object-oriented constructs.

Non-basic Refactorings

- A *non-basic refactoring* is a refactoring that exceeds the scope of a basic refactoring as addressed by Fowler. This category includes the large refactorings discussed in this chapter as well as those restructurings which [Fowler 99] calls *big refactorings*.

Besides distinguishing between basic and non-basic refactorings, the safe execution of a refactoring is also very important to us. In this context, 'safe' means that the developers can be certain not to introduce any new errors in the course of their respective refactoring.

Safe Refactorings

- *Safe refactorings* are refactorings that can be executed without risking changes to the system's behavior or creating new errors. If, for example, a tried step-by-step instruction for a refactoring is available (such as the *Mechanics* in [Fowler 99]), the refactoring can be carried out with no risk of creating new errors.

Unsafe Refactorings

- *Unsafe refactorings* are refactorings for which no tried step-by-step instructions are available that would allow their safe, incremental execution. One example of an *unsafe refactoring* is the renaming of a class.

Modern, integrated development environments allow a completely automated execution of certain refactorings. Such tools can turn unsafe into safe refactorings. This is, for example, the case for renaming a class. Whereas no mechanics exist for this refactoring that would allow a safe manual execution, it can be carried out automatically via an IDE, which guarantees that the system's behavior will remain unchanged. In consequence, in this case the renaming of a class with an IDE belongs to the category of safe refactorings.

A modern IDE's refactoring support has a quite significant impact on many refactoring activities. Not only it is remarkable that unsafe refactorings can quickly be made safe with the help of an IDE. Moreover, some refactorings can be carried out in a short time although they change many lines of code in the system. For instance, if we rename a method, this step can potentially affect many places in the system (e.g. those calling that method). Even though this refactoring is also considered safe if it's done manually, the IDE's refactoring support changes the work with such a refactoring. We thus make a distinction between automated and manual refactorings:

Automated Refactorings

- *Automated refactorings* are refactorings that are supported by an IDE and therefore can be executed automatically. In this process, the IDE ensures that the system's behavior will not be changed. As a rule, automated refactorings are always safe refactorings. In addition, automated refactorings can be carried out – regardless of the system's size
– in a very short time.

Manual Refactorings

- *Manual refactorings* are not supported by the IDE and must be conducted manually by the developers.

4.1.2 Beyond Automated Refactorings

Until now, theoretical works dealing with refactoring issues foremost discussed the functional realization of refactorings. They focused on the automation of basic or even quite complex refactorings or provided mechanics for refactorings. In contrast, the development process aspect has rarely received any attention. While it is often stated that refactorings fit in well with agile development processes, the effects of refactorings on the development process are hardly ever considered. This may not be necessary for many basic refactorings, because they can easily be supervised and handled by a single developer. Here, neither a specific development process is required, nor must particular organizational conditions be provided.

Large Refactorings Behave Differently

Large refactorings behave differently. As Fowler and Beck in [Fowler 99] already remarked for big refactorings, large refactorings can affect the whole team and create certain requirements that must be met by a suitable development process, that is, for large refactorings we must explicitly address problems of how to plan, communicate and execute large refactorings in a team.

4.1.3 Can Large Refactorings Be Avoided?

Large refactorings create additional development problems and accordingly require additional efforts. Here, we notice that the added problems can be solved, but they do create an extra demand for planning, communication and discussion, which leads us to the legitimate question if there is a way of avoiding large refactorings altogether.

First, one could state that continuous refactoring during development renders large refactorings obsolete. This is the basic idea behind refactoring technology as it is applied, for example, in Extreme Programming. Occasionally larger refactorings are needed in XP projects too though.

This is due to the underlying assumption that software development is a learning process. It also means that developers must revise design choices that were made earlier on if new software requirements demand a new software-ergonomic design. Depending on how well the new requirements fit the software's existing structure, these refactoring measures will be smaller or bigger.

Project Example

For instance, in one project it took us quite a while to find out that we had, until then, misunderstood a part of the field of application. Since our wrong model of that field of application naturally had become a part of the software, the software design needed to be adapted to suit our improved understanding of this field of application.

Spike Solutions

Surely such effects can be attenuated through the implementation of either spike solutions or prototypes for the basic system's architecture at the beginning of a project. However, of course there is no guarantee that the assumptions leading to the implementation of the new design, made at the beginning of the project, will prove to be right.

Thus we arrived at the conclusion that large refactorings cannot always be avoided. Regular refactoring during ongoing development helps to keep the design flexible and up-to-date. Design problems will be noticed early on and therefore can be solved quickly. This protects developers from postponing refactorings and thus letting the design degenerate (which, in consequence, would require large refactorings). Yet misunderstandings regarding the field of application cannot be avoided entirely.

Violations of Architecture

Furthermore, in connection with our use of the Sotograph, we observed that violations of architecture can easily happen because developers cannot always recognize them right away. If, for example, developers integrate cycles on the subsystem level, there will be no indication that something is wrong. The cycle remains unnoticed in the system. Only a systematic analysis will reveal the potential problem. But even with the aid of the Sotograph, architecture smells cannot be prevented. The Sotograph will help us realize the actual problem only after we have already detected the smell. Nevertheless, a large refactoring will often be required to eliminate it.

Excursion: Refactoring – Not as Hard as Expected

A contribution by Berrin Ileri (berrin.ileri@it-fws.com, it-FWS GmbH) and Henning Wolf (henning.wolf@it-wps.de, it-WPS GmbH)

Motivation and Background

Together with ten colleagues we are involved in a project for a major municipal utility. It is our task to develop an individual solution in Java that mainly serves to support prearrangements for work processes. Since altogether four different organizational units (OU) of our employer are involved in this project, parts of our solution turn out to be specific to certain fields of application, in addition to those parts that serve all units.

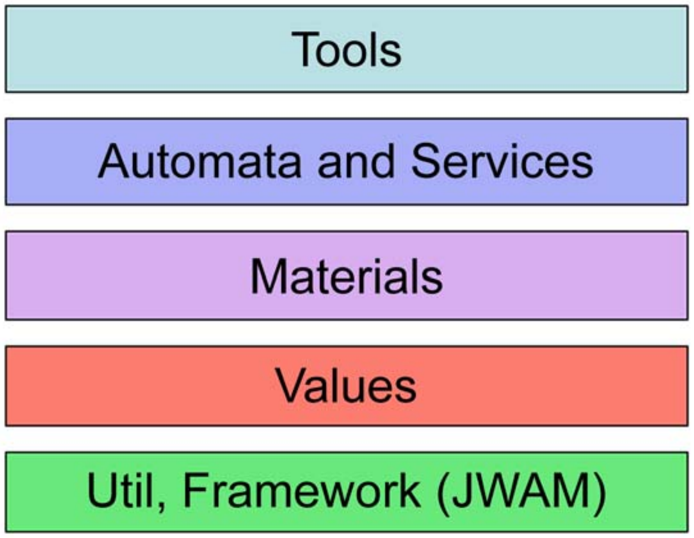
Our development background is heavily influenced by the German metaphors '*W*erkzeug' (tools), '*A*utomat' (automaton), services and '*M*aterial' (materials) that constitute the *WAM* concept. Of course, we also apply the JWAM framework (http://www.jwam.de), which already offers a series of abstractions for these design metaphors.

The parts of the system we developed until today comprise almost 550,000 lines of code with about 3,500 classes (of which 1,000 classes are anonymous inner classes). Nearly 1,000 man-days were needed to reach the current state. The scale of the scheduled system upgrading is assessed to require another 3,000 man-days.

Our First Target Architecture

In our project work, we adhere to a layered architecture (see figure "The Original Layering") that was familiar to most developers from other projects. Each class of a layer is allowed to access any other class of that layer as well as all classes of the layers beneath, i.e. the layering is not strict. The corresponding package structure looks as follows (in this and in the following examples we always show two organizational units; the other two behave accordingly):

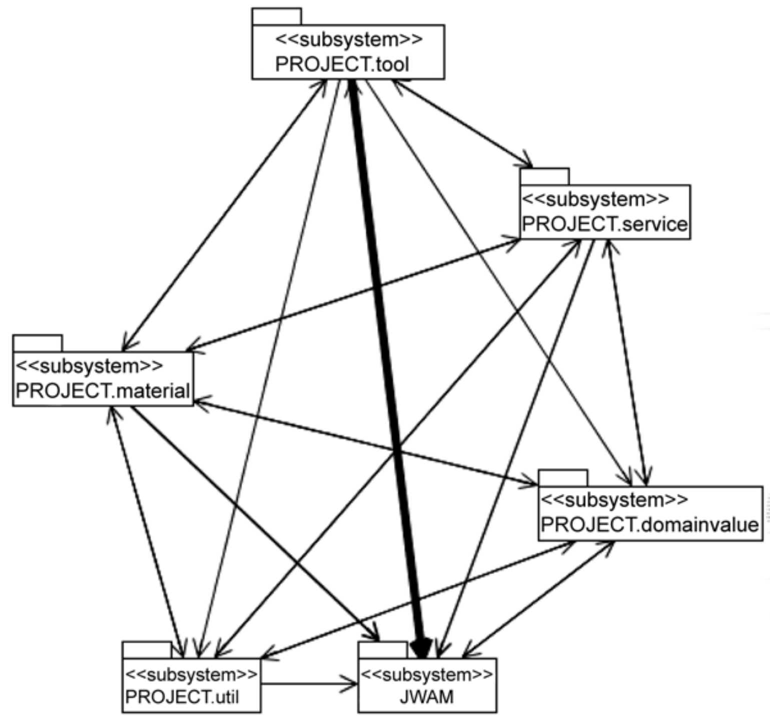
- de.customer.project
  - tools
    - ou1
    - ou2
    - general
  - services
  - materials
  - values
  - util



The Original Layering

The First Disillusionment

A short while ago we had our first opportunity to have our software architecture tested with Sotograph. Of course we had hoped that the result would confirm our skills as software engineers. You may take a look at the general survey graph below. The lines represent all kinds of relations (inheritance, usage) between architectural units. The line width as well as the width of the arrows convey the relative number of relations.



The First Survey Graph

The high number of double arrows (regrettably) shows that the targeted architecture was violated in many places. In defense of our approach we'd like to point out that the majority of violations was caused by (JUnit) test classes that we always put in the package next to the class to be tested.

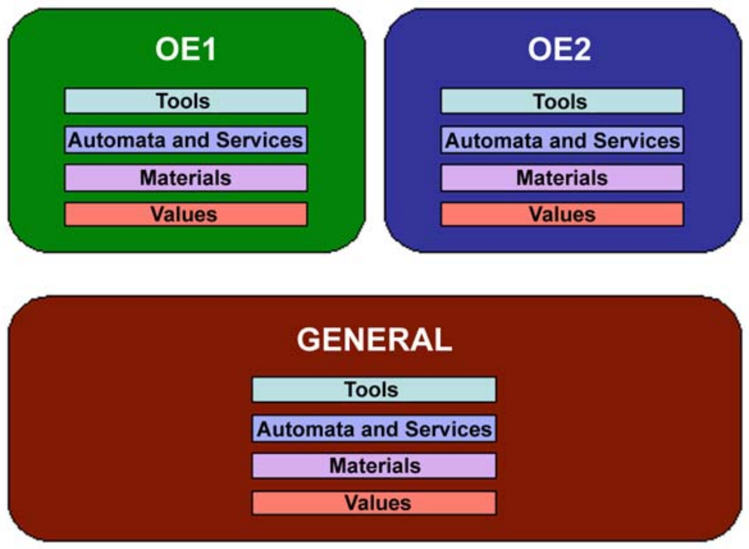
With the aid of Sotograph we analyzed those violations in detail and generated a to-do list containing a significant number of classes to be moved to another package and a large refactoring for our central tool. This tool had until then been insufficiently accessible to the organizational units, forcing them to take turns in using it.

Our Second Target Architecture

Since the project shall become much more comprehensive in the future, we at this point decided to alter organizational units to obtain a clear-cut structure. The package structure looks as follows now:

- de.customer.project
  - ou1
    - tools
    - services
    - materials
    - values
  - ou2
    - tools
    - services
    - materials
    - values
  - general
    - tools
    - services
    - materials
    - values
  - util

The logical structure is shown in the following diagrams:

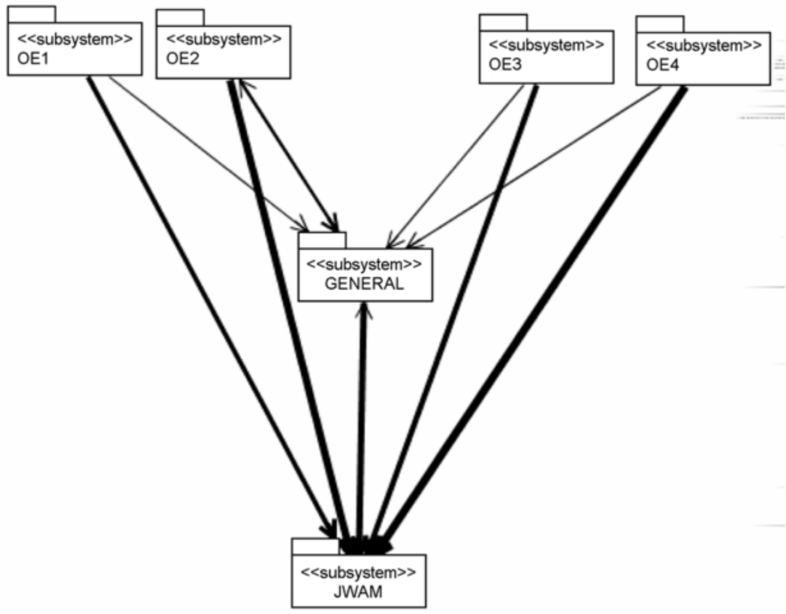


The Targeted Logical Structure

The First Large Refactoring

The already mentioned to-do list formed the basis of our large refactoring. It mainly consisted of simple relocations of classes into other packages. The big challenge here was the modification of a rather complex tool that needed to be broken down into one general part and specific parts for both organizational units to be supported. Contrary to our misgivings, this restructuring work was dealt with rather smoothly, requiring little more than 100 hours (of which 90 were dedicated to the tool's modification). However, we were aware that this would not solve all our problems, although it erased a remarkable amount of 'white noise' during architecture analyses.

The result of our refactoring can be seen in the following figure if you take a look at the top level. The organizational units are independent of each other, but there are still two double arrows left, which means two subsystems still depend on each other. To put this result into perspective, it should not go unmentioned that only a single reference exists between the subsystems *GENERAL* and *OU2*, whereas subsystem *JWAM* displays six references to subsystem *GENERAL*. Especially the latter references may be confusing, since there can hardly be any calls from general framework classes that address specific project code. The reason is that certain framework classes are overshadowed and project-specific classes are referenced within them. This can easily happen when today's development environments are used, due to the automated generation of import statements. On the other hand, we could break down our project into various source code projects for development purposes to constructively prevent such impermissible calls on the compiler's side.

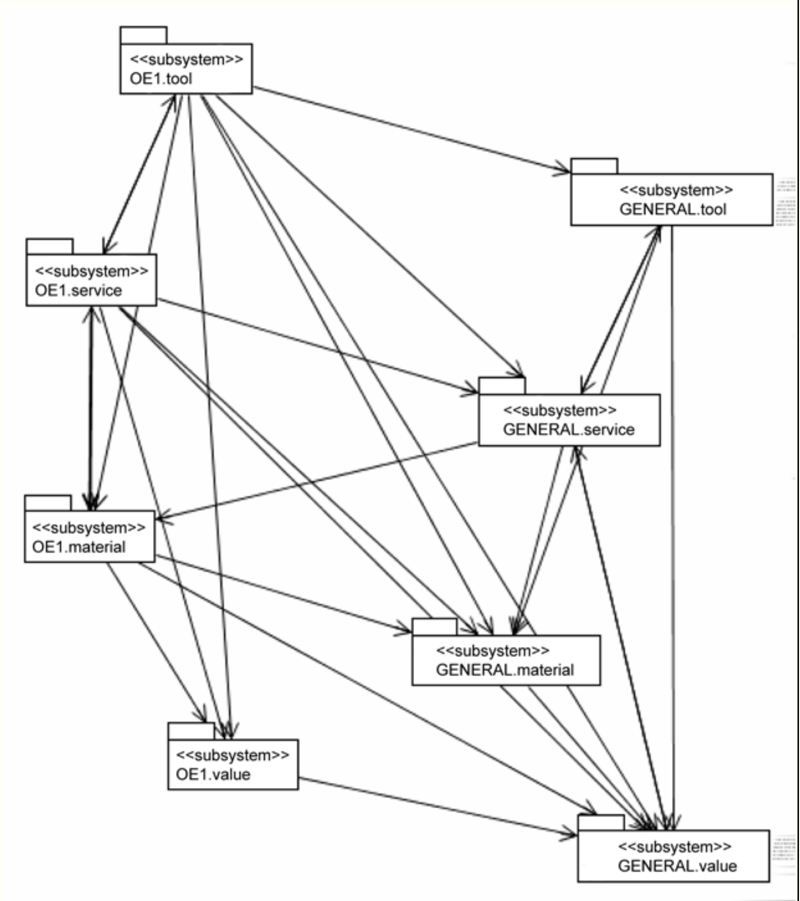


The Refactoring's Result on the Top Level

Beneath the organizational unit level we still have got the tool, services and materials levels that we need to consider in context with the subsystem *GENERAL*. When we did this for the organizational unit (OU) 1, the outcome was what you see in the following figure. The architecture we targeted allows the OU tools layer to access all other OU layers plus all *GENERAL* layers, while the OU services layer should neither access the OU tools layer nor the *GENERAL* tools layer. This rule also applies to the materials and values layers. This aspect was observed, although we were confronted with the following architecture violations:

- 5 references from OU services to OU tools
- 2 references from OU materials to OU services
- 71 references from *GENERAL* services to *GENERAL* tools
- 10 references from *GENERAL* values to *GENERAL* services

In addition, we discovered a reference between *GENERAL* services and *OU materials* that points from *GENERAL* services toward *OU materials*. References in this direction were not planned and thus constitute another violation of architecture.



<<subsystem>> OE1.tool
<<subsystem>> GENERAL.tool
<<subsystem>> OE1.service
<<subsystem>> GENERAL.service
<<subsystem>> OE1.material
<<subsystem>> GENERAL.material
<<subsystem>> OE1.value
<<subsystem>> GENERAL.value

Dependencies on the Next, More Detailed Level

If we add the details of architecture violations we detected when we looked at other organizational units to the ones we already discussed here, we'll get a second to-do list which will become the venturing point of our next refactoring.

Conclusion

Architectures provide an overview of complex software systems. However, architectures are always tailored to meet the status quo. They cannot apprehend changes that occur in the course of a project. Without continuous checks if the targeted architecture's requirements are met, the architecture will merely remain a UML diagram or an outline on paper. It will not noticeably contribute to structuring the source code. Contrary to our negative expectations, most architecture violations could relatively easily be cured with little effort. The less sophisticated refactorings were those most needed though, as they concerned parts of the system that urgently required adaptations of details.

4.2 Best Practices for Large Refactorings

We will dedicate this section to the difficulties that are either of an organizational nature or stem from the development process itself. Such difficulties occur quite often during large refactorings. Typical problems of large refactorings are:

1. *The effects of large refactorings*: A large refactoring can affect big parts of a system.
2. *Breaking down large refactorings*: Large refactorings must be broken down into smaller increments.
3. *The use of basic refactorings*: Large refactorings can only partly be constructed from basic refactorings. They are more than just a series of basic refactorings.
4. *The process of breaking them down*: The breaking down of large refactorings into single steps is a quite demanding task.
5. *Detours in the Code*: The introduction of temporary detours in the code is often necessary. The system structure must deteriorate first before it can be improved.
6. *Assessment of consequences*: It is difficult to predetermine the consequences of single steps in large refactorings.
7. *Unfavorable or wrong refactoring routes*: It will frequently happen during a large refactoring that developers realize that they've chosen an unfavorable or even a completely wrong refactoring route.
8. *Interruptions*: Large refactorings must repeatedly be interrupted to meet new software requirements.

9. *Loss of Orientation*: It is difficult to stay up to date as far as the actual state and the goal of the large refactoring are concerned.
10. *Large refactorings are teamwork*: They should not be carried out by single developers without continuously consulting the project team.

To deal with these problems, a number of techniques that are widely applicable have been established in the course of various projects. They range from planning and project organization questions to concrete technical implementations. Because of the vast scope of this field, these techniques are not as elaborate as Fowler's basic refactorings. Even when aided by the techniques described here, large refactorings will still require a lot of thinking and creativity.

4.2.1 Practice: Scheduling Large Refactorings

Problem

Large refactorings might be needed in the course of an evolutionary development process, and they can be executed in various ways. Apart from that fact, in our projects it soon became clear that it is easy to lose track of large refactorings, and that they will not executed completely if at the same time the project happens to be subject to further development. One of the reasons thereof is probably that new features appear to be more important than a refactoring. Time pressure can contribute to further delay when certain functions need to be urgently realized for a specific iteration.

However, it cannot be our goal to put off large refactorings indefinitely. After all, we already learned that refactorings will become the more difficult to execute the longer they are postponed. This is why we have to make sure that even large refactorings will not perish in everyday development work.

Solution

We can solve this problem by implicitly integrating refactoring work in the planning process. This means that we will include large refactorings in the iteration and release schedules.

In practice, we observed three different options:

Refactoring Budget per Iteration

- *Option 1:* For each iteration, we schedule roughly the same amount of time for each refactoring, thus allowing enough time for the team to carry out refactoring work and to advance large refactorings. We are quasi concealing refactorings behind technical requirements.

- The advantages: From the customer's point of view, the project progresses continuously. The customer will not get the impression that the developers interrupt the project in order to 'clean up' and to eliminate earlier mistakes.
- The disadvantages: Refactorings are sacrificed due to technical requirements. For this variety it is very likely that large refactorings will either be forgotten or not even begun.

Refactoring Iterations on Demand

- *Option 2:* Specific refactoring iterations are introduced on demand. These iterations only serve the execution of refactorings. In the meantime, system development is on hold.

  - The advantages: This option constitutes a quite simple approach, since the focus is exclusively on the required refactoring work. Thus interferences between work on large refactorings and the realization of functionalities are ruled out.

  - The disadvantages: The customer will not be able to observe any technical progress. From the customer's point of view, it looks like the project is dormant while he/she is paying. In consequence, it is possible that refactoring iterations cannot be planned due to time pressure.

Regular Refactoring Iterations

- *Option 3:* Frequent refactoring iterations. In one project, for example, we conducted three technical iterations and one refactoring iteration of a week each and created a release from them.

  - The advantages: This procedure is simple and allows the team to solely focus on refactoring work for a defined period. One achieves an alternation between tense periods (technical iterations) and relaxed periods (refactoring iteration).
  - The disadvantages: Clearly defined refactoring iterations might turn out to be too formal and too strict for the team. If, e.g., the code is very clean and no large refactorings seem necessary, a rigidly scheduled refactoring iteration does not make sense. In addition, this sort of planning invites the neglection of small refactorings during routine development work. The developers are possibly tempted to put off refactorings.

One cannot generally say that one of these options is better than another one. In practice, the decision which one is chosen must be made based on the respective project situation.

4.2.2 Practice: Refactoring Planning Session

Problem

Large refactorings can be noticeably more difficult and complicated than their smaller relatives. They clearly require more time and seriously influence a team's work. Therefore, large refactorings gain more and more importance in the development process. It is no longer safe to assume that they – like small refactorings – can be easily dealt with as part of a developer's everyday routine, and that they need no attention during their implementation.

To be able to efficiently integrate large refactorings in an agile development process, we have to bear in mind the larger picture and think beyond the refactoring itself. After all, a large refactoring can affect the work of an entire development team.

The execution of single, partial steps of a large refactoring which are integrated in the common code repository of the development team, creates uncertainties for developers who are not immediately participating in the refactoring work. Once the team has carried out one half of the refactoring, the code contains portions of the new as well as of the old structure. In addition, detours are integrated in the code to allow for these intermediate steps. For the developers it becomes increasingly difficult to keep track of the entire refactoring. The question, asked by developers, why a specific method is suddenly *deprecated*, is convincing evidence.

Solution

A simple and at the same time basic means is to discuss and plan large refactorings with the entire team. Similar to a *quick design session* in *Extreme Programming*, all developers shall participate in a brief refactoring session, during which the design problem can be discussed and a possible refactoring route outlined. In addition, the developers can discuss a rough time schedule for a large refactoring to permit a rather uncomplicated proceeding.

The refactoring session also fosters direct communication in the team. After such a refactoring session, the design problem has been made known to all team members, and they all have been informed that the respective part of the system will undergo change. Also, all team members are familiar with the goal of the large refactoring and thus able to integrate it in their daily work.

For us it has become a significant part of the development process to discuss major design modifications with the team and schedule them as part of the process.

4.2.3 Practice: Refactoring Plan

Problem

Typically, a large refactoring will take place over a longer period. In the course of their development work, the developers will frequently interrupt the refactoring to further develop other parts of the system or generally add new features.

Once the important core of a large refactoring has been implemented, in some cases the refactoring is not completed, i.e. there is no such thing as a 'clean finish.' For large refactorings this means that, for instance, implemented detours will remain in the code, or only parts of the system will be adapted to the new structure. The large refactoring is left incomplete, with the consequence that the system structure is suspended in an intermediate state. This preliminary structure contains parts of the new design as well as parts of the old one, including detours. It becomes much more difficult to understand and change the system.

If the developers totally forget about the refactoring and do not finish it, the system will still be runnable, but it will possess a structure that is inferior to its structure before the refactoring. The superclass *List* still has the method *insertAt* and thus also its subclass *SortedList*. Eliminating this constellation was the original goal of the refactoring. For the team's developers, this new structure might be much harder to understand than the old one if they are not aware of the respective refactoring.

Solution

Besides conducting a common refactoring session, it has proven useful for us to write down an outline of the previously discussed refactoring route and put it up somewhere where everyone can read it. For our work, such a refactoring plan typically contains the single steps of the large refactoring. Developers shall place such a schedule in a prominent location. Thus it will be visible and present for all developers.

The refactoring plan initially discussed in the team is by no means written in stone for the whole refactoring period. First and foremost it serves as a representation of the large refactoring, that is, to bring the large refactoring to the developers' attention every now and then. It can also serve as a guideline for working on the large refactoring. As a consequence, the single steps of such a refactoring schedule can be altered, or their order can be rearranged. A refactoring plan is no work regulation, but an aid for keeping track of the refactoring process.

It is important to point out that the single steps of a refactoring plan do not exclusively depict modifications of a system (for example changing class names), but also clarify the intention of that particular step (class A inherits from class B). A mere listing of modifications makes a refactoring schedule vulnerable for modifications of the system that take place simultaneously.

It has proven especially helpful to check off which steps were already successfully executed on the refactoring plan. In this way, all developers of the team can see how far the refactoring has progressed, and what steps will probably be tackled next.

However, a refactoring schedule does not substitute direct communication between the developers of a team. Instead it promotes awareness of a large refactoring and its discussion. It helps to keep it in mind and realize its progress.

Excursion: Electronic Refactoring Schedules

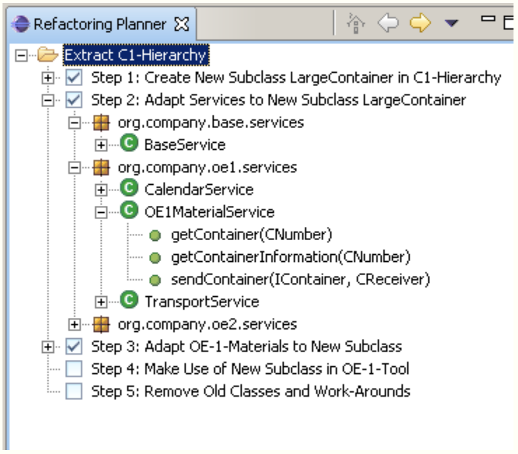
Martin Lippert's Vision

The manual schedules we introduced in the previous section already provide some support to the developers of a team for the execution of a large refactoring. Realizing the effect of a refactoring on the system's concrete source code, however, remains difficult for developers. They can see that a large refactoring is being carried out that is not yet finished, and they can recognize what intermediate state it is in. It remains difficult though to reference, for instance, completed steps of a refactoring plan to changes in the source code. The question why a certain method is currently *deprecated* is not answered in this context.

To offer the developers even more comprehensive support, we wish to create a connection between the refactoring schedule and the system's source code. To this end, it evidently makes sense to convert the refactoring schedule into a digital version and make it an integral part of the project's source code.

First of all this means that the refactoring plan must be digitalized and integrated in the shared code repository. Then it can be directly accessed by each developer from his or her workplace[a]. A simple text document would serve the purpose. A comparable result could be achieved if the developers decide to manage their refactoring plans in a project Wiki web.[b]

A much better accessibility can be achieved when the digital refactoring schedule is directly integrated and made visible in the used IDE. In the Eclipse IDE, this could be realized via a special view.



```
Refactoring Planner
Extract C1-Hierarchy
    ☑ Step 1: Create New Subclass LargeContainer in C1-Hierarchy
    ☑ Step 2: Adapt Services to New Subclass LargeContainer
        org.company.base.services
            BaseService
        org.company.oe1.services
            CalendarService
            OE1MaterialService
                getContainer(CNumber)
                getContainerInformation(CNumber)
                sendContainer(IContainer, CReceiver)
            TransportService
        org.company.oe2.services
    ☑ Step 3: Adapt OE-1-Materials to New Subclass
    ☐ Step 4: Make Use of New Subclass in OE-1-Tool
    ☐ Step 5: Remove Old Classes and Work-Arounds
```

Refactoring Planner View in Eclipse (Mock-up)

The refactoring planner view in the figure above shows the five steps of the 'Extract C1-Hierarchy' refactoring. The single steps are each placed next to a check box which announces whether the respective refactoring step has yet been executed or not. Below each refactoring step the changes brought about by the refactoring are displayed (grouped by packages). The changes are made visible down to the level of single methods.

Optional navigation is possible from classes or methods to either the corresponding editor or to a diff-viewer showing detailed changes for each single refactoring step.

IDE integration enables easy changes of the refactoring plan and committing it back to the common repository. Additionally, IDE Integration should allow checking off single steps in the refactoring plan.

With a digitalized refactoring schedule as opposed to a handwritten plan on the development lab's wall, not much has been won yet. Of course it is easier for developers to modify a digital schedule, but this plan does not possess the same charm as a handwritten plan on the wall.

A noticeable and important advantage will be won the adoption of a digital refactoring plan if it is linked to the system's source code. In this case, the developers can detect correlations between single steps of the refactoring schedule and modifications of the source codes. Consequently, a step in a refactoring plan could be assigned certain code-modifications. A refactoring step would then hold the information which places in the code were altered.

Once that information is available, a two-way navigation could be realized: On one hand, developers would get an overview of the altered code sections based on the plan. On the other hand, they could also navigate from the code to the refactoring plan if the former was modified in the course of a large refactoring. If, for example a method has been marked *deprecated*, the developer can find out to which refactoring step this change can be attributed.[c]

Refactoring Maps

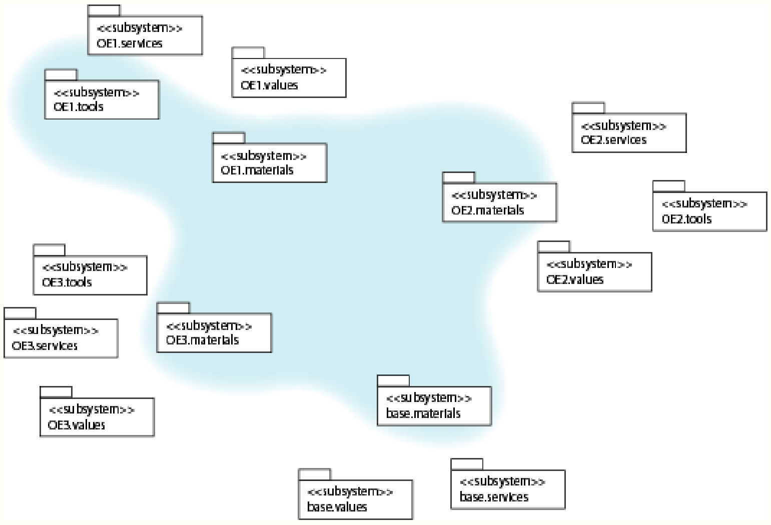
Electronic refactoring plans possess a number of advantages (see previous sections). While the refactoring plan serves to visualize the execution of a large refactoring and makes the single refactoring steps transparent for each team member, it does not necessarily help the developers in the team to assess the refactoring's impact on their daily work outside the refactoring context. Often I – the developer – wish to know how a refactoring will affect my daily routine. Do I have to look at the refactoring plan at all or can I do my job without keeping the refactoring in mind?

I want to be able to see at a glance if the refactoring concerns me, if the part of the system on which I plan to work is already being refactored, or if that part of the system is approaching refactoring. If the latter is the case, the refactoring plan will help me get an idea of the refactoring itself and let me recognize what I have to observe in my work. Should the refactoring take place far away from my own 'construction site,' I can probably ignore the refactoring plan.

But how can I see at once whether the refactoring is 'closing in' or already affecting the part of the system I am working on?

Our idea is to use a so-called *refactoring map* to present the required information in a concise format. A refactoring map represents the system in two dimensions. On this level, the different parts of the system are arranged based on a particular pattern. For very small systems a class diagram will suffice; more comprehensive systems require a package or subsystem diagram[d]. The developer must be able to identify his or her own 'construction site' right away on this map. This can be achieved, for example, if the developer is able to isolate the source code on the refactoring map.

Moreover, a refactoring in progress is visualized on the map through the use of colors. The affected parts of the systems are tinted the color assigned to the refactoring. Thus, each developer can easily see what parts of the systems have already been altered.


<<subsystem>> OE1.services
<<subsystem>> OE1.values
<<subsystem>> OE1.tools
<<subsystem>> OE1.materials
<<subsystem>> OE2.services
<<subsystem>> OE2.materials
<<subsystem>> OE2.tools
<<subsystem>> OE2.values
<<subsystem>> OE3.tools
<<subsystem>> OE3.services
<<subsystem>> OE3.materials
<<subsystem>> OE3.values
<<subsystem>> base.materials
<<subsystem>> base.values
<<subsystem>> base.services

View of a Possible Refactoring Map

This figure shows a first version of a possible refactoring map. It represents the system on the subsystem level. The underlaid area of the map indicates those parts of the system that have already been changed by the refactoring. In our example, until now the refactoring seems to have primarily affected *materials* systems. A single *tools* subsystem has also been included.

a. In this manner, team members working in different locations can benefit from a refactoring schedule.
b. We used the Wiki web option in one of our projects and learned that the Wiki web is easy to handle. It does not offer the same immediate visibility as a poster on a wall or the source code in the IDE though.

c. The technical realization could be accomplished with meta tags in the source code. The refactoring meta tags could either automatically be submitted to the central repository at check-in, or manually inserted in the code. As of yet, no implementation of this mechanism does exist.
d. The various display formats introduced here are only the first proposals. Other formats are also conceivable.

4.2.4 Practice: Refactoring Detail Plan

Problem

A publicly posted refactoring plan that has been discussed by the team is an important instrument for the coordination of work on a large refactoring. However, since it is kept somewhat vague on purpose, it is hard to determine which risks are involved and how demanding the refactoring will actually turn out to be.

During our project work, we were in for a few nasty surprises: our refactorings proved to be very complex although they had looked rather harmless in their flip chart versions.

Solution

The refactoring plan is supplemented by a chart of refactoring details. This chart should be created by a single developer or a pair of developers (in keeping with the Extreme Programming approach of utilizing pairs of programmers) rather than the entire team. The refactoring plan must be specified, breaking down single steps into basic refactorings as far as possible.

High-risk and Low-risk Steps of a Large Refactoring

Nevertheless, this does not mean that a large refactoring merely consists of a series of basic refactorings. Additional modifications are required, for example, if one wishes to exchange the superclass of a class. Modifications for which no safe refactorings are available pose the main risk for a large refactoring. Often it is not clear how such a step shall be executed and what consequences would follow in its wake, i.e. during and after refactoring. In the chart of refactoring details, the distinguishing criterion is whether the single steps qualify as (low-risk) basic refactorings or as (high-risk) other modifications.

Especially high-risk modifications must be analyzed thoroughly. Often it makes sense to begin by simply taking one single step or another. In many cases, the source of the problem will become obvious in a matter of minutes. Once this observation has been made, the respective changes can be discarded.

Visualizing Intermediate Results

It helps to create a graphic representation of the targeted intermediate results as part of the chart of refactoring details (typically using class diagrams). This will help to visualize the larger picture and to stay on top of the overall large refactoring process.

Often large refactorings reach stages where the system structure has already significantly improved. If a large refactoring has reached such a point but is then interrupted, the system structure will none the less be better. If a refactoring is stopped prior to reaching a point of improvement, the system will often still contain detours, and the system structure will have deteriorated compared to the original version. In that case, it is advisable to either undo the refactoring completely or at least revert it to its last stage.

The stages that mark improvements of the system structure are called *save points*. They should be highlighted in the refactoring plan as well as in the chart of refactoring details. If you work with branches, make sure to integrate the branches into the main development process as soon as you reach a save point.

Automating Large Refactorings

When a large refactoring is carried out, the developers will use a number of basic refactorings as a rule, although a large refactoring does not exclusively consist of a series of basic refactorings. It also requires additional development work.

This observation lets us arrive at the conclusion that large refactorings can only be insufficiently automated. It is not enough to plan a large refactoring beforehand, then break it down into small refactorings that can be automated and proceed to apply the sequence of the basic refactorings to the code with the aid of a specialized tool. The developers can execute some step of a large refactorings aided by refactoring tools, but other steps need to be executed manually.

4.2.5 Practice: Assessing Consequences

Problem

In many cases, a refactoring consists of two parts: on one hand, we alter the structure of the code. On the other hand, we also adapt parts of the system to that altered structure. This will be necessary if interfaces or inheritance hierarchies were changed.

Both parts of a refactoring may concern varying amounts of code. Either the changed or to be changed structure itself contains a large amount of code, so that the refactoring will become complicated and comprehensive, or those parts changed via refactoring are used by many other portions of the code. If we modify an interface or a type dependency in the course of a refactoring, the clients of the involved class need to be adapted as well. This may turn out to be a task of considerable scope if many clients exist in the system.

The consequences of single refactoring steps are partially hard to assess. Quite often during a large refactoring one will notice that the scheduled single steps cannot be carried out in the originally planned way.

While a refactoring route may be fundamentally wrong, it can also (and quite often) happen during large refactorings that only certain steps turn out to be faulty, or that other necessary steps have been overlooked in the first considerations regarding the refactoring. Those particular refactoring steps must be reorganized and/or supplemented. In a worst-case scenario, the developer team is unable to plan each single step of a refactoring ahead. Whereas the status quo of the system and the goal of the change are clearly defined, the approach to getting there will be established in the process.

In our development processes, we always assume that software development is a learning process. However, this is not only the case for the implementation of new features. Large refactorings are more time-intensive and will restructure complex and/or central parts of the system. Thus it doesn't come as a surprise that a large refactoring are subject to a learning process, which is not always completely plannable. What we are learning on our way influences the choices for our further proceedings.

Solution

One source of the problem is that we cannot foresee all consequences of a refactoring beforehand. The larger the system, the more complicated it will be to apprehend even a few refactoring steps.

Modern IDEs allow displaying a system's call graph. This feature enables developers to determine from which other methods in the system a particular method is called.

This function can be used to get a first impression of the possible complexity of changes to a method. Through the call graph's visualization, developers can learn with little effort how many places in the system access the scrutinized method, and how this method is typically used.

Of course this function is only useful when the effect of changes to a single method shall be analyzed. Changes to a class hierarchy cannot be analyzed with this function.

Refactoring Prototyping

Many IDEs offer functions for displaying the inheritance hierarchies of a class, but the majority of these tools is not able to analyze the effect of changes to such a hierarchy. Often developers are left with the sole option of prototyping single refactoring steps in a branch. The resulting refactoring prototypes will be able to analyze the impact of changes. Such refactoring prototyping can become quite time-consuming. In order to be able to anticipate future steps, the first ones must have been executed almost completely.

Predictive Impact Analysis

A similar problem exists in the software maintenance debate. Here, the approach towards mastering the situation is the use of sophisticated *impact analysis* algorithms. Impact analysis aims at enabling an analysis of the effects that modifications of a software system create. Not only are changes to an already modified system analyzed (*Comparative Impact Analysis*), but also the possible impact of future modifications (*Predictive Impact Analysis*).

Recognizing a Bad Strategy Early On

The equivalents of these tools can be utilized for more comprehensive refactoring work. They let developers analyze how single refactoring steps will affect a system. This can be useful for recognizing complicated refactoring routes as early as possible. Further information can be found, for example, in [Hoffman 03].

4.2.6 Practice: Branches

Problem

In fact small steps should support us in breaking down a large refactoring. This procedure also aims at reducing the required merge work, because the small increments can continually be integrated in the shared code repository.

However, some refactoring steps can be executed in a couple of seconds with the aid of modern IDEs. In those cases, taking small steps will not be necessary. If a central method or class is changed though, this will potentially lead to many automated modifications to the rest of the system. If the developers commit these changes back to the shared code repository, comprehensive merge work will very likely be the result.

Today, automated refactorings can be found as part of each professional IDE. They permit modifications of many source code texts at once (for example, if we wish to rename a central class). As simple and comfortable this functionality appears to be – in some scenarios it can turn out to be quite tricky. If, after such a refactoring, we find out that we took the wrong step, we have to undo that refactoring. Should the IDE neither support such an undo-functionality, nor should the complementary refactoring exist in an automated version, the developers will face a lot of work to undo the refactoring.

If developers have proceeded implementation of application requirements, the new classes may interfere with the refactoring steps. That makes is very hard to undo the refactoring – the application requirements would be undone too.

Solution

At a certain point a branch is separated from the team's current development work, then the team's developers proceed to carry out the refactoring in that branch, while the whole system is further developed in the repository's HEAD. As soon as the refactoring in the separate branch is completed, it is committed back to the up-to-date version of the system.

Advantages of Branches

This option of conducting large refactorings has a number of advantages:

- The large refactoring can be carried out step by step, and developers can continue to work on a system that runs without interruptions.
- The current working version of the system does not contain any detours, which would otherwise be required because of the refactoring.
- A refactoring can simply be 'rolled back' when the developers discover they've decided on a totally wrong refactoring route.

Disadvantages of Branches

While this option initially sounds quite attractive, it also harbors a couple of disadvantages:

- Developers involved in the refactoring are forced to switch between two different versions of the system if they continue to work on it after the implementation of a refactoring increment, or after integration of a new feature to further the refactoring. This switching between contexts is difficult and can delay a large refactoring.
- If the large refactoring is integrated in the current system, a substantial demand for merges is created, because the system development has advanced. Depending on how strong the refactoring's impact on the system is, the merge demands can be rather high. The further the refactoring progresses, the higher these merge demands will become. One of the major risks of such a proceeding is that a refactoring that has been in progress for a long time will no longer be integrated due to the high merge demand, and thus will eventually be discarded.

- As the case may be, a merging of the current system version and the large refactoring can take a relatively long time. During that time, the whole system is no longer runnable. The more time developers need for the merging, the higher is the risk for the entire team of creating a not fully runnable system.
- Once the refactoring has been fully integrated in the system, the developers who are not directly involved in the refactoring work have to familiarize themselves with the new system, because a lot of code may have changed literally from one day to another.

Our experiences made us realize that separate branches are better suited for larger refactoring projects when the changes brought about by that refactoring can be restricted to a part of the system. This is the case, for instance, if a large refactoring only affects the implementation of a fraction of the system. Here, it is important to observe that the interface to other parts of the system remains unchanged. In the ideal case the respective part of the system can be replaced with the modified version – as long as its functionality hasn't been altered during refactoring. If this part has been changed, these changes must be integrated in the refactored version, but the merge demand will be limited.

Eclipse Runtime: An Example

In the course of the Eclipse project, an alternative Runtime has been developed as part of the work on the software's version 3.0. The new runtime was developed parallel to the scheduled development of Eclipse version 3.0. Halfway between Milestone M5 and M6, the old Eclipse Runtime was replaced with the new Equinox Runtime. This was accomplished with minimal effort, because the runtime is accessed by other parts of the system via a fixed interface. In order to ensure an even smoother transition, the developers of Equinox Runtime attached great importance to making sure that compatibility with the old interface was guaranteed.

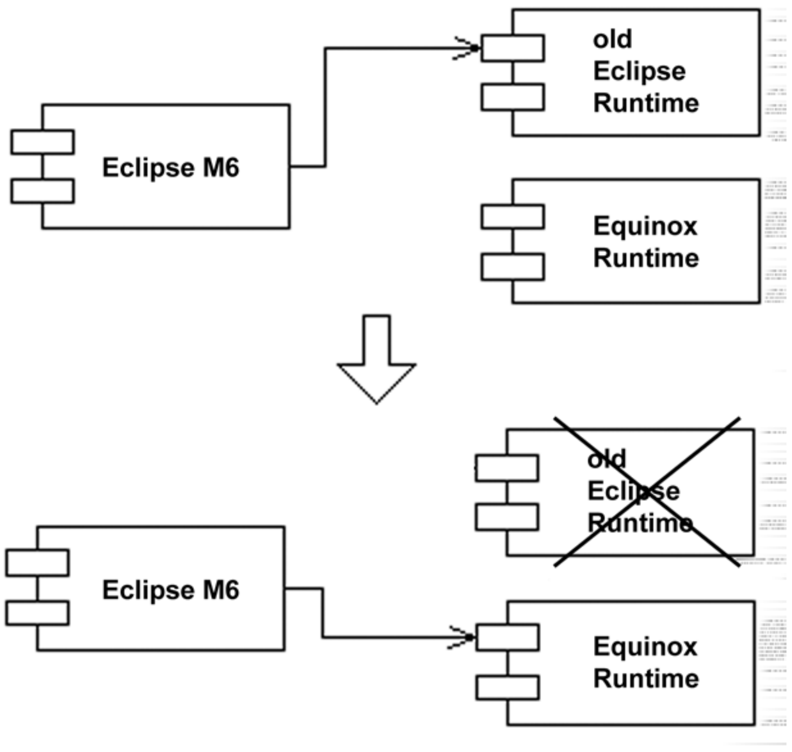


Fig. 4-1
Exchanging Eclipse Runtime

Even though the new Eclipse Runtime is a new feature rather than a refactoring of the old Runtime, this example demonstrates that the part of the system that shall be modified can be particularly well developed in a separate branch and merged later on, when only implementations are modified

Unfortunately, large refactorings are not exclusively limited to implementations of system fractions. This seems logical if the refactoring shall improve the system's structure in a significant part of the system. For such modifications, the disadvantages of the branching approach clearly outweigh the advantages. Under these circumstances, we therefore prefer the integration of large refactorings in the normal development process.

4.2.7 Practice: Acceptance Tests

Problem

In our introduction to the refactoring topic we learned that tests and refactorings are inseparable. Refactorings can only be carried out securely when a good test coverage is guaranteed. Of course, a good test coverage is also one of the prerequisites for the success of large refactorings.

Nevertheless, large refactorings do not only require various modifications of the program code, but also modifications of the test code. The effort for the large refactoring will diminish if one occasionally throws away the odd test and executes a new implementation after refactoring.

Experiences from the C3 Project

A similar approach seems to have been chosen by the developers of the C3 project. On the Wiki web, Chet Hendrickson writes:

"About every 3 or 4 iterations we do a refactoring that causes us to toss or otherwise radically modify a group of classes. The tests will either go away or be changed to reflect the classes' new behavior. We are constantly splitting classes up and moving behavior around. This may or may not affect the UnitTests."

This is why the traditional unit tests no longer offer a stable framework for large refactorings. In each individual case the developers must decide what a unit test failure during a large refactoring means. Was the last refactoring step faulty, or does the test have to be adapted or deleted?

Solution

Automated Acceptance Tests

Automated acceptance tests (as well as function tests) will prove useful here. They will check the system's behavior from the users' point of view, whereas unit tests check the functionality of single classes from the developers' point of view. Thus modifications of unit tests are often needed during refactorings. Changes to acceptance tests are only permissible to a very limited extent though, because otherwise the refactoring would alter the observable system behavior, not just the internal program structure. Let us resume: in the course of large refactorings, modifications of unit tests are allowed, but modifications of acceptance tests are not.

For automated acceptance tests, FIT or Fitnesse can be used quite elegantly (see References). Both tools read tests from HTML tables, conduct tests on the application level and document the results in

HTML. At the same time, the test results are connected with the application via *fixtures*. The fixtures receive their input values from the HTML tables. With these values, they then call system functions and return their function results. *TestRunner* compares the return values to the expected values based on the test specifications, and again the result is documented. This process is visualized in Figure 4-2:

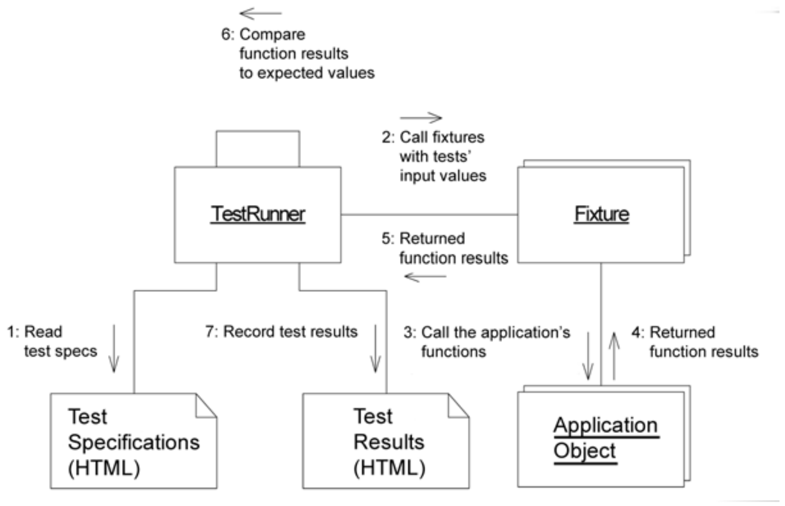

6: Compare function results to expected values
2: Call fixtures with tests' input values
TestRunner
Fixture
5: Returned function results
1: Read test specs
7: Record test results
3: Call the application's functions
4: Returned function results
Test Specifications (HTML)
Test Results (HTML)
Application Object

Fig. 4-2
Acceptance Tests with FIT / Fitnesse

4.2.8 Practice: Detours

Problem

If changes to vital parts of the system are required (e.g. renaming of a vital method), many dependent parts of the system must be adapted as well. During this transition period, the system will neither be compilable nor runnable. Since it is desirable to have a runnable version of the system in the central source code repository, changes can only be integrated when they are complete and the system is runnable again.

Because these changes are rather time-consuming, considerable merge requirements may follow in their wake. Moreover, only after the refactoring is complete, tests can be run to determine if the refactoring process was executed correctly and the system is indeed operational. If the system does not run correctly, it will be very difficult to identify the one error or the errors. In principle, any class that's been changed in the course of the refactoring could be responsible for its misbehavior.

Solution

In order to break down a large refactoring in small increments, detours are built into the code (analogue to the detours for basic refactorings if they are executed stepwise). At the end of a refactoring, these detours must be removed from the code. If the developers additionally integrate single steps of their large refactoring into the shared code repository, the detours that have been introduced into the code will also become visible for other developers.

This leads to a situation that at first sight seems paradox: the detours will first impair the system structure with the goal of eventually improving it. In many cases, the course of a more comprehensive refactoring will look somewhat like this:

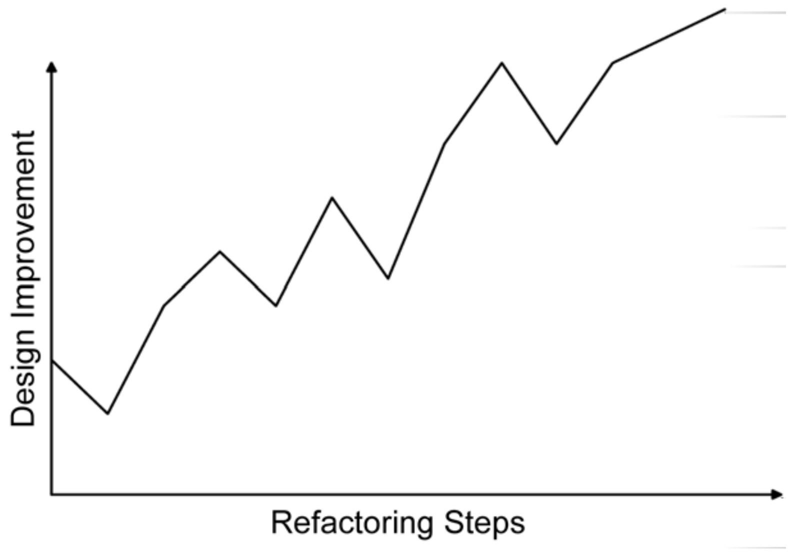


Fig. 4-3
One Step Back, Two Forward

The graphic clearly illustrates that most steps of a large refactoring will improve the software's design and bring the developers closer to the design they are targeting with their refactoring. However, developers will always have to deal with small steps actually leading in the opposite direction. Usually these steps are identified as being wrong at some point and corrected. However, it is important to recapitulate that a large refactoring can, as a rule, not be planned 100 per cent, from start to end.

Based on the aforementioned observation, we pointed out that single steps of a refactoring might turn out to be wrong. Moreover, in the course of a large refactoring, developers may find out that an entirely different refactoring route would have been preferable.

Example: Detours in the Code

The simple refactoring for renaming a method is – unless this is done automatically by an IDE – a good example of such a detour. Let us assume that the following method *print* shall be renamed *printDocument*:

```
public void print (Document obj) {

  ... implementation of print ...

}
```

In a first step we create a new method with the desired name and move the implementation of *print*. We will get:

```
public void print (Document obj) {

  this.printDocument(obj);

}

public void printDocument (Document obj) {

  ... implementation of print ...

}
```

In our next step, we mark *print* as *deprecated*:

```
/**
* @deprecated use printDocument instead
*/
public void print (Document obj) {
  this.printDocument(obj);
}

public void printDocument (Document obj) {
  ... implementation of print ...
}
```

In the next step, we can by and by adjust all places in the code that until now used *print*. In these places we are simply going to exchange the call *print* with the call *printDocument*.

Once all calls have been replaced, the old method can be deleted. It served as a detour for as long as we weren't able to use the new method consistently. During this transitional phase, two versions of the *print* method existed simultaneously. The system's structure was worse than before in this period. Only after the refactoring was complete, the old method was deleted and a better system structure emerged.

4.2.9 Practice: Errors and Warnings as To-do Lists

Problem

Large refactorings harbor the danger of getting lost in minute details. With each step, the number of compile errors grows, and it becomes more and more difficult to integrate a completely functional version of the system into the shared repository.

We want to be able break down even extremely large refactorings in such small increments that a functional system is guaranteed after each implementation step. We will adhere to this goal even if a large refactoring is carried out in a branch.

Deprecated warnings are a popular way of implementing stepwise changes. The old method or class is marked *deprecated*, and the compiler highlights all sections of the code which still bear references to the element marked *deprecated*. The purpose of this mechanism is an incremental transition of the marked references from the old structure or method to the new one. It is easily possible though that the sections marked *deprecated* in the source code cannot be arranged in any desired order. If, for example, a class with *Inline Class* is removed, all references to this class must be replaced with references to the new class. Should a method call another method while simultaneously committing an object of the old class, the calling class must be modified first and generate an object of the old class prior to the other method's call. If the method of the called class is changed first, the calling method must be adapted as well, because it will expect a parameter of the new class's type. Regarding the calling method, the question arises from where it should get the new class's object. The calling method can no longer simply generate the object because it may need to contain more information than the fields of the class to be deleted.

Solution

Do large refactorings in a way that:

A Consistent Number of Compile Errors

- After each refactoring step, a consistent number of compile errors shall occur. This particularly means that the number of compile errors shall not correlate with the size of the respective system, but exclusively with that of the refactoring step. Thus the single steps of a large refactoring can be carried out in as little time as possible.

No Specific Order for Deprecated *Warnings*

- We will often work with *deprecated* warnings in order to execute successive refactorings. It must be irrelevant in which order *deprecated* warnings are processed because otherwise it will become very hard to determine their correct sequence in large systems. Some system cycles might even prevent a stepwise processing of the *deprecated* warnings. Moreover arbitrary removal of *deprecated* warnings significantly simplifies planning and – last, but not least – allows the team parallel removal of *deprecated* warnings.

Continuous Integration

- The single refactoring steps shall leave the system runnable, so that integration is possible on a daily basis.

Behavior Conflicts

Structure and Behavior Conflicts

If we change the type structure of a class hierarchy or single classes of the system, different kinds of conflicts can arise. On one hand, structure or type conflicts, which are noticed by the compiler, can occur. An example hereof are polymorphic assignments. On the other hand, behavior conflicts that will not be found by the compiler can emerge. For instance, this is the case when methods inside the inheritance hierarchy get overloaded, or when the type is checked via *instanceof*, and, as the case may be, a downcast takes place.

Modifications of the Type Hierarchy

- Modifications of the type hierarchy are extremely problematic. They must be thoroughly analyzed and planned. Choosing the right refactoring route is of utmost importance. Even the renaming of a class within a hierarchy can lead to difficulties if it is not automated by the IDE.

Additional Problems through Overloading

- The overloading of methods in an inheritance hierarchy can lead to unsolicited behavioral changes during large refactorings. For instance, if we change a method's signature, a number of problems will follow in the wake of this change: if the original method overwrote a method from the superclass, this must not necessarily be the case with the changed method. The opposite can also occur: a changed method in its new version unintentionally overwrites a method from one of the superclasses.

Relocating Methods

- The relocation of a method to a superclass can cause difficulties when the same method already exists in the superclass, but with a different kind of implementation. This raises the question if the implementation can also be adopted in the superclass. If not, the method cannot simply be moved to the superclass.

4.2.10 Inline Method

Problem

During stepwise refactorings, in most cases old and new structures will exist side by side for a limited time. The old structures are marked with the *deprecated* tag. This procedure primarily serves to track references to the old structure and incrementally remove it. In a Java environment this is particularly easy to do, because the compiler lists references to *deprecated* classes and methods with corresponding warnings.

Developers will often proceed to search all references for the deprecated class or method and adjust the according code to the new method or class. For a very comprehensive system this can cause a lot of work.

Solution

In an article for the XP-2003 conference, Tammo Freese suggests using the *inline method refactoring* in such a refactoring process. The basic idea is an implementation of the method marked *deprecated* based on the new method. The next step is to disperse the deprecated method via inline method.

Example

Let us look at a brief example: We wish to replace the method *print* with *printDocument*. For this purpose, we already marked the old method *print* as *deprecated* and moved its implementation into the new method *printDocument*.

```
/**
 * @deprecated use printDocument instead
 */
public void print (String doc) {
  printDocument(new Document(doc));
}

public void printDocument (Document obj) {
  ... implementation ...
}
```

Now, we will find several calls of the old method *print* in our system's source code, for example:

```
...
String myDocument = ...;
...
myPrinter.print(myDocument);
...
```

If we proceed to conduct an inline method refactoring of the method *print* with the aid of the correlating IDE, all calls of the old methods will be replaced by its implementation. Cleverly, we implemented the old method in such a way that it simply calls the new method (while converting parameters or return types, if these have changed, where applicable). Thus, after inline method refactoring, the reference to the old method will be directly replaced by a call of the new method:

```
...
String myDocument = ...;
...
myPrinter.printDocument(
              new Document(myDocument));
...
```

Using Inline Method for Large Refactorings

In his article, Freese takes this approach even further and shows how refactorings can also be used for APIs. For the large refactorings we are surveying in this chapter, the simple handling of the case described here will be sufficient in most situations. The inline method refactoring lets us elegantly alter those incidents in the code that call *deprecated* methods. Since for an inline method refactoring it doesn't matter how many occurrences in the code must be modified, this refactoring is very helpful in the restructuring of big systems.

Limitations

Of course the inline method refactoring will work only if the old method, marked *deprecated*, can be implemented based on the new method. Should it not be feasible to move the implementation into the new method because both implementations are needed, an inline method refactoring does not make sense.

The example we just gave also shows that the inline method refactoring can in some cases introduce exactly the kind of 'pollution' that was supposed to be eradicated by the new method (here the use of the string instead of the class *Document*), to the calls.

4.3 Fragments of Large Refactorings

After we have discussed organizational and development process-relevant aspects of large refactorings, we will now deal with functional patterns that can help us with our large refactoring work. However, we must admit that the field of large refactorings is of yet very young. Therefore, we are unable to provide a catalogue similar to Fowler's refactoring catalogue (see [Fowler 99]). It may well be possible that there are too many variations of large refactorings to allow the creation of a refactoring catalogue.

Nevertheless, we would like to take a first step towards creating a refactoring catalogue and present fragments of large refactorings. In the last chapter, we saw that architecture smells accumulate in relationships *between* classes, packages, subsystems and layers: often lumps must be disentangled.

A project-specific analysis of what refactoring route makes sense to remove an existing architecture smell is required. In these refactoring routes, certain fragments will recur.

4.3.1 Moving Classes

It is amazing how many architecture smells can be eliminated by simply moving classes. Often cycles between packages, subsystems and layers do not imply the existence of cycles between classes.

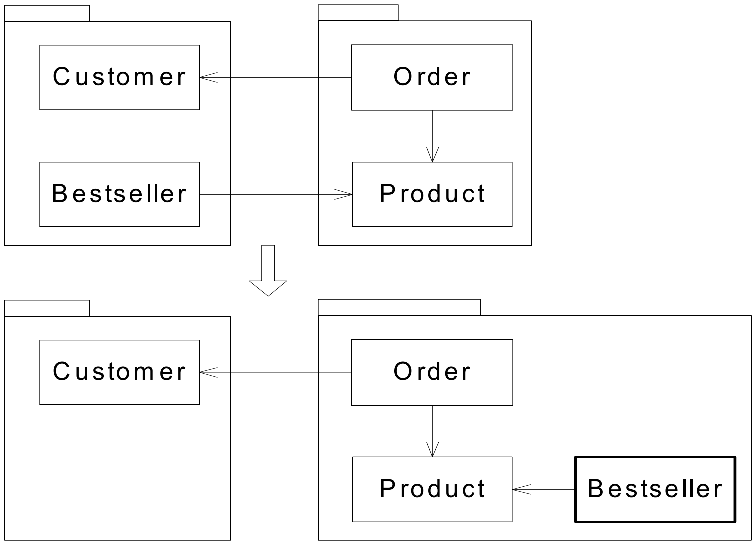


Fig. 4-4
Moving Classes

One should be careful though not to make the mistake of moving classes around without heeding their meaning, before all cycles have been eradicated. If single packages, subsystems or layers lose their [internal coherence], the damage will be greater than the benefit.

In Java, the moving of a class usually means that the class is put into a new package. Since the package name is part of the [fully qualified] name of the class, moving means first merely changing the name of the class. In the case of Java, it should be taken into account that package-wide visibility is given (*protected* modifier or leaving out package visibility). Thus moving a class into another package could cause visibility problems.

The single steps for moving a class or an interface are:

1. If the class or the interface are merely visible package-wide: Set class/interface to *public*.
2. Set all attributes and methods in the class/the interface that are visible package-wide to *public*.
3. For all attributes and methods that are *protected*, check if they are used via classes/interfaces of the same package. Also, set attributes and methods to which this applies to *public*, too.
4. Change the package of the class/the interface.

- If you are dealing with a development environment that offers refactoring support or a refactoring browser, consider yourself lucky. It will allow you to alter the package name automatically.
- Should there be no support available for automatically changing package names, this process will be quite arduous. This is because the package name of a class cannot be changed in small steps; just as this is impossible for renaming a class (the reason why [Fowler 99] does not offer a refactoring labeled *rename class*).
- In this case, you'll have to swallow the bitter pill: Change the package of the class/interface and then – one by one – fix all error messages. You can either do that in a branch, or you commit the class with a new package name. Afterwards all developers have to adapt all references in a single, concerted effort.

The renaming of a class with IDE support can be made easier if you use the following little trick: Instead of renaming the class directly, introduce a new class with the respective name. The old class's method declarations are copied into the new class and a delegation will be implemented. To this end, an object of the new class must be referenced to an object of the old class, and a temporary cyclic uses relation between the old and the new class be introduced. This uses relation enables skipping between types within the system. It is indispensable to generate a new type for each object of an old type (and vice versa). Thus, all references to the old class can be adapted to the new class step by step.

This approach will be problematic though if the class to be renamed possesses subclasses, because subclasses can either only inherit from the old class or from the new class.

4.3.2 Introducing an Dependency Graph Facade

In order to structure dependencies between packages, subsystems and layers it can be useful to hide a number of classes behind a facade. Whereas in [Gamma et al. 94] facades are employed to simplify the handling of multiple classes, we can also use a facade to hide a subsystem's dependency graphs from the client of that subsystem. This allows easier modifications of relations between single classes within that subsystem without having these modifications affect the subsystem's clients.

In the context of the refactoring described here, we assume that the client only depends on the class graph via uses, but not via inheritance. If inheritance relations exist, these must first be removed, for example by replacing them with uses relations.

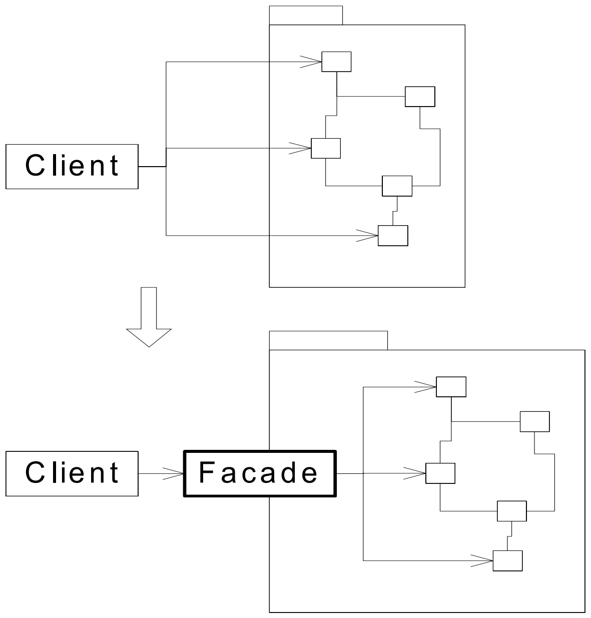


Fig. 4-5
Introducing a Facade

We proceed on the assumption that the client requests all objects to be encapsulated directly or indirectly from a root object. This means that the client uses the class of the root object as well as those classes used by the root object via *getMethods*. This situation can easily emerge if the 'Tell don't ask' principle has been violated.

The facade can be introduced in the following steps:

1. Create the facade class, which will generate a root class object in the constructor. The facade class contains the same constructors as

the root class. Add a method to the facade class that can request the encapsulated object[1].

2. For each method that calls a client on the root class, create an identical method in the facade class. The methods in the facade class call the respective methods in the root class.
3. Step by step, change all instantiations of the root class to the facade class. Then directly invoke the encapsulated root object on the facade object and proceed to work with the latter.
4. Delay the calling of the root object incrementally and change the method calls from the root object to the facade object. Proceed stepwise. Depending on the circumstances, it might be possible to ease the search for method calls that must be altered by temporarily setting the root class methods to *deprecated*. Add a comment to the source code stating that the *deprecated* tags are only transitional. Otherwise, some over-eager colleagues of yours might accidentally delete them.
5. Proceed similar to step 4 with all objects that are directly or indirectly referenced to the root class.
6. Remove the temporary *deprecated* tags.
7. If applicable, reduce the visibility of the root class to the package, so that the class can only be accessed indirectly via the facade.

The introduction of a facade resembles the 'Hide Delegate' refactoring described in [Fowler 99], pp. 157.

4.3.3 Moving a Class within the Inheritance Hierarchy

Errors in inheritance hierarchies emerge very fast during development. They can have quite unpleasant consequences. The system gets flooded with *instanceof* type checks. The inheritance hierarchy will become difficult to understand and to extend. The desired flexibility through polymorphy becomes a source of errors.

Therefore, classes must frequently be moved within the inheritance hierarchy (see Figure 4-6).

1. We assume that the root class itself is not going to be used as a facade here to affect its interface as little as possible.

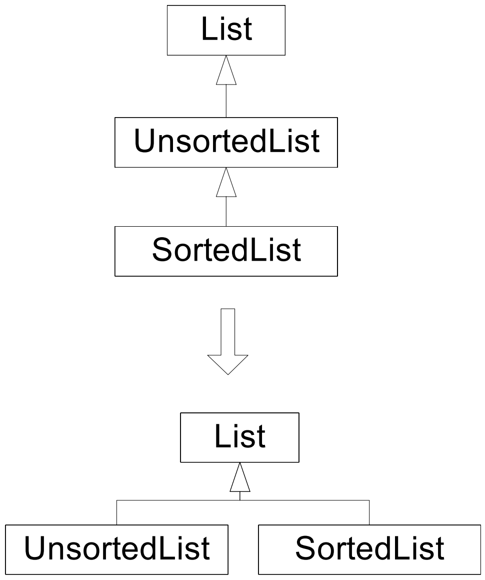


Fig. 4-6
Moving a Class in an Inheritance Hierarchy

Modifications of the inheritance hierarchy are problematic, particularly the moving of classes within an inheritance hierarchy. The resulting problems and type errors of polymorphic assignments are often not curable in a step-by-step revision. The complete system will only be flawlessly compilable when all type errors have been erased.

The following single steps can at least contribute to alleviating this problem:

1. To move a class within an inheritance hierarchy, first create a new class in the selected place.
2. Copy the old class's implementation into the new class.
3. Set the old class to *deprecated*.
4. Work off the *deprecated* warnings and step by step adapt all occurrences of the old class in the code .
5. Once the old class is no longer in use, it can be deleted.

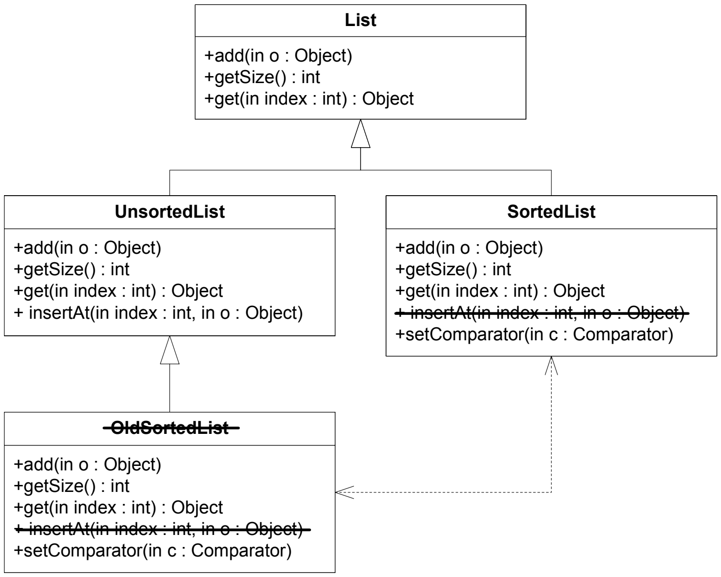
Setting Inheritance Relations to deprecated

To support step 4 in a more elegant fashion, we would like to see the compiler alerting us to all polymorphic assignments of the old class as well as their superclasses. Instead of setting the old class to *deprecated*, we'd rather mark the inheritance relation between the old class and its superclass as deprecated. The compiler should then issue warnings to indicate where the obsolete inheritance hierarchy is still in use. This is, for example, the case with polymorphic assignments.

We have implemented an Eclipse plug-in as a prototype functionality in the course of the JMigrator project (see [JMigrator 04]).

Adapters Ease the Refactoring Process

In addition, an adapter construct can help to simplify the transition from the old subclass to the new one. The trick is to introduce a temporary uses relation between the old and the new subclass. A method *getOld* that is also temporary can request an according object of the old subclass from an object of the new one. At runtime, there will always be an object of the new subclass and an object of the old subclass as a pair, and the new subclass delegates its methods to the old subclass.



List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+ insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
~~+ insertAt(in index : int, in o : Object)~~
+setComparator(in c : Comparator)
~~OldSortedList~~
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
~~+ insertAt(in index : int, in o : Object)~~
+setComparator(in c : Comparator)

Fig. 4-7
Temporary Uses Relation between SortedList *and* OldSortedList

If a class has until now used the old subclass *OldSortedList*, the method can be adapted to *SortedList* in a rather segregated manner. If an invoked method still expects an object from the old subclass, the respective object can be requested from *OldSortedList* via *getOld* and passed on to the calling method. The opposite proceeding is also feasible.

Comparisons can be problematic if you use the adapter solution. Naturally, the methods *equals* and *hashCode* must be implemented in Java in such a way that the old object and new object are identical. If you check for identity in the system using ==, disparity between the old and the new object will be stated. For each single occurrence, you must decide whether this behavior is desired or not.

Since == does not constitute a method, it is usually not possible in development environments to get a report of all occurrences in the source code where objects of a type are compared to ==.

4.3.4 Changing Class Inheritance to Interface

Inheritance couples classes more strongly than the implementation of interfaces. Whereas in inheritance relations between classes the subclasses must on principle know the superclass's supervisory data flow, this is not the case for the implementation of interfaces. After all, an interface alone will not implement a supervisory data flow. Of course a class that implements an interface must often know in which context the single interface methods are called by clients. All in all, the dependency is expressed more explicitly though.

Of course inheritance between classes continues to be useful. When dealing with inheritance between classes from different subsystems, you should check if it isn't smarter to have the subsystem define an interface via the superclass.

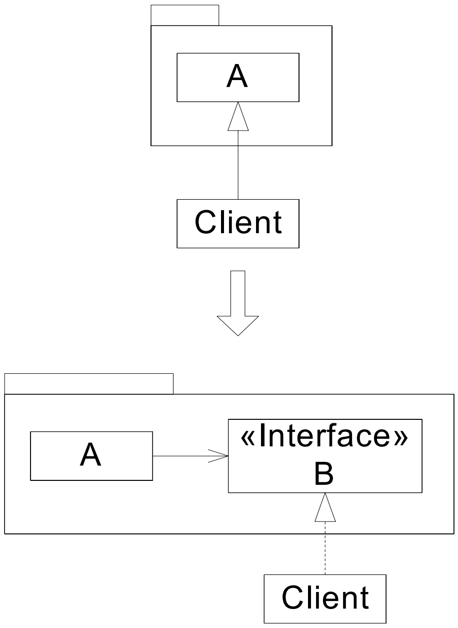


Fig. 4-8
Changing Class Inheritance to Interface

For this refactoring, you will have to execute the following steps (see also Figure 4-9):

1. Create an interface B with all methods from A.
2. Let interface B implement all subclasses of A that are located outside this subsystem:
3. Step by step, add default implementations from B's methods to each client which has not yet implemented them. The default implemen-

tations must display the same behavior as those methods of the same name in A.

4. Adapt A incrementally in such a manner that instead of hook methods, methods from B will be called. This will temporarily create a situation in which A will know the clients via inheritance and use them at runtime (step 3 in Figure 4-9).
5. If A does no longer call any methods on itself which may have redefined clients, delete inheritance relations between the clients and A.
6. In the second step, the new methods of the clients have been implemented in such a way that they adopted the behavior of A's methods of the same name. This may have created redundancies that can be removed by placing the redundant implementations in a help class.

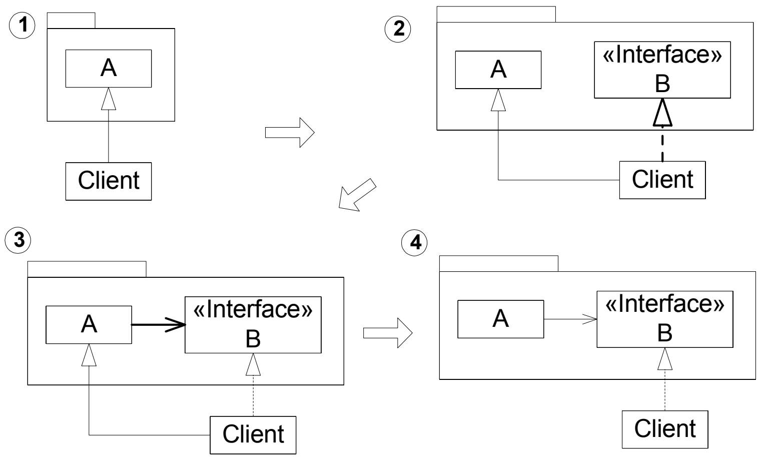


Fig. 4-9
The Single Steps of the Refactoring

The switching from class inheritance to interfaces is a typical step to be executed when a white box framework is further developed into a black box framework (see [Foote & Opdyke 95]).

4.3.5 The Classic Removing of Cycles

Software engineering knows a classic, universal procedure for removing cycles between two artifacts A and B. To this end, B is split into two segments B1 and B2, so that B1 is used by A and B2 uses A. Depending on the situation, either B1 and B2 are independent from each other, or B2 must use B1 (see Fig. 4-10, here the variant on the left hand side). The opposite case – having B1 using B2 – would be a mistake in the selection of B1 and B2. Then A would again be part of a cycle (A->B1->B2->A, see Figure 4-10, the variant on the right).

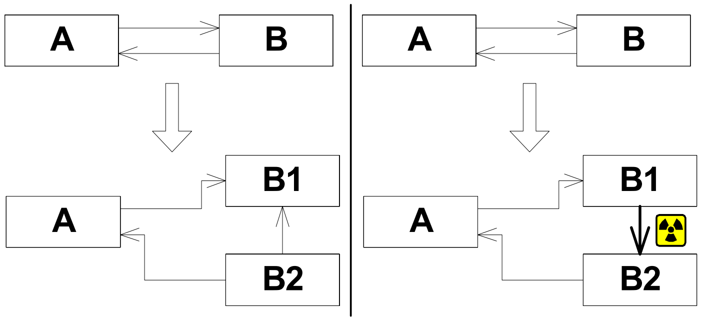


Fig. 4-10
A General Procedure for Removing Cycles

It is also thinkable that a cyclic dependency exists between B1 and B2. Then the procedure depicted here must also be applied to B1 and B2. In this way, the cycles can iteratively be made smaller and smaller until they eventually disappear altogether (see Figure 4-11).

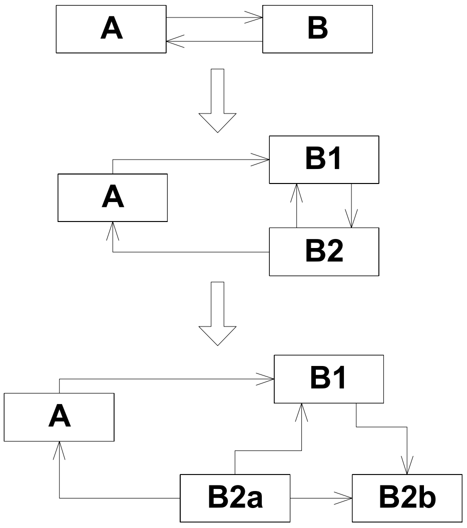


Fig. 4-11
Iterative Removing of Cycles

This method can be used to remove cycles between classes, packages, subsystems or layers.

The previously described moving of classes works like the procedure depicted here, if the moved class is delegated to a new package or subsystem.

The procedure presented here will work universally, but without utilizing the possibilities object-orientation offers: The *removing of cycles with DIP* (see next section) utilizes the inheritance relation for the removal of cycles. This method will often let you remove cycles between classes in a smart fashion.

4.3.6 Removing Class Cycles with DIP

Cycles between two classes A and B can be removed in an elegant manner when DIP (*Dependency Inversion Principle*, see [Martin 97]) is used. For this purpose, we introduce a new interface that contains all methods that A calls on B. A only knows the interface that is implemented by B (see Figure 4-12).

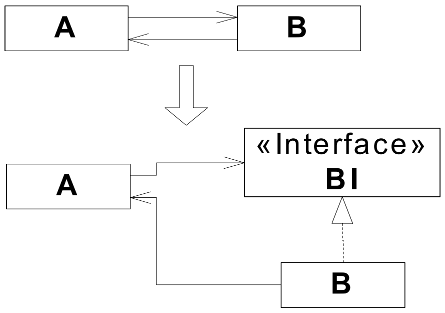


Fig. 4-12
Removing Class Cycles with DIP

If you compare Figure 4-12 with the structure depicted in Figure 4-10, it becomes obvious that the dependency relations turn out to be identical after restructuring. Only the nature of B2's dependency from B1 will have changed.

This refactoring requires the following proceeding:

1. Extract interface BI from class B. The interface must contain all methods of B that are needed by A. B implements BI.
2. In A, set all references that are not required for object generation from B to BI.
3. If references exist in A for the generation of B, introduce a plug-in (see next section).

You will experience the limitations of this refactoring when A does not only use class B, but also generates instances of B. After all, A cannot generate instances of the newly introduced interface BI. Clearly, the generation must be relegated from A. This can happen if, for example, the client of A generates instances of B and passes them on to A. Alternatively, you can introduce *plug-ins* (see next section).

4.3.7 Introducing a Plug-in

It is possible to use interfaces to reduce couplings – particularly between subsystems. A client will then no longer directly use a certain class, but only an interface. Thus it is feasible to use any classes at runtime, as long as these classes implement the used interface.

However, this does not answer the question of where the objects implementing the interface come from. If the client itself generates the objects, it must know the concrete classes for their generation. On top of everything, the client must also implement the case statement which serves to determine from which class the object shall be generated.

One solution to this problem can be found in the *plug-in* pattern (see [Fowler 03], pp. 499). The interface defines an extension point into which plug-ins – the classes implementing the interface – can be plugged. To enable the plug-in's integration into the system with as little effort as possible, the classes are registered in a *PluginRegistry*. Objects of the plug-in classes can be generated using the *PluginRegistry* (see Figure 4-13).

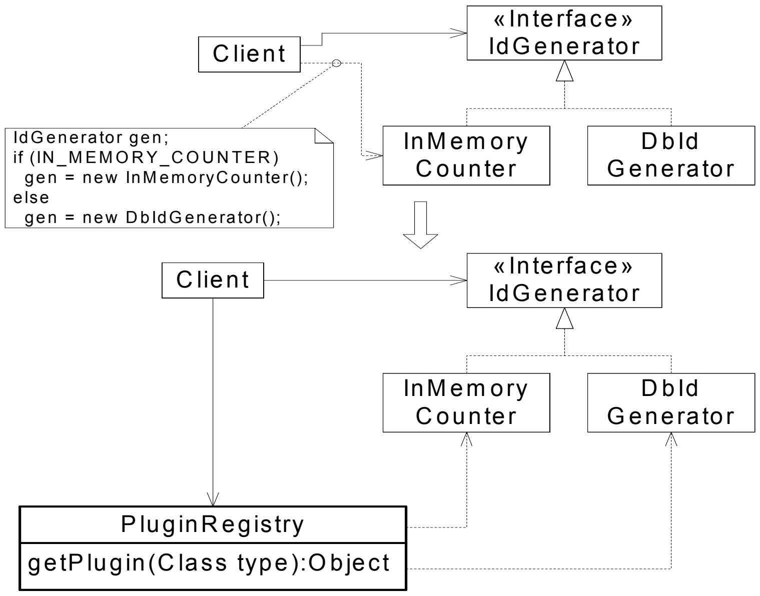


Fig. 4-13
Introducing a Plug-in

The following single steps must be taken for this refactoring:

1. Search for all references to those classes that implement the interface. Replace all references that are not used for object generation with the interface. If the interface does not provide a required method, pull it up into the interface.
2. Program the class *PluginRegistry*. The *PluginRegistry* will be parameterized with information that allows you to find the suitable class for a requested interface. Alternatively, the *PluginRegistry* itself can get the needed information (for example from a property file).

3. Search for all remaining references to the concrete classes that implement the interface. These references can only be object generations, since all other references have been eliminated in the first step. Replace these generations with calls of the *PluginRegistry's* generation method.
4. To prevent the direct instantiation of subclasses in the future, it is in many cases advisable to set the subclasses or at least their constructors to package-wide visibility. It is recommended that you place the interface, the *PluginRegistry* and the classes implementing the interface in the same package.

The development of plug-ins is a logical consequence of DIP (*Dependency Inversion Principle*). They also facilitate the reduction of dependencies during object generation (see also [Fowler 04]).

Today, the plug-in concept is discussed in various literary sources. Besides Fowler, Evans describes a similar design for domain modelling with his *Pluggable Component Framework* (see [Evans 03], pp. 475).

The Eclipse development environment offers a plug-in model that enables third-party vendors to expand the development environment by their own plug-ins (see [Bolour 03], [Daum 03]). The Eclipse development environment itself is based on plug-ins. Therefore, it is possible to use the Eclipse plug-in mechanism without the development environment for application development. The developers of Eclipse used this ability to build the *Eclipse Rich Client Platform* for application development.

Excursion: Refactorings are the Work of Human Beings

A contribution by Dierk König, Canoo Engineering AG (dierk.koenig@canoo.com)

In dealing with refactorings, organizational and functional aspects must be observed. These can be described by scrutinizing the processes and tools that are involved. This is the objectively comprehensible side of refactoring processes.

Moreover, there is another side that is much more elusive: this is the part concerned with the people involved and their interactions[a]. This is the area that I will try to chart here, and I am going to introduce an explanatory framework which helped me at least to find some direction.

In the year 2003, we conducted a nice project: about 7 developers dealt with web technology and a strong database component for more than 5 months. On the agile practice side, we had soon installed automated testing, continuous integration, brief release cycles as well as incremental/iterative development. We could cooperate locally and also were in close proximity to our customer. Pairing took place only in part. And the scheduling procedure was an entirely different story...

The ‘basic’ refactorings didn’t pose a problem. Thanks to the support of conventional tools, only a few errors occurred. Those were recognized and intercepted by functional tests. Typical error sources were the symbolic references in Struts/JSP.

We even managed to get a grip on the ‘common’ database refactorings including adjustments to the schematics. Here, the first human aspect came into play: All of us had to simultaneously work on the same database instance, and we constantly stepped on each others toes. This fact continued to get on our nerves until at last one developer took the initiative and
– without it being scheduled – extended the database abstraction layer in such a way that everyone could ‘virtually’ use their own database instance[b]. Time needed: about half a day.

Now we were able to completely erase our own databases prior to each test run, newly construct the schematics and populate them with data. Afterwards, this part ran smoothly.

In the middle of the third iteration, things started to become critical...

Our database abstraction layer and our entire shared work on the code base – our architecture, if you will – became increasingly unclear. Our developer with the strongest knack for architecture took on the task of changing that, saying he wished “to clean up here”.

This task had a clear-cut, functional aspect, the usefulness of which was unquestioned. But our developer decided to tackle another aspect that was more about how ‘one’ can solve such a problem ‘correctly’ and ‘elegantly’.

Our efforts to solve the problem dragged along. For many days no commit would be entered into the repository. A week passed. A second week went by. The developer didn't explicitly refuse pairing offers, but he clearly preferred to work alone. He was also against committing intermediary increments that weren't 'perfect' yet. Team members piped up: "What is he actually doing there?", "Do we really need this?" and so on.

He repeatedly had to interrupt his work to provide support in his field of specialization. This led to further delays.

To keep up with team's progress, he had to increase his synchronization efforts[c].

In the end, the required integration demand was extremely high by our standards.

We all acknowledged that the new solution actually was an improvement. However, we regretted that it had arrived so late it could no longer be fully effective, and that it had cost so much precious project time.

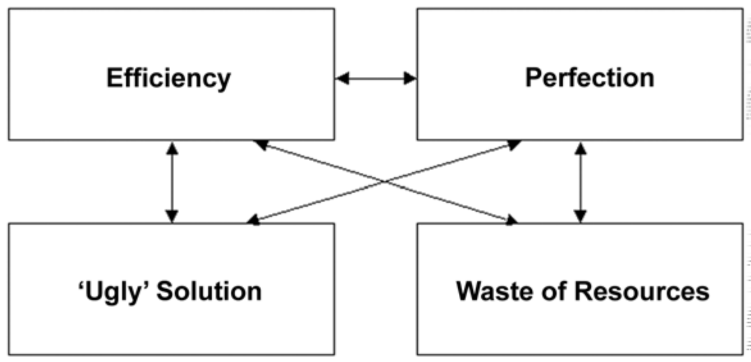
Non-one was really happy with that large refactoring. The solution was not 'perfect' yet. The effort that had gone into it wasn't justified by the result. Spirits were low. Should we have opted for another architecture right from the start? Should we have done without the refactoring altogether? Both seemed wrong alternatives to us.

We certainly would have gotten a better result if we had been able to read this book before taking on that project. Then we would have had:

- the whole team plan the large refactoring; and
- conducted it in pairs; and
- realized it in small increments.

However, the question of how we could have dealt with different tendencies in the team remains: Is efficiency more important ('good enough' and quickly developed), or perfection (the only way to do it 'right'). What is valid? What is better?

I am not able to take sides with one party. Both are right. In my view, a fifty-fifty compromise is not an adequate solution here. Instead, I tried to look at the positions from a systemic standpoint.



Development and Value Diagram

This is a development and value diagram according to [Schulz von Thun 89].

- Efficiency is a value, and it is good to achieve it. Let's say: E represents this value.
- Perfection is also a value, and it is good to achieve this value too. Let's say: P represents this value.
- If you overdo your striving for efficiency, you will obtain a solution that is 'ugly'.
- If you overdo your striving for perfection, you will waste resources.

The arrows in this diagram are pivotal. The diagonal arrows represent fears and (possibly unspoken) accusations:

- P accuses E of building 'unclean solutions,' having no sense of quality etc. At the same time, P is afraid of being reproached for the same things.
- Vice versa, E accuses P of wasting resources. On the other hand, E fears that he could face the same accusations.

One should keep in mind that "fear and aspiration are siamese twins". [Schulz von Thun 89]

- P would like to be as efficient as E, if he could only maintain his quality level at the same time.
- E would like to be as perfect as P, if this were possible without losing time.

This recognition forms the basis for a development approach that unites the striving for perfection with the striving for efficiency.

This approach does not imply that you do less of what you think is important. It is about uniting one's own position with the other's point of view, so that you – speaking in terms of our diagram – 'develop upwards'.

How can this be accomplished?

Sometimes it is enough to let all team members participate in the creation of the development and value diagram, to let them find their own solution. If no solution is found, here are a few suggestions:

- The advice to take small steps is backed up by another argument: small steps lead to an added value within a short time, and this will appease E. Smaller tasks can be handled more easily in sufficient quality. This will alleviate P's fears.
- The team's 'Go!' in favor of a refactoring usually diminishes the risk of accusations.
- E and P team up in a pair for the refactoring to keep each other in line. This is not possible if both – which is often the case – have problems on the relationship level. Such problems must be settled first. "[The] technology [group] (detached, controlling, bent on proving themselves and aggressive-debasing) [has] to learn most in this scenario: Used to operating argumentatively and solution-oriented on the content level, they will often oscillate between exaggerated distancing themselves from others, dogmatism, helplessness and aggression on the relationship level." [Schulz von Thun 89], p. 248.
- A coach can help to hear voices coming from the lower corners of the diagram, to put them into perspective and offer possible supplementing values. Typical phrases are: "let's finish it quickly before...", "only this one here" and on the other end "one", "correctly", "elegant", "architecture" etc.

As our project's coach, I wasn't as successful with this approach as I would have liked to. As far as I can judge, I could contribute to fostering the mutual understanding and appreciation between those involved, but their actual behavior did not change visibly for the short term.

Mutual respect is the basis for each progress in a dialogue. Once Kent Beck and I had a long e-mail discussion with differing opinions. He finally replied: "Progress comes from the disagreement of friends." There is a whole world of meaning summed up in this one sentence.

What is valid for a team can also be valid for a single person. Friedemann Schulz von Thun explains in his book the analogy between a team that consists of various persons and the various 'voices,' that are united inside a single person (see [Schulz von Thun 98]). He calls this phenomenon "the inner team".

Whenever I am facing a refactoring, I can feel the dispute between E and P in myself. At best, a programming partner will be at my side, with whom I can discuss openly to resolve stalemate situations. At worst, I will reproach myself until I have a bad conscience or until I suppress any thought of either E or P.

If the previously described model finds your favor, you can find an even wider field for its application, for example:

- To counteract the fear of an unnoticed introduction of mistakes through refactorings and of 'encrusted' code;
- To counteract project-bureaucracy (fear of loss of control) and hacking (fear of loss of freedom);
- To obtain a concrete, detailed view of the code, e.g. through unit tests that deliver fast feedback and an abstract, architectural view – for example with the aid of Sotograph[d]
- Etc.

Where do I stand in this system if I either adamantly refuse to apply Big Design Upfront, the Life Cycle model, or MS project charts, or if I think that they are indispensable?

a. The Agile Manifesto: "...people and interactions over processes and tools...".
b. Realized with conventions for the table names.
c. Merging of the HEAD with its branch.
d. I call it "programming distance".

Excursion: Sustainable Architecture

A contribution by Klaus Marquardt, marquardt@acm.org

Redesign

You may be familiar with the following situation because of your own, painful experiences: A project has been finished, but with a lot of stress for all involved. Many goals have been reached, but the more experienced developers are left with a very bad feeling: They know that the code basis will continue to exist, but on the way to the last milestones, too much of the originally wanted structure has been changed and undermined by faulty, half-hearted solutions. They'll walk up to their boss or customer and ask for two team months to carry out a 'redesign,' i.e. to 'clean up' – and they won't get permission to do so.

When I assume the customer's position, I can perfectly understand this decision. The project came dangerously close to failing, now it's time for it to make money. If I were in the customer's position, I wouldn't allow any further budgeting without seeing clear cut advantages for my business either. What value can an investor possibly gain from a struggling team that even admits to working sloppily, if not even new functions are being added?

Dependency Management

There are projects where the software's internal structure directly serves to achieve business success. I will use the project Olymp[a] as an example here: Olymp is a plug-in architecture[b] for the software of a family of medical devices. The basis of all these devices forms a framework for shared functions and abstractions. Via this framework, specific domain components (applications) are implemented as plug-ins. Products are created by 'plugging together' various applications with as little integration effort as possible. The software required for integration is also realized as a plug-in.

This architecture suits the underlying organizational structure very well. Each department can manifest its specialized knowledge in an application software. Each single department is responsible for its software. Each product has a definite source, which is also responsible for production, marketing and customer relations. However, all applications profit from the extension of the framework, and all products will potentially profit from powerful applications. The architecture enables parallel development as well as congruency of tasks and competencies; the placing of functions rather 'low' in this building set system increases their reusability and also fosters a uniform exposure of the products to and behavior in the market.

On this level, the architecture of Olymp consists almost entirely of the definition of responsibilities, dependencies and their management. All further technology and complexity are secondary to these aspects. This level directly supports the organization and its internal business model. Thus the existence and maintenance of the architecture makes sense. Refactorings that serve to maintain the structure or increase its reuse will be actively supported by the investor.

Recognition

Each of these different plug-ins contributes to a series of layers, among others, data, rules, algorithms, displays and operating elements. This view builds orthogonally on the separation of single plug-ins. In its entirety, the structure of the whole software is almost ideal for a static analysis. As a matter of fact, I built a kind of software tomograph for my own purposes and used it to analyze the system in rather long intervals. Due to system's clear organizational structure, I was very rarely surprised.

If I feel that dependency structures are important, these violations will document a communication problem. Violations that are discovered in the course of formal checks are always found too late, and pointing them out will only be partially convincing. This is why I always perceived the use of these tools as a last resort which would offer me an apparently objective justification for my goals.

It is much more difficult to assess if a class or a package have been assigned to the correct unit regarding its task. I have yet not come up with an idea for an automated test for this purpose. Instead, I made the question what would be the right location into a standard issue in design reviews. For each possible placement, I worked out criteria and defined a specific order stating which unit should preferably contain classes.

Relocation within the logical structure frequently occurred in the course of the project. It was suitably infamous, and eventually it was dubbed ‘Cat-Ball’[c]. Relocation wishes stated in the design reviews were not always popular, but for the sake of the greater good they were accepted and the refactorings executed.

No Change Without Suffering

Actually, several projects are part of the Olymp architecture: one for each plug-in. The plug-ins in turn contain subprojects, because these usually comprise code for various processors and embedded systems. In such a complex system, it is hard to make progress, especially during the early learning phases, because each change caused by a refactoring has political consequences. To reach easier controllability and escalation paths, the first of these projects were united under a common management. For some projects, this would happen by and by.

Nevertheless, fundamental changes concern many places in the code of various sub-projects. In most cases, the developers that are involved perceive refactorings as disturbances of their routine – after all, their own code works well, and they will not experience any improvements that concern them. This perception goes as far as having an imaginary barbed wire fence run around one’s ‘own’ field of work; a fence that has even been established by the immediate project manager. Our motto for compromises made under a common project management was: *Those who want to bring about change must suffer.* The person who carries out a refactoring is also responsible for modifying the entire code of all affected plug-ins right away and for getting the refactoring to run properly.

In spite of this at the first glance frightening prospect, this proceeding has proven to work well in the Olymp project. It reduced the developers’ fear of interferences, because no-one could be accused of having introduced thoughtless and arbitrary changes. At the same time, the path was cleared for really important changes. We decided we would (and wanted to) do without aids such as *deprecated* tags. Last but not least, *deprecated* means that we are dealing with a ‘slow’ refactoring – one that has been partly put off to maintain compatibility. Such careful approach was not necessary for a clear-cut, comprehensive project under common management. The prerequisites for our approach were a certain amount of shared code ownership, continuous integration, automated build & test, active support by the version control system, as well as a team with a common goal.

Inside the Booth

Developments and modifications that cannot be completed in the course of a few days or that require the combination of various expert competencies are separated from the normal development process and relegated to a booth. A booth is a separate branch of the version control system that is being run parallel to the main development process. As a rule, each developer (or pair of developers) will have their own branch and deliver their results into the integration stream. Here, a baseline will be drawn every couple of hours or days. Prior to delivery, each developer must synchronize with the latest baseline and carry out the required merges[d]. This leads to pressure in exactly the right place: Synchronization with colleagues can be timed individually, but those who neglect synchronization for a longer period will eventually have a lot to catch up with. However, it is important that the decision when to synchronize can be made individually and is thus able to suit each project situation as well as each work style.

Bigger refactorings, like those that concern an API and several components, can even take a couple of weeks. Afterwards, the colleagues in the booth must merge a lot – unless they had the foresight to regularly synchronize with the current stage of integration during this period. This synchronization cannot only affect the main branch, but also occur inside the booth.

The booth creates a setting which makes sure that customers who use a component or interface will not notice the modifications that are going on, nor are they forced to make the necessary adjustments themselves. Instead, they can go through with their original plan. As long as all projects and partial projects can be handled as a unit, no compatible interface must be serviced.

Active, But Patient Waiting

Once the software architect working with such a system has completed the preparatory work of creating a fitting structure and mutual understanding and has established an adequate work process, he can lean back and relax a bit. Further interventions are not productive as long as the developers are coping well – on the contrary, it is more likely that they would evoke defensive reactions from the developers. Nevertheless, the architect must be alert at all times and respond at once when problems or irregularities emerge.

This status quo reminds us of the work technique that is occasionally dubbed ‘active, but patient waiting’ in the medical profession[e]. It is a matter of one’s personal experience to keep the balance between waiting and intervening and to recognize when threshold values have been reached.

Many aspects that an architect must consider in the course of the project can wait until the right moment for dealing with them has arrived. I like to document the points of recognition when these moments have arrived in the form of diagnoses and the according remedies as therapies. The advantage of this description method is that very different solution strategies from different points of view that are all valuable on their level of application (technological, process-oriented and that of human interaction) are all summarized in one place. Similar to a doctor of human medicine, a software architect can treat problems solely based on their symptoms or try to find the cause of these symptoms. He or she can work in an exclusively technical way or choose a holistic approach. Some of the smells in this books are also depicted as diagnoses[f].

Sustainable Architecture

The blend of a decent technical solution, two-way adaptations of both architecture and organizational structure, compliance of technology and a process that heeds the developers’ pace, the architect’s attitude, as well as farsighted concepts for handling the software’s entire lifecycle – all these aspects account for a sustainable architecture in my opinion. Such an architecture meets today’s needs without existing at the expense of future releases or the developers.

a. Depicted in [Marquardt & Völter 03].
b. See [Marquardt 99].
c. Derived from the tossing of a category.
d. This is the typical mode of work under ClearCase UCM.
e. Thanks to Dr. Kerstin Marquardt for this verbalization.
f. See [Marquardt 01].

4.4 Example: Lists

Let us take a look at an example to illustrate the discussions in this chapter. This example is similar to a real-life large refactoring in one of our projects.

4.4.1 The Starting Point

Initially, a system contains a class *List* to allow the saving of objects in lists. During development, it turns out that a sorted list is also required. Its behavior is very similar to that of the already existent class *List*. Consequently, we will derive *SortedList* from *List* (see Figure 4-14).

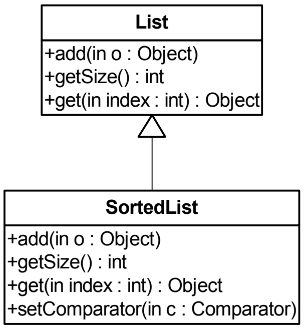

List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

Fig. 4-14
List and SortedList

Naturally, very rarely one will implement one's own list classes. There is no need to, because the required container classes are provided by standard libraries for all popular programming languages. *List* classes are well-suited for our example though, because they are easy to understand. The problems we depicted in the *List* class example also recur in domain-specific classes. We, too had trouble cracking these nuts. We had the following experience:

System development proceeds and the method *insertAt* is introduced to the class *List* (see Figure 4-15).

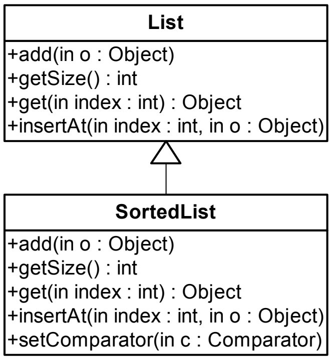

List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)

Fig. 4-15
insertAt in List

In practice, it can take a lot of time until somebody realizes that the design is somewhat crooked: *insertAt* is meaningless in *SortedList*, since the sort sequence specifies its position. As there is no reasonable way of not inheriting methods from superclasses, something must be wrong with the inheritance hierarchy. The inheritance hierarchy though can effortlessly be corrected by introducing another class labeled *UnsortedList*, which contains the method *insertAt* (see Figure 4-16).

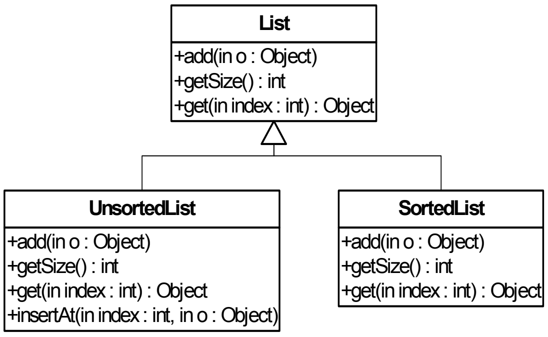


Fig. 4-16
insertAt *in* UnsortedList

Getting there is not quite that simple, because the classes *List* and *SortedList* are already being used all over the system. In one project, we were facing a related problem and chose the following approach.

4.4.2 The First Approach

The class *List* is renamed *UnsortedList* to emphasize the problem in the inheritance hierarchy (see Figure 4-17).

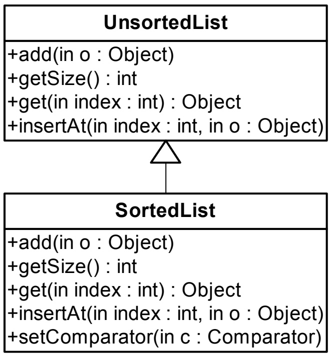

UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)

Fig. 4-17
Renaming List *in* UnsortedList

Now the new class *List* is introduced as the superclass of *UnsortedList*. *List* receives the common methods *add*, *getSize* and *get* (see Figure 4-18).

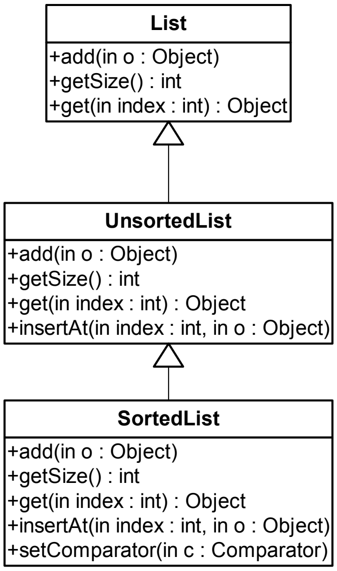

List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)

Fig. 4-18
New Superclass List

Positive is the fact that until now the changes of the inheritance hierarchy took place mainly locally. Only the renaming affected other parts of the system. Since most development environments carry out renamings automatically and adjusts all references, no significant effort on our side was required.

The next step will be to move the class *UnsortedList* within the inheritance hierarchy (see Figure 4-19).

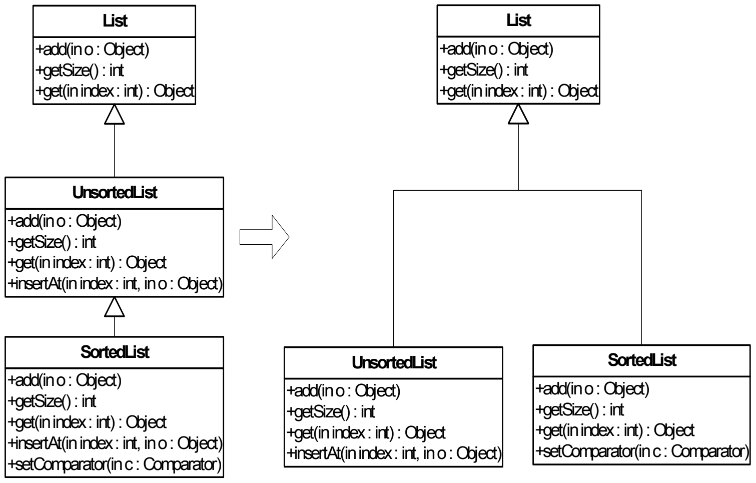

List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)
List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

Fig. 4-19
Correct Insertion of UnsortedList

The Drama Unfolds

The class *UnsortedList* is moved in a split second. And now the drama unfolds: we get pelted with hundreds of error messages. Soon enough, the reason becomes clear: some hundred or thousand times we find method declarations of the following sort:

```
public void doSomething (UnsortedList list)
```

Originally, the parameter type *List* was in this place, but now it's been changed to *UnsortedList* in the course of renaming the classes. Not only unsorted, but also sorted lists were generated in the system. The latter create type problems:

```
SortedList list = new SortedList();
...
doSomething(list); // here the typo occurs
```

So, what can be done now? The initial impulse is certainly to swallow this bitter pill and eradicate the type errors one after another. Unfortunately this means that no compilable system state will be available for quite some time. Depending on the number of occurrences that need to be corrected, it might take unacceptably long for all errors to be eliminated.

Actually, things can even get worse, because faulty inheritance hierarchies are often accompanied by very unpleasant long-term consequences. The inheritance hierarchy will be straightened out in the client code; usually with direct type queries (*instanceof*) and downcasts.

Most likely, our project example will present us code of the following kind:

```
public void doSomething (UnsortedList list) {
  if (list instanceof SortedList) {
    SortedList sl = (SortedList) list;
    sl.setComparator(comp);
 }
 doSomethingElse(list);
}
```

However, we do remember that the parameter type was originally called *List*, and the method implementation did not always look so devastated.

Of course another type error arises here. The compiler knows that *UnsortedList* cannot be casted after *SortedList*. After all, *UnsortedList* is no longer a superclass of *SortedList*.

It is obvious how the method is supposed to look like instead. Fortunately, we did correct the parameter of *doSomethingElse* beforehand.

```
public void doSomething (SortedList sl) {
  sl.setComparator(comp);
  doSomethingElse(sl);
}
```

Now, the whole matter is getting weird: two type errors have disappeared because the initially protested call and the invalid cast were corrected. Instead, ten brand-new type errors have popped up. All of a sudden, new portions of the code show type errors. A closer look reveals code like this one:

```
UnsortedList list = new UnsortedList();
doSomething(list);
```

Wait a second – what is going on here? *doSomething* always used to work with sorted lists, although this could not be deduced from the method declaration. Now, this must be some glitch: due to the *if*-construct in the original method implementation of *doSomething*, the method call was without effect. Accordingly, it seems safe to delete the *doSomething* call in this instance. If we weren't so busy eliminating all those type errors, we could run our tests now. And they would clearly prove that our assumption of a useless method call is wrong. There was a trick hidden in the original implementation of *doSomething*: the method basically executed *doSomethingElse* – for the sorted as well as for the unsorted lists. Only if a method with a sorted listed was called as a parameter, the comparator would produce a certain sort sequence.

Starting All Over

Thus we find ourselves in a major chaos with our refactoring. The only means of escape from this scenario seems to be this one: We'll throw away our entire refactoring work, retrieve the latest working version from the version control system and start over with a different strategy. Admittedly, this is a worse idea than it appears to be at first sight, because we have not only carried out the refactoring in a single branch of the system, but also integrated at least its first steps into the shared repository. This means that all developers have to return to the last status quo prior to refactoring. Thus a couple of man weeks or even months can easily be completely lost. Alternatively, one can try to reverse the commenced refactoring step by step or execute it in a branch, but the latter procedure is not without its drawbacks either. We will come back to the discussion of branches later on.

Perhaps you'll first deem the representation of this refactoring somewhat hypothetical. Who would assemble such a messed-up system without noticing it? We have actually repeatedly seen such systems. Especially during long-term projects unspeakable accumulations of oddities appear to be the rule rather than the exception.

Interruptions of Large Refactorings

We already talked about how large refactorings can be broken down in small increments. Furthermore, large refactorings often cannot be completed within in a short time frame. A development team will need several days or even weeks until the whole large refactoring is finished.

In many projects, the developers do not have the option of dedicating several days or even weeks exclusively to one large refactoring. At the same time, the software system's development is supposed to progress as well. To enable this, developers will put down the large refactoring after a few steps have been carried out and continue with another task (e.g. work on a new feature). Normally, they will resume the refactoring at a later date.

4.4.3 The Second Approach

Let's have another look at our list example. How should we have executed it otherwise to successfully circumvent the cited pitfalls?

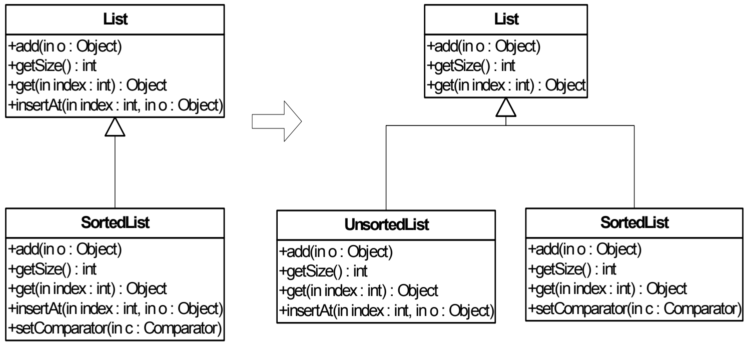
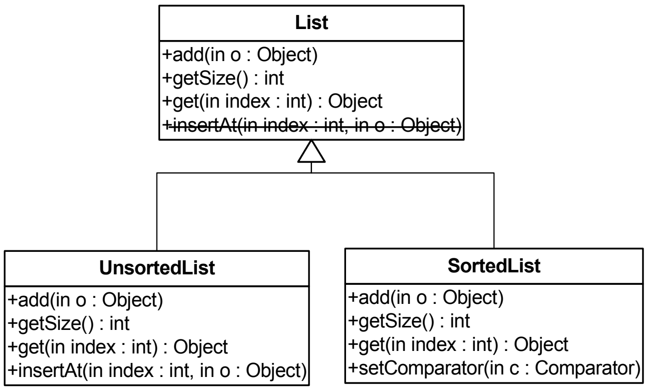

List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
+setComparator(in c : Comparator)
List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

Fig. 4-20
Start and Goal of the Refactoring

In our first attempt, we argued that *List* actually is an unsorted list and renamed the class. Then we extracted a new superclass *List*.

Step 1

Now we'll assume a slightly different perspective and argue that there is nothing wrong with the class *List*. Only the method *insertAt* has no business in this particular class. We set the method to *deprecated*. In our next step, we generate a new subclass of *List* that we label *UnsortedList*. The implementation of the *insertAt* method is copied to *UnsortedList*. We will get:



List
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
~~+insertAt(in index : int, in o : Object)~~
UnsortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+insertAt(in index : int, in o : Object)
SortedList
+add(in o : Object)
+getSize() : int
+get(in index : int) : Object
+setComparator(in c : Comparator)

Fig. 4-21
Step 1: New Subclass
UnsortedList

In our first step, we followed the advice not to move classes inside inheritance hierarchies, but to expand the hierarchy instead by creating new classes. Immediately, we can see one positive effect of our action: We didn't get any compile errors. In their place we merely received a number of *deprecated* warnings. They point to uses of the method *insertAt* under the type *List* in our system.

Step 2

Step by step, we can now analyze the calls of *List.insertAt* and adapt them to *UnsortedList.insertAt*. This process can be quite time-consuming, if the method *insertAt* is called on the type *List*. However, it doesn't matter in which order the *deprecated* warnings are processed.

Let us take a closer look at various code sections in the system that use *insertAt*. These sections reveal to us how these sections of the code can be rearranged.

Replacing *insertAt* with *add*

The simplest option is to substitute the *insertAt* call with the call *add*. This is only feasible though if it doesn't matter in which position the new object is inserted.

```
public void whatever (List list) {
  ...
  list.insertAt(0, myobject);
}
```

... becomes ...

```
public void whatever (List list) {
  ...
  list.add(myobject);
}
```

Adapting Parameter Types

Naturally, we cannot replace all calls of *insertAt* with *add* calls. If we did this, we could delete the method *insertAt* altogether from *List*. Therefore, we will again use a method that calls *insertAt* and gets an object of the type *List* as parameter:

```
public void something (List list) {
  ...
  list.insertAt(i, myobject);
}
```

For this method, the method *insertAt* is called on purpose to allow insertion of the object *myObject* in a certain position in the list. This means that in the future the method will no longer get an object of the *List* type, but one of the type *Unsorted List* instead. We want the method *something* to look like this:

```
public void something (UnsortedList list) {
  ...
  list.insertAt(i, myobject);
}
```

If we change the parameter type of this method in a single step to *UnsortedList* though, it is very likely that we will be confronted with a multitude of compile errors, because the method is still used in its old version in some places in the system. For instance:

```
public void useList () {
  List list = new List();
  ...
  something(list);
}
```

Or:

```
public void useList (List list) {
  ...
  something(list);
}
```

Now, we actually need to adapt all these references to our recently altered method *something* to make the system compilable again. However, such proceeding is against both the principle of taking many small steps rather than a single big one and the principle of adhering to a consistent number of compile errors. So what can we do instead?

Let us recall what our method looks like:

```
public void something (List list) {
  ...
  list.insertAt(i, myobject);
}
```

As an alternative to simply changing the parameter type to *UnsortedList*, we will create a new method with the parameter *UnsortedList* and proceed to copy the old implementation.

```
public void something (List list) {
  ...
  list.insertAt(i, myobject);
}

public void something (UnsortedList list) {
  ...
  list.insertAt(i, myobject);
}
```

Initially, the new method is not going to change the system's behavior, because the methods are bound to the parameters via static types. It is only applied where the new type *UnsortedList* is also already being used, which suits us just fine here.

Now we only have to set the old *something* method to *deprecated* and incrementally adapt its references to the new method.

```
/**
 * @deprecated
 */
public void something (List list) {
  ...
  list.insertAt(i, myobject);
}

public void something (UnsortedList list) {
  ...
  list.insertAt(i, myobject);
}
```

instanceof

In the section about the first refactoring route the following piece of code gave us quite a headache:

```
public void doSomething (List list) {
  if (list instanceof SortedList) {
    SortedList sl = (SortedList) list;
    sl.setComparator(comp);
  }
  doSomethingElse(list);
}
```

The new refactoring route has rendered the same code less critical. As long as the method *doSomethingElse* continues to expect an object of the type *List* as parameter, the method can remain as it is. If *doSomethingElse* is adapted to require a parameter of the type *UnsortedList*, the method *doSomething* must be duplicated as well.

Step 3

Once we have removed all calls of deprecated methods, we can finish the refactoring in a third step. To this end, we must merely remove those methods that are obsolete and marked *deprecated*.

4.5 References

[Bolour 03] Azad Bolour: *Notes on the Eclipse Plug-in Architecture*. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, 2003.

Article about the Eclipse plug-in model.

[Coplien & Schmidt 95] J. O. Coplien, D. C. Schmidt: *Pattern Languages of Program Design*. Addison-Wesley. Reading, Massachusetts, 1995.

Contains many important articles about software architectures and patterns.

[Daum 03] B. Daum: *Java-Entwicklung mit Eclipse* 2. dpunkt.verlag, Heidelberg, 2003.

This book discusses the Eclipse development environment and how one can program one's own plug-ins for Eclipse.

[Eclipse 04] http://www.eclipse.org, 2004.

Website focusing on the open source development environment Eclipse. Here you can download Eclipse itself. You will also find documentations about various aspects of Eclipse.

[Evans 03] Eric Evans: *Domain Driven Design*. Addison-Wesley, 2003.

This excellent book discusses domain driven design. For the context of this chapter, the Pluggable Component Framework is relevant. (See pp. 475).

[FIT] http://fit.c2.com

FIT is a tool for conducting automated acceptance tests (including function tests). The tests are specified via HTML tables (e.g. for tables containing input values and expected output values for specific system functions), which are executed by a test runner. Using fixtures, the test runner binds the application to be tested to the tables containing the tests. In turn, the test results are documented in HTML pages.

[Fitnesse] http://www.fitnesse.org

Fitnesse is based on [FIT]. In addition to FIT, it also offers a Wiki web which allows easier specification and organization of tests.

[Foote & Opdyke 95] B. Foote, W. F. Opdyke: *Lifecycle and Refactoring Patterns That Support Evolution and Reuse*. In: [Coplien & Schmidt 95], pp. 239-257.

Groundwork article about frameworks.

[Fowler 99] Martin Fowler: *Refactoring - Improving the Design of Existing Code*, Addison-Wesley, 1999.

This standard work on refactorings contains a chapter about big refactorings that belong in the category of the large refactorings addressed in this chapter of our book. Fowler describes big refactorings as significant and recurring refactorings. Moreover, four typical larger refactorings are explained, but there is no information whatsoever available how large refactorings should be treated in general.

[Fowler 03] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Contains many important design patterns for the programming of comprehensive business applications, also the plug-in pattern (among others).

[Fowler 04] Martin Fowler: *Inversion of Control Containers and the Dependency Injection pattern*. http://martinfowler.com/articles/injection.html, 2004.

Here, Fowler focuses on the inversion of control containers and discusses several approaches to the generation of plug-ins.

[Freese 03] Tammo Freese: *Inline Method Considered Helpful: An Approach to Interface Evolution*, in: Michele Marchesi, Giancarlo Succi (eds.) *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2004*, Genova, Italy, Springer LNCS 2675, 2003.

In this article, Freese depicts how the Inline Method Refactoring can be used to enable a stepwise evolution of interfaces. In our book, we are using a simplified variety of that technique to resolve deprecated methods.

[Gamma et al. 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

The design pattern bible. Also contains the facade pattern.

[Hoffman 03] Michael A. Hoffman: *Automated Impact Analysis of Object-Oriented Software Systems*, in OOPSLA 2003 Companion, ACM press, 2003.

In this extension of his abstract, Hoffman writes about a tool that allows the conduction of several types of impact analyses. Particularly interesting is the predictive impact analysis option to anticipate the impact of changes.

[JMigrator 04] *http://sourceforge.net/projects/jmigrator*, 2004.

JMigrator is an open source project that provides support for modifications to subsystem APIs. Parts of its functionality can be utilized for large refactorings, e.g. for the detection of polymorphic assignments. JMigrator is realized as an Eclipse plug-in. At press time of this book, JMigrator is still in an early stage of development.

[Lippert 04] Martin Lippert: *Towards a Proper Integration of Large Refactorings in Agile Software Development*. In Proceedings of XP 2004 International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer LNCS, 2004.

This XP-2004 conference contribution elaborates on the problems of large refactorings in an agile development process. It focuses on the organizational aspects and obstacles and suggests the use of refactoring plans.

[Marquardt 99] Klaus Marquardt: *Patterns for Plug-Ins*. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999). Paul Dyson, Martine Devos (eds.). Universitäts-Verlag Konstanz, 2001.

This article describes typical patterns of a plug-in architecture and offers a number of patterns. In addition to patterns on the architectural level, it contains patterns for organizational and process-related issues and scrutinizes some design decisions.

[Marquardt 01] Klaus Marquardt: *Dependency Structures. Architectural Diagnoses and Therapies*. In: Proceedings of the Sixth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 2001). Andreas Rüping, Jutta Eckstein, Christa Schwanninger (eds.). Universitäts-Verlag Konstanz, 2002.

In this article, a series of bad smells is portrayed in the shape of diagnoses and therapies. The collection of diagnoses primarily refers to architectural aspects and offers a series of possible therapies for each smell that will help to cure it.

[Marquardt & Völter 03] Klaus Marquardt, Markus Völter: *Plug-Ins - Applikationsspezifische Erweiterungen*. In: JavaSpektrum 2/2003. Available online at:
http://www.sigs-datacom.de/sd/publications/pub_article_show.htm?&AID=1117&TABLE=sd_article

The functional concepts of plug-in architectures are introduced in this source and compared to those of other component architectures. The main topic is the impact of a plug-in architecture and related contract issues as well as a decision guidance, if this architecture type is useful for a specific project or not.

[Martin 97] Robert C. Martin: *Stability.* C++ Report, 1997

Even though this contribution is several years old, its content has neither collected dust, nor is it C++-specific. In this article, DIP (Dependency Inversion Principle) is also described.

[Schulz von Thun 98] Friedemann Schulz von Thun: *Miteinander reden*. Vol. 2. Rowohlt Taschenbuch, 1998.

Volume Two of this bestselling series. How to develop a systemic view of mutual restraints and vicious circles in communication. Development and Value Diagram, personal ways of communication, approaches to communication improvement.

[Schulz von Thun 98] Friedemann Schulz von Thun: *Miteinander reden.* Vol. 3. Rowohlt Taschenbuch, 1998.

Volume Three of this bestselling series. Analogies between teams consisting of several persons and the inner team, the various voices within one person. Work methods of these teams: side by side (without contact), disordered (without structure), against each other (restraining), with one another (fostering). How to remove blockades, integration of all members, team development, coherent presentation of the (inner) team to the external world.

5 Refactoring of Relational Databases

In application development, mostly relational databases are employed nowadays. Other than object-oriented programming languages, relational databases hardly offer any options for building modules. Therefore, there is no way of locally limiting the effect of changes to a single module.

Changes to relational database schemas (e.g. removal of a foreign key from a table) will often affect expansive areas of the schemas and thus create a need for comprehensive adaptations of the program.

This chapter addresses what modifications of relational database schemas occur, and how these can effectively be embedded in large refactorings.

Modifications of relational database schemas and the required work in their wake (program adaptations, data migration etc.) were much discussed in the context of agile methods. This chapter will survey several of the discussion results, which means that the authors of the concepts quoted here should be honored rather than us. We, the authors of this book, merely assembled the information.

5.1 Differences between Databases and OO Programming Languages

Before we get started, it is helpful to take a closer look at the differences between relational databases and object-oriented programming languages:

- The focus of relational databases is on the definition of data structures. Encapsulation through methods or the like is not allowed. In consequence, data access cannot be sensibly restricted.
- Tables are connected via foreign key relations. These links too cannot be encapsulated.

- Data in databases is persistent and outlasts a program run. If the database schema is altered, the data must migrate.
- If more than one installation of the system exists (e.g. at different customers), there will also be different databases. Should the database schema be changed, the respective change must be made for each installation and the data of each schema must migrate.
- Different users can access data simultaneously, while conceptually each single user has his or her own copy of the program.
- Classes can inherit from each other; tables and data can't.
- Source code can be changed locally from the developers' terminals and tested prior to re-integration in the shared code repository. Conflicts can be recognized and eradicated with the aid of powerful tools. In most projects, the database is run centrally for all developers.
- Source code can be managed with the support of version control systems and administrated in variants. Database structures and data can only be versioned with significantly greater effort.
- Accessing data in the database takes a multiple of the time required for accessing objects in the RAM.
- The data structures in relational databases are shallow, whereas they are deep and interlaced in OO systems.

5.2 Problems in the Interaction of Programs and Database

The interaction of programs and database creates additional problems:

- Program and database are often not coupled typesafe (as it happens to be the case with JDBC, for example). The compiler has no means of assessing if program and database are structurally compatible. Suitable mapper classes or persistence layers will take the problem elsewhere without solving it. Typesafety will be lost inside the mapper classes or the persistency layer, not already outside.
- Databases will 'hide' objects when in one place of the program objects are written to the database and then read out somewhere else. Thus objects can be exchanged between parts of the program without this process becoming visible at the program's interfaces.
- Frequently, a 1:1 relation between classes and tables is assumed, which is often not correct for data-intensive applications. Specifically for reading in objects from the database, several tables must be joined, or certain views must be applied for performance reasons. As a result, there is no simple way of determining which

classes must be adapted in the course of database modifications. Conversely, it is not always clear how changes of classes will affect queries and views.

- There is the odd case where a mapping between types in the database and the primitive data types of the used programming language will cause difficulties. For instance, the granularity of time stamps (TIMESTAMP), floating point numbers of varying precision or strings of various character sets can deviate (milliseconds versus nanoseconds).
- In object-oriented systems, containment relations are modeled based on the container (i.e. an account will know its balance). In relational databases, 1:N relations are modeled precisely vice versa (due to foreign key relations balances know to which account they belong). This means that there is no predefined course of action for a refactoring.

Thus we have to consider three major areas for refactoring:

1. Refactoring of the database schema/the data model.
2. Migration of data between different versions of the database schema.
3. Refactoring of the database access code.

5.3 Refactoring of Relational Database Schemas

Development and Production Database

In practice, a number of database schemas will exist parallel. There are at least *two* variations: one for the developers (the *development database*), and one for users (the *production database*).

Thus developers can try out modifications of a database without affecting the system's users. Only when the changes to the database have been thoroughly tested and adapted to the system on which the database is built, the program and the new database schema will be made available to users with the next release.

One Database for Each Developer

Moreover, each developer should have his or her own database instance to be able to test changes to the database in isolation from the rest of the team. The existence of several database instances makes the migration of data between various types of database schemas a pivotal topic. We are going to discuss this topic in the following section.

Refactoring a Database Schema

In many refactorings a central principle of a stepwise evolution of programs as well as of data structures is recurring: old structures are not immediately replaced with new ones. Instead, old and new structures will exist side by side for some time. The old structure will be marked *deprecated* to keep new parts of the program from accessing it. Then the existing programs are modified; step by step they will be adapted from the old structure to the new one. Once this has been accomplished, the old data structure will finally be deleted. Figure 5-1 shows the evolution of the table *Customer.* Initially, first and last name were stored together in one field labeled *Name*. The two pieces of information shall now be submitted to the fields *First Name* and *Last Name*. To this end, both fields are added to the *Name* field and the latter marked. After all programs have been adapted to use *First Name* and *Last Name*, the *Name* field is deleted.

In its intermediate state, the table *Customer* contains redundant information (*Name*). Either the application can ensure that the data will be consistent, or the problem is solved with the support of suitable triggers.

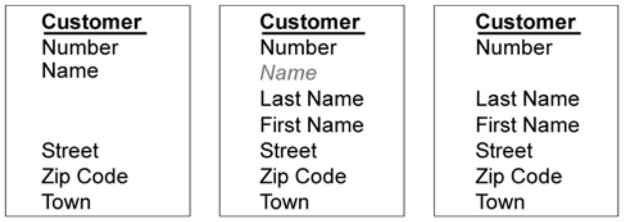


Fig. 5-1
Coexistence of Old and New Fields in a Table

In Java, program elements can be marked obsolete with the *deprecated* tag. In other programming languages as well as in the database field, the search for an equivalent of this tag will be fruitless in most cases.

In the relational database field, columns, tables, views or even entire schemas will be marked *deprecated*, depending on the refactoring that is applied. How these elements are marked as being obsolete is primarily determined by how the database is accessed. If, for example, an OR mapping tool that generates Java access classes is used (such as Apache's *Torque*), the generated classes or single methods can simply be marked *deprecated* – on the one condition that no other system directly accesses the database. Figure 5-2 shows this proceeding. The application does only access the database via mapper classes. Therefore, it suffices to mark the access methods for the field *Name* as *deprecated* in the second version (crossed-out methods).

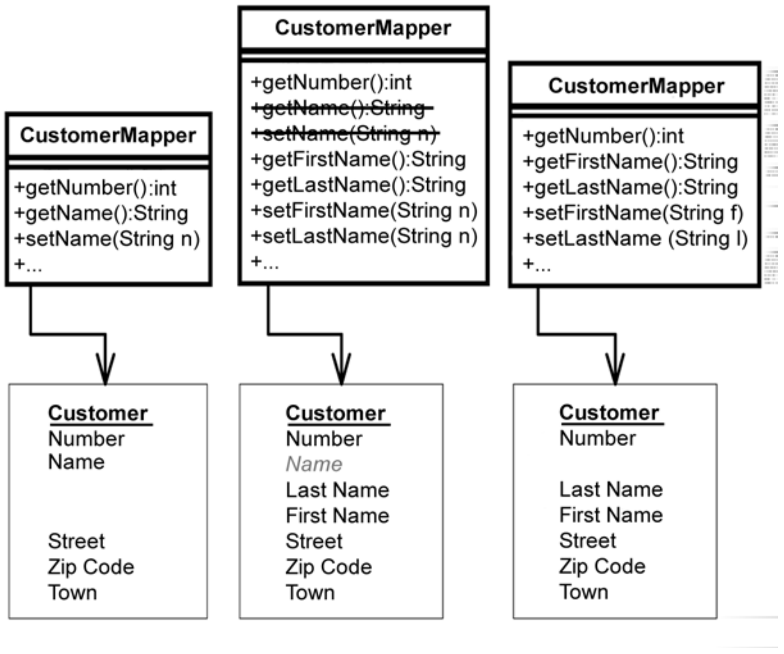


Fig. 5-2
A Database Access Layer Makes the Marking of Deprecated Elements in Data Structures Easier

If access does not take place strictly channeled via specific access classes, one usually will have to make do with conventions. One can either maintain a list of all obsolete elements or add comments to the elements that state that those elements are deprecated. Of course it is important that all developers know the convention agreed upon and observe it.

5.3.1 Database Refactorings

On his website, Ambler (see [Ambler 2003b]) has collected frequently required refactorings for relational database schemas. The refactorings depicted there are a useful reference source.

Ambler's refactorings aim at improvements of database schemas. Therefore, the addition of a column alone does not constitute a refactoring. The added column by itself will not improve the database schema.

Database refactorings fall into various categories: refactorings that will either improve the data quality, structure, performance, referential data integrity, or the database architecture.

The descriptions of refactorings refer to the database structure. The adaptation of programs or the migration of existing data is only mentioned as a side note.

The *deprecated* concept is also applied to database refactorings. Since tables and columns cannot be marked as being deprecated in relational databases, another way for communicating what is deprecated must be found (see previous page).

5.4 Migration of Data between Different Versions of a Database Schema

As soon as the system has been released for users, a refactoring of the database schema will no longer be sufficient. For instance, if a column is moved from one table into another, this will be realized with SQL. The column is deleted in the source table and newly generated in the target table. As soon as the restructured system is released, we must face the problem of migrating the data from the old to the new version of the database schema. We will also have to observe how the selected migration strategy impacts the database refactoring.

5.4.1 Versioning Database Schemas

Therefore, we need to migrate the existing data to the new schema after the database schema has been changed. Of course the old and the new schema need to exist side by side during migration. Only after data migration has been completed, the old database schema can be deleted.

The coexistence of two schema versions can be realized in different ways. One option is to define a schema exclusively for the respective version. The version number is then incorporated in the schema's name, i.e. schema *V1*, schema *V2* etc.

Alternatively, the version number can also be incorporated in the table name, e.g. *Customer V1, Product V2*, but this will also have consequences for the refactoring of the database schema. After all, foreign key relations, constraints, and triggers all contain references to the tables' names. Once a new version of a schema has been created, all these references must be adjusted.

For this purpose, we assume that for each schema version a corresponding database schema is created.

The schema name enables the application to identify in which version the schema exists. This is an important prerequisite for the data migration of software products that are used by numerous customers. In such a scenario, one cannot take for granted that each customer uses the most recent version of the system. Thus it should be possible to migrate data from any older version to the newest one. To achieve this, one has to discern in which schema version the data originally exists.

5.4.2 Connecting Migration Steps

As mentioned before, when developing software products for a greater number of customers, it is not safe to expect that each customer uses the most recent version of a system software. Customers are likely to skip one or another version of the system.

Each migration transfers a database schema's data into the new version of that schema. This new version will be the starting point of the next migration.

Consequently, the newly installed system version must determine in which schema version the data is available and then proceed to carry out all required migrations until the process is complete for the youngest version. This presupposes that each installation must contain all migration programs that were ever created. For very expansive systems which have been in use for a long time, this can pose a problem, because very old migration programs might not work, for example, with the current version of the operating system. In such cases, the system must be broken down into generations and only deliver those migration programs as part of the installation which belong to the youngest generation.

The customer cannot expect that a migration from an older generation to the most recent one can be executed in a single step. If necessary, several migration steps must be carried out.

Figure 5-3 illustrates the principle of connected migration. If migration takes place from one version to the next, only one of the migration programs *MigA, MigB or MigC* will be carried out. Should migration happen from *V2* to *V4*, the application system will first execute the migration program *MigB* and then *MigC*.

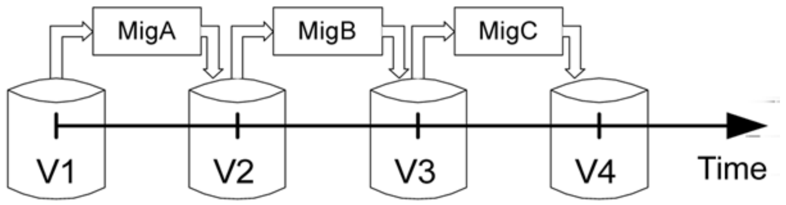


Fig. 5-3
Connecting Migration Steps

5.4.3 Migration of Very Large Data Amounts

Problem: Very Large Amounts of Data

When dealing with very large amounts of data, a single migration program can cause time problems: the migration of one billion data sets is hardly a feat that will be casually accomplished. Shutting down, for example, the main system of a bank house or an insurance company for 24 hours in the middle of the week to migrate their data is out of the question.

Organizational Incorporation

However, this problem can be approached either from an organizational or a technical perspective. For the organizational solution, one has to precisely schedule modifications of the database system and conduct the migration when there is enough time. The Christmas and Easter holidays are ideal for this purpose.

Incremental Migration

If the overall technical conditions allow such a proceeding, the migration can be executed incrementally. The data will be migrated only when the system needs it. The migration period is thus prolonged, but the system's operation will not be interrupted. In a scenario with strict 7x24 runtime requirements, an incremental migration is often the only feasible solution.

Parallel Versions of Database Schemas

One prerequisite for incremental migration is that the application system is able to handle various versions of the database schema simultaneously. When accessing the database, the system must know in which schema to find the required data. Altered data will always be committed back to the new database schema though and then deleted from the old one. Once the old schema does no longer contain any data, it can be deleted.

Many Parallel Versions of Database Schemas

If the incremental migration takes longer than one release cycle, more than two variations of the database schema will exist at the same time.

An elegant way of keeping data simultaneously in different database schemas is to save the objects in BLOBs as well as fieldwise saving (see 5.9).

5.4.4 Data Migration Techniques

ETL tools can greatly simplify data migration. ETL stands for *Extract*, *Transform*, *Load*. ETL tools support the extraction of data from a data source, transformation of that data, plus loading it to another data storage. ETL tools are generally used to exchange data between applications which are not integrated. This makes them important tools in the EAI (*Enterprise Application Integration*) field.

Using ETL tools for data migration between different versions of database schemas was originally only a by-product – as a matter of fact, ETL tools have capabilities that go far beyond such application. Unfortunately, herein also lies the main disadvantage in utilizing them for data migration: they are very expensive. Buying them solely to deal with typical migration tasks is often not worth the money.

Fortunately, a less costly ETL tool is available for each relational database: SQL. With the help of SQL, data can easily be extracted (SELECT) and reloaded into the database (INSERT, UPDATE). However, SQL does not offer any direct support for transformation tasks, but often recoding tables or stored procedures will come in handy here. In such recoding tables, source and target values are specified for single fields. A problem-free migration of field contents is enabled by the *Insert-Select* command. This course of action is recommended if one decides to change the display of enumeration types. If a field was, for instance, coded with the character M for 'male' and 'F' for 'female' and is now supposed to be displayed with the digits 0 for 'male' and 1 for 'female,' we have an ideal area of application for a recoding table.

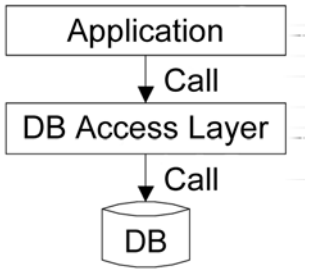
If more complicated data migrations are necessary, programs that will carry out the data transfer must be written. Today, many databases allow the running of Java programs directly in the database. This can be beneficial for data migration, because the data no longer must be transferred from the database server to the machine handling the migration over the network. Thus migration can be noticeably sped up.

5.5 Refactoring Database Access Codes

One of the oldest demands in software development is the call for encapsulating database access in a database access layer. A database access layer provides the option of exchanging the persistence medium. The developers only have to adapt the database access layer instead of rewriting the whole application system.

With agile methods and large refactorings, the demand for a database access layer is supported by another argument: The effects of changes to the database schema will be limited to the database access layer. This is the only means of keeping the subsequently needed efforts sufficiently small. Figure 5-4 shows the schematic architecture of an application with a database access layer.

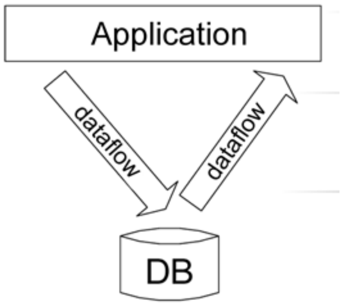
Fig. 5-4
Database Access Layer



5.5.1 Synchronized Changing of the Database Schema and Database Access Code

Generally speaking, application systems store data in databases to read them out later on (see Figure 5-5).

Fig. 5-5
Dataflow of the Database Connection



Redundant Structures of Business Objects

At least when integrating relational databases, structures that are in part redundant will be created: the structures of business objects in the application as well as in the database bear a strong resemblance to each other. This is why usually both application *and* database must be adapted in the course of refactorings of such business object structures.

This means there are four potential places that must be modified during refactoring:

1. The portions of the database schema that are relevant for the altered business object structure;
2. The classes that define the modified business object structure in the application;
3. The database read-in operation for the business object;
4. The database write operation for the business object.

Centralizing Read/Write Access

In principle, the read and write operations addressing the business objects can be arbitrarily distributed over the whole system. A well-designed system will at least ensure that there is only one place in the system where a business object is stored in the database. During read-in from the database this is unfortunately not always possible, because for performance reasons entire business object graphs are at once loaded in the *select* instructions via joins.

The most common case for refactorings of business object structures is execution of the following procedure:

1. Changing the table in the database exclusively in increments:
   - New fields are added;
   - Fields to be deleted remain in the code and are marked *deprecated*;
   - Modified fields are duplicated; the old fields are marked *deprecated*.
2. New fields in the database are assigned appropriate default values.
3. Business objects classes are changed in such a manner that the new data fields from the table can be stored in the business object and values for the *deprecated* fields can still be delivered; if necessary, the fields of the business object class must be set to *deprecated*.
4. All database write operations must be extended by the option to fill new fields.
5. All database read operations must be adapted to allow the read-in of these new fields; all read access to *deprecated* fields in the table must be eliminated.
6. All write access must be removed from *deprecated* fields.
7. *Deprecated* fields must be deleted from the table.
8. *Deprecated* fields must be deleted from the business object class.

This procedure does not constitute a generic, universal solution. It should specifically be amended for the respective refactoring.

A Simple Example

Let us now assume that we wish add a new field for country codes to the class *Customer* (so far, we only had German customers; now we will deal with customers from all over the world). To this end, we will first introduce the new field into the class *Customer.* Here it is assigned the default value 'G' for Germany. As of yet, the field will not be saved and loaded.

In the next step, the new field is added to the database schema, and all existing data sets are assigned the default value 'G' for the new field. Now the loading of customers will be adapted, followed by saving. Finally the new field is made visible in the GUI.

Here is short version of each the single step:

1. Add a new field to the class and assign the default value.
2. Introduce a new field to the database schema and assign the default value.
3. Enable loading of the new field.
4. Enable saving of the new field.
5. Make the new field visible in the GUI.

After each single step, the system is in a consistent state.[1] The desired effect will be achieved with the last step. Until this step has been taken, the system can only handle customers from Germany.

A More Complex Example

Not always are things so simple: Let us assume that we wish to store the country code no longer as a character code, but as a number instead. To realize this, the country code field must be changed from the type *String* to *Integer* in the database as well as in the program code. At the same time, the persistent data must be recoded.

The following refactoring steps will render the desired result:

1. Add a new field to the class and assign the default value.
2. Introduce a new field to the database schema and assign the default value. Add the new field to saving.
3. Add the new field to loading.
4. Switch the GUI and all other access to the old field to the new field.
5. Remove the old field from loading.
6. Remove the old field from saving.
7. Delete the old field from the class.
8. Delete the old field from the database schema.

Concerning points 1 and 2: Here a static default value cannot be employed because the default value depends on the existing country code. A recoding algorithm is required to calculate the numeric country codes for the existing *string* country codes. For this purpose, the recoding algorithm will probably have to access a recoding table in the database.

A closer look at the single steps makes it clear that the modifications for loading and saving cannot be finalized in one step. As a matter of fact, both parts of the system must be adapted several times.

Of course making the final adaptation in one step is extremely seductive. If loading of the field in step 3 is removed at once, the sys-

1. Steps 1 and 2 as well as 3 and 4 can also be executed in reverse order.

tem initially appears to be in a consistent state (no compile errors will be reported, and tests working with the new field will also run error-free). Admittedly, after loading, the field in the objects would show a default value that does not match the field's new value. This constellation can create all kinds of problems in the rest of the program code.

5.6 Roles in a Project

In projects that use agile methods, the previously described database refactorings and the procedures following in their wake (data migration) are the rule, not the exception. The whole procedure must be organized without impairing its progress.

Foremost this means that the understanding of the DBA's (database administrator's) role undergoes change. He or she will not personally make every single change to the database. This would encumber the developers' work and overburden them with the sheer number of modification requests.

Instead, the DBA has to accept the role of the person who supports the developers with changes to the databases. After all, he or she usually has a more detailed knowledge of it. As a side-effect, the DBA can also keep track which modifications to the database are made and can intervene, if – in his or her opinion – development takes a wrong turn.

This changed understanding of the DBA's responsibilities will last but not least be reflected in the allocation of rights. Developers in agile projects need more database rights. At least for their local database and the shared development database they must have the right to make changes to the database schema. Modifications of other schemas can, as before, be executed by the DBA, who will also function as a quality-ensuring checkpoint.

5.7 Tools

Graphical administration tools are available for most database types. Normally, they also allow changes of database structures. However, these tools cannot be considered refactoring tools for databases. On one hand, the effects of modifications are simply ignored, on the other hand they don't offer any mechanisms to take back changes or to version them etc.

However, the majority of projects that apply refactorings will not execute changes of the database schema with the aid of such tools. Should no other tools be at the developers' disposal, SQL scripts will serve instead to realize modifications of the database schema. The scripts are versioned in the version control system. The gradual execution of these scripts enables migration of an database schema existing in any version to any successive version. Thus, writing additional migration programs for the migration of production databases will often be unnecessary.

5.7.1 OR-Mapping

The mapping of objects to relational databases is supported by a variety of commercial and open source tools. For Java, often *Torque*, *Castor*, or *Hibernate* from the open source field as well as the commercially available *TopLink* are used. With the introduction of *JDO* (Java Data Objects), a standardized programming interface for such an OR mapper in a Java environment has now also been defined. For the future it is expected that the existing OR mappers will be able to support JDO.

Most OR mappers generate SQL scripts and source code for database access from a description of persistent data structures. Rarely the application developer will directly have to deal with SQL.

The source code generated in such a way constitutes a good basis for the database access layer and significantly improves typesafety of the database access. Access is not always 100% typesafe, because direct changes of the generated classes or the database schema will again result in a loss of typesafety. In addition, the formulation of queries can lead to type errors.

Overall, the generated source code brings about a clear enhancement of the situation compared to direct database access. In this way, code-generating OR mappers support large refactorings in a minimalist way: the effects of changes to persistent data structures will become visible as soon as the OR mapping source code is newly generated. The affected parts of the application will now display compile errors. We are yet far away from having achieved a refactoring in small steps, let alone an automation. The application developer must still ponder which refactoring steps he or she wishes to take.

Equally, OR mappers are of little help for the migration of existing data. Here again, the application developers must decide what is to be done.

5.7.2 ETL Tools

ETL tools (see also chapter 5.4.4) read out data from a data source, transform this data and load the results into another data source. ETL tools are, for example, used in the EAI field (*Enterprise Application Integration*) to synchronize the data of different applications.

A welcome side-effect is the usefulness of ETL tools for data migration between different versions of a database schema. In comparison with the writing of individual migration programs, the application of ETL tools saves a lot of work. However, it should not go unmentioned that the licensing costs for commercial ETL tools range in the five-digit dollar zone. Many projects will discard such an investment that will 'merely' speed up development in a fringe area right away.

Should ETL tool licences have been purchased for other reasons though, their application is an extra benefit.

5.7.3 Scripting

If no ETL tool is at the developers' disposal, most of the data migration for refactoring purposes should be done with scripts. After all, the migration programs/scripts will be only used once for data migration and then never again. Thus their maintenance requirements are not as high as those for the actual production system.

5.8 Tips

- Develop a database access layer that hides the database structures completely from the application. As a result, modifications of the database schema will be limited to the database and the database access layer.
- Define all table and column names as constants and use the constants for database access. Typical OR mappers will generate the constants from the description of persistent data structures. If no OR mapper is at your disposal, write your own program to generate the constants from the database.
- Adhere to the naming conventions for primary and foreign keys to enable easier detection of dependencies in the database schema.
- Use different database instances or at least different database schemas for *staging*: Unit test DB per developer, DB for integration testing, DB for acceptance testing, DB for production.
- Do not use the database manufacturer's tool to change the database schema. Write SQL scripts instead for changing the schema. Write the scripts in such a way that existing data can be migrated. Developers can use these scripts to adopt modifications of the local database instance to the development database, but also to migrate the database schema and data from the production database.

- Apply the following strategy in regard to the database: make sure that each database instance contains both an old and a new version of the database schema. It is the only way of migrating the data with minimal effort when switching the schema.
- If you are using OR mappers, version the definition of persistent data structures in the version control system.
- If you are not using OR mappers, version the SQL scripts for creating and changing the database schemas in the version control system. Alternatively, in many cases you can let the database itself generate the description of its structure (SQL Create Statements) and then version it.
- Ensure that your source code stays independent from the concrete data that is stored (i.e. it should not presuppose the existence, for instance, of customer no. 999). Should this not be feasible, proceed to make the dependency explicit, e.g. let the program verify the existence of the required data at program start.
- Unit tests should see to the existence of the required data themselves. They should either generate the data directly from the program code or load it to the database, using a script. Then the script must be versioned together with the source code.
- More complex data models do not always permit that tests generate their required own data. Thus the tests will depend on data in the database. In such cases, make sure that the tests will exclusively depend on the data in the unit test database instance.
- When the system expands, the performance of database-dependent unit tests can become problematic. In these cases, design patterns such as *ObjectMother* or *MockTypes* are useful. Where appropriate, an in-memory database can be utilized for testing.

5.9 Atypical Data Models

If an application system requires extremely flexible data structures (for example, because the users shall be able to change the data structures themselves), developers will often work with data modeling on a meta level. They will create a data model that allows saving any type of data structure. In principle, this can be realized in two ways: *Saving of BLOBs* and *fieldwise saving*.

Such data models enable a flexible handling of constantly changing data structures and simplify the refactoring process.

5.9.1 Saving of BLOBs

When saving BLOBs (*Binary Large Objects*), information about the actual data structures is only present in the application system. Only the application knows how these BLOBs are structured internally. Often such systems work with two tables: a data table and a search table. The data table possesses only two columns: ID and object. In the ID column the unique ID of the saved object can be found, whereas the object can be found in the object column as a BLOB. The ID column is the primary key.

Searching for BLOBs in the Database

All criteria which are potentially searched for are listed in the search table. The search table has two columns too: the ID as well as the search criterion. The ID column is a foreign key to the data table ID. For each search criterion which can be looked for with an object, a data set is stored in the search table. The primary key consists of ID and search criterion. Figure 5-6 gives an overview of the two tables.

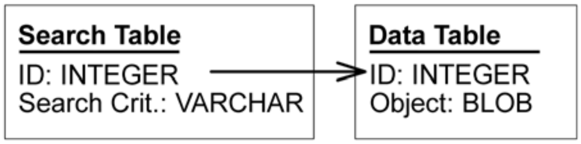


Fig. 5-6
Saving Objects as BLOBs

References between BLOBs

When saving in BLOBs is desired, developers must decide how uses relations between objects can be circumvented. Here, a distinction is made between *containment relations* and *references*. Objects that are contained in other objects will be saved and also read as a whole in a BLOB with the parent object.

References in class definitions are specially marked (e.g. through saving only the referenced object's ID in the field instead of applying a uses relation). In this way, at first only the original object will be loaded. The referenced objects will either be loaded directly afterwards or when they are actually required.

The advantages of this type of modeling are:

- Changes of the data structure require only modifications of classes in the application system, but not of the database schema itself. Therefore, no simultaneously active database schema variations are needed.
- A refactoring of data structures is limited to changes of the program code and the migration of existing data.
- Data can be migrated stepwise from an old to a new data structure during loading. The old and new *BLOB-Mapper*[2]

versions are required for migration, but only *one* database schema.
- OR mapping is simplified altogether.
- Objects with a complex structure can be read and write very performantly.
- It is easy to realize flexible data structures that allow adjustments by the user.

Disadvantages

And these are the disadvantages:

- The stored data cannot be used without the application system.
- As a rule, the stored data cannot be used by other systems, i.e. such that were written in other programming languages. The database cannot be utilized as an integration medium for different systems.
- Report and list generators based on the database cannot be applied.
- Data can only be analyzed as far as this function has been programmed into the application.
- Inconsistent data cannot be repaired manually via the provided database mechanisms.
- The saved objects constitute the smallest locking level.
- Where high transaction rates are present, search and data tables can turn into a bottleneck for locking.
- The number of data sets, especially in the search table, can be very trying regarding the database's performance capacity when a large amount of data is involved.

5.9.2 Fieldwise Saving

For fieldwise saving, like for saving with BLOBs, basically two tables exist: the search table and the data table. However, the data table contains no BLOBS, but a data set for each stored field instead, so that for each field of each saved object a data set is created. The number of data sets in this table is easy to calculate: Number of objects * average number of fields for each object. To make sure that objects can be reconstructed from the data table, at least columns for the object ID and the field name are needed besides the ID and field value columns.

2. If, for instance, Java serialization is used directly, the classes must be able to load objects of earlier class versions.

Figure 5-7 shows both tables. The ID in the data table constitutes the unique primary key. This key is not imperative though. It is also possible to define the primary key as a composite of object ID and field name.

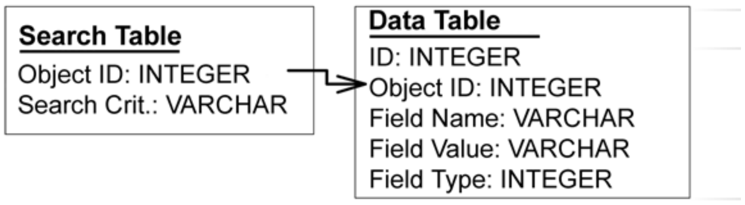


Fig. 5-7
Fieldwise Saving

Advantages

The advantages of this type of modeling are:

- Changes of the data structure require only modifications of classes in the application system, but not of the database schema itself. Therefore, no simultaneously active variations of the database schema are needed.
- A refactoring of data structures is limited to changes of the program code and the migration of existing data.
- Data can be migrated stepwise from an old to a new data structure during loading.
- OR mapping is simplified altogether.
- It is easy to realize flexible data structures that allow adjustments by the user.
- Advantages as compared to saving in BLOBs:
  - The data can be used by other systems and tools.
  - Inconsistent data can be manually repaired.
  - Ad hoc analyses can be carried out directly with SQL.

Disadvantages

In contrast, these are the disadvantages:

- Where high transaction rates are present, search and data tables can turn into a bottleneck for locking.
- The number of data sets, especially in the search table, can be very trying regarding the database's performance capacity when a large amount of data is involved.
- Access to the database is slowed down, because now not only one data set per object, but many data sets must be processed.
- In comparison, the ratio between the share of user data and the overhead is relatively bad. For many values less bytes would suffice for storage (e.g. for Integer fields), but storage capacity

is always reserved for String saving. Moreover, for each persistent field its own key information is stored. Sometimes this key information requires more storage than the actual, saved data[3].

5.10 An Example

This section will use a more comprehensive example to further elaborate on the previously introduced principles for refactoring with databases. We will use a time recording system for IT consultants, which lets all consultants access a web interface to enter their actual work hours. This input serves as the basis for calculating the consultants' salaries as well as for billing their customers.[4]

5.10.1 Our Starting Point

Subsystems in a Time Recording Example

The subsystems are depicted in Figure 5-8. The consultants access the systems via the subsystem *Web*. The accounting department uses the subsystem *Report* to generate the necessary print lists and analyses.

Subsystem Web

The subsystem *Web* works with the subsystem *Business Objects*, which provides concepts such as *Employees* and *Time Entries*. These business objects are saved in the database and reconstructed from the database with the subsystem *Administration*. To this end, a purchased subsystem *DB* (the driver for accessing the concrete database; for Java this will usually be JDBC) is utilized.

Subsystem Report

The subsystem *Report* employs the subsystem *Analysis* in order to carry out all necessary analyses for the print lists (e.g. all hours for each employee for one project). Of course, the subsystem *Analysis* uses the subsystems *Business Objects* and *Administration* to access the persistent business objects. The print lists are created with the aid of a commercially available report tool (subsystem *Report Tool*).

3. This problem can be solved by placing field name and field type in a table of their own (normalized variation).
4. This example is also used in the chapter about API refactorings. We decided to reprint it here in its entirety, so that both chapters can be read independently of each other.

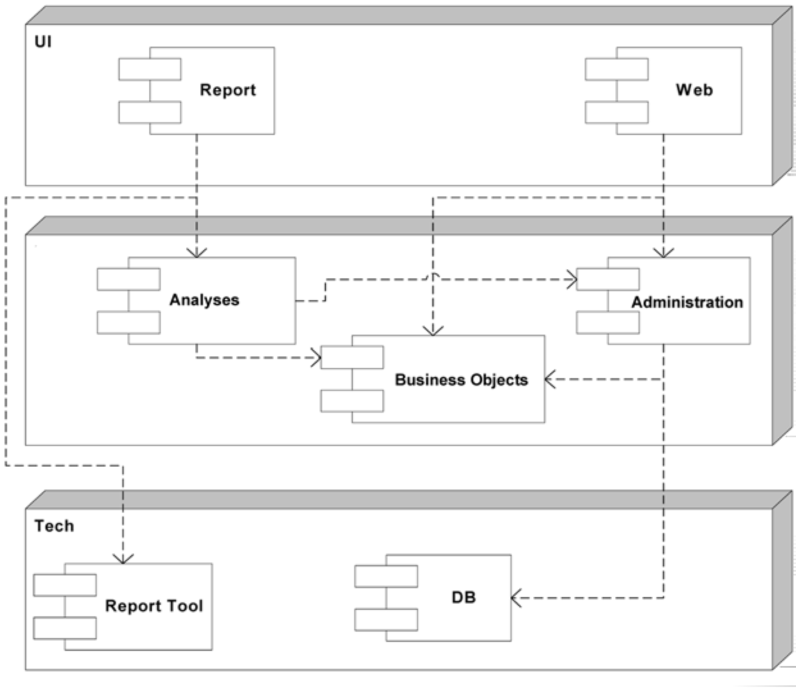


Fig. 5-8
Subsystems of the Time Recording Example

The subsystems are arranged in three non-strict layers: user interface (UI), domain model and technology.

Essentially, the time recording system is based on the business objects from Figure 5-9: *Time Entries* has a vital position here: besides date, start and end time, *Time Entries* also displays references to *Project*, *Activity* of the project, as well as to *Employees*.

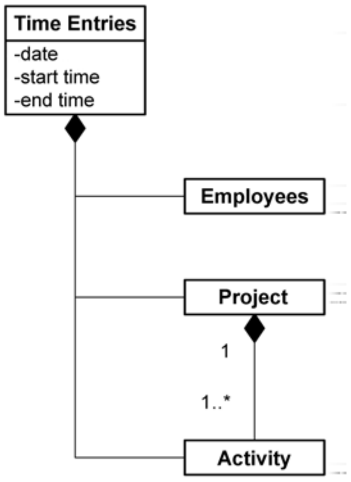
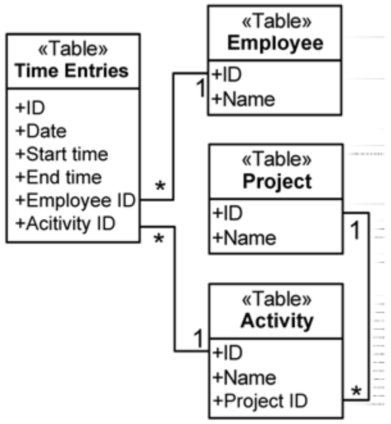


Fig. 5-9
Business Objects

Figure 5-10 shows a simple data model for storing business objects.

Fig. 5-10
Data Model for Business Objects



5.10.2 Motives for a Refactoring

The modeling of the subsystem *Business Objects* strongly influences the API of the subsystem *Administration* and thus also the interaction between *Analysis* and *Administration*.

Basically we have to implement the respective low-level functions for most analyses in *Administration*. The business objects are too 'stupid' to allow the subsystem *Analysis* to execute complex functions on them. Theoretically, it is also possible for the subsystem *Analysis* to directly access the database. However, this would also mean that the subsystem *Administration* no longer encapsulates the database, thus making modifications of the database schema more difficult.

Therefore, the subsystem *Business Objects* should be restructured in such a way that it becomes 'smarter' and the API of the subsystem *Administration* does not inflate so strongly.

5.10.3 Goal of the Refactoring

The New Object Model for Business Objects

This object model of the subsystem *Business Objects* shall now be modified in such a manner that the model of the core business objects will look as follows: each one of the *Employees* has got a *Month Folder* for each month with a *Calendar Sheet* for every work day. On the *Calendar Sheet* all *Time Entries* are recorded, including start and end time, *Project* and *Activity* in the project (see Figure 5-11).

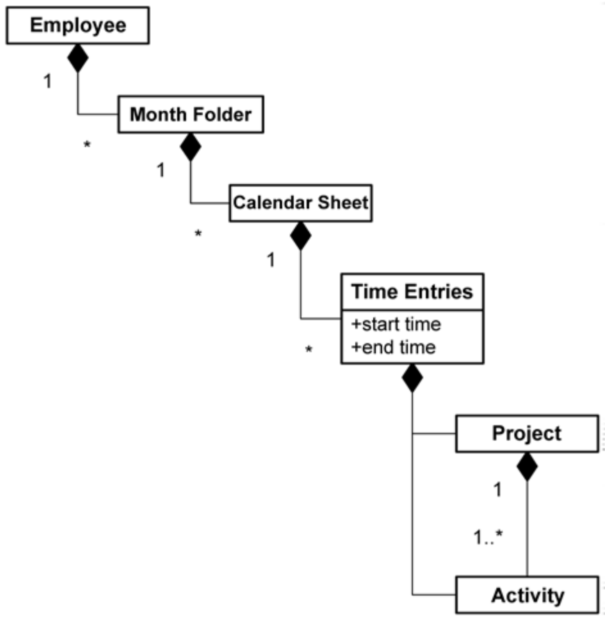
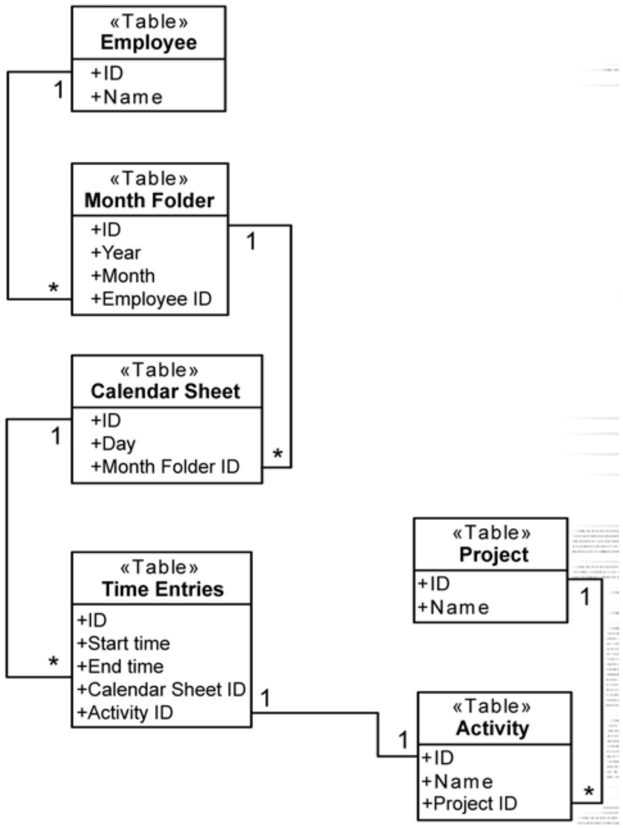


Fig. 5-11
Business Objects After Restructuring

Figure 5-12 shows the corresponding data model.

Fig. 5-12
Data Model After Restructuring



With this restructuring of the subsystem *Business Objects* we venture deeply into the system's vocabulary. We can expect a demand for comprehensive restructuring measures of the entire system. Here, we are going to focus on the refactoring's impact on database access, that is, on the data model and the *Administration* subsystem.

5.10.4 Refactoring Proceeding

The Refactoring Challenge

The difficulty this refactoring poses lies in the coordination of changes to the classes and those to tables. Both have to match for each separate step.

Unfortunately it is impossible to first view the class structure or the data model isolated from the rest and then deduce the respective other model from it. The main obstacle is that the uses relations in 1:N relations of the class model constitute a reversal of the data model. The *Calendar Sheet does* have a number of *Time Entries*, whereas in the data model *Time Entries* knows to which *Calendar Sheet* it belongs. If you transfer this example to the whole model, you will get a smell because cyclical relations are present.

The First Step

Therefore, we will proceed step by step, as we are used to. First, we are going to reverse the relations between *Time Entries* and *Employees* in the class model: Now *Time Entries* will no longer know the *Employees*, but *Employees* is assigned a certain amount of *Time Entries*. Figure 5-13 illustrates this refactoring step:

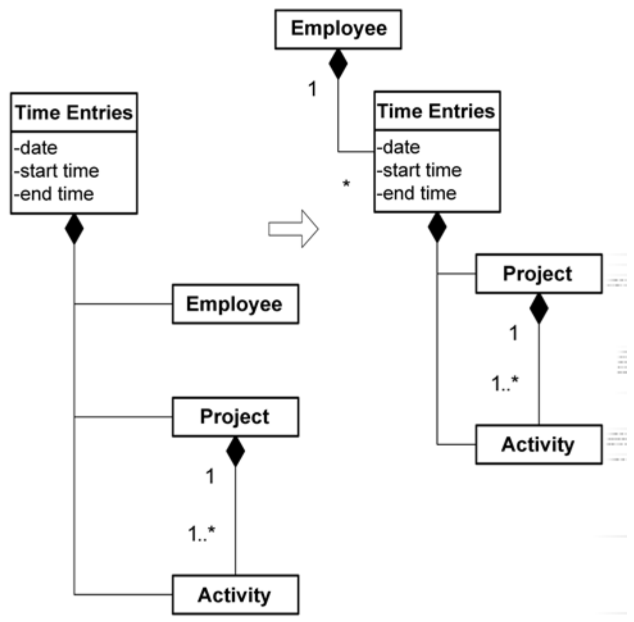


Fig. 5-13
Reversing the Relation between Employee *and* Time Entries

The class structure we just created must correspond with the data model of course. Interestingly, no modification of the data model is required to achieve this. The data model shown in Figure 5-10 can also display the new class model.

The Second Step

We will now extract the date information from *Time Entries* and put it in the class *Calendar Sheet*. For now, we will work without the *Month Folder* and store the complete date in *Calendar Sheet*. Figure 5-14 visualizes this refactoring step on the class model.

Fig. 5-14
Calendar Sheet *Contains Date Information*

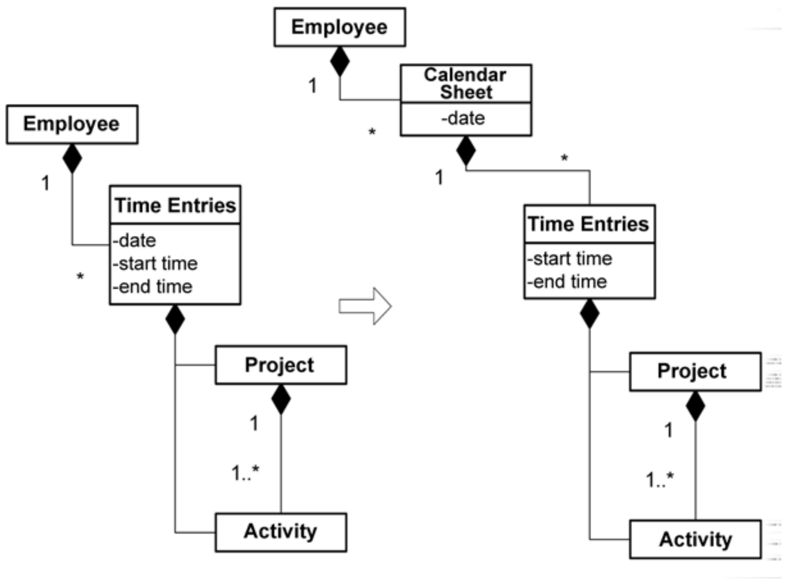


Figure 5-15 describes the matching data model. If the data model's tables have exclusively been used to load and save the business objects, the restructuring explained in Figure 5-15 can be executed as described, together with the class model's restructuring:

1. Add the new table *Calendar Sheet* to the data model.
2. When existing data shall be adopted: Copy data per SQL script from the existing tables into the new table.
3. Delete the fields *Date* and *Employee ID* from the *Time Entries* table.
4. Rearrange the business objects' class structure.
5. Adapt the mapping functionality in the subsystem *Administration*.

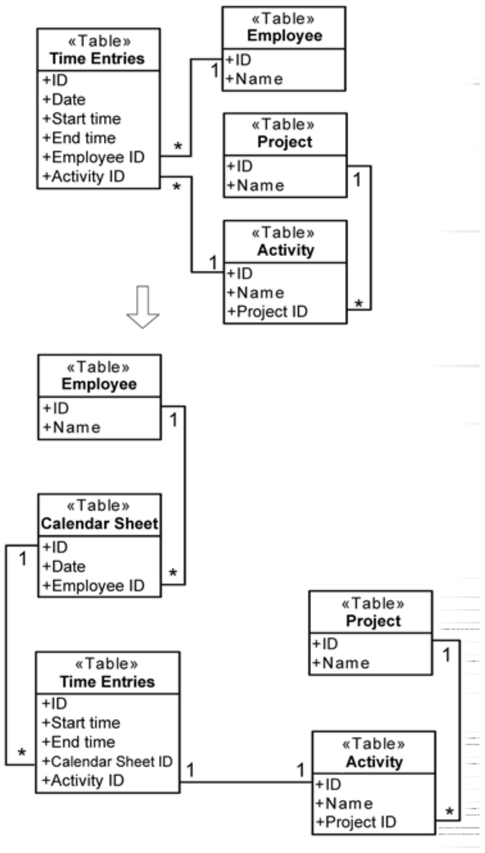
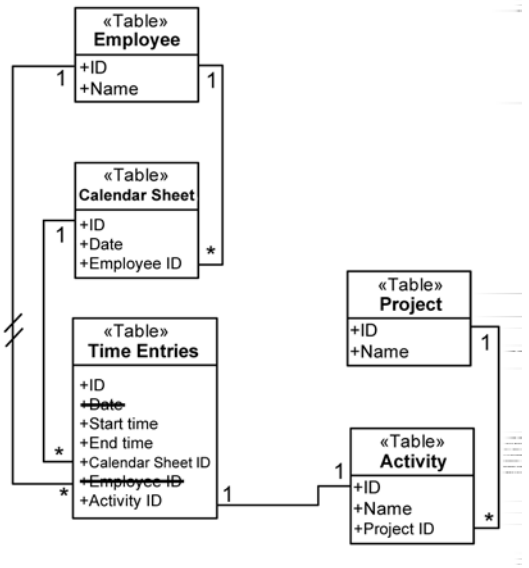


Fig. 5-15
Calendar Sheet
Contains Date Information

Should, however, several parts of the system access the tables, one cannot simply remove fields from the tables (*Date* and *Employee ID* from the *Time Entries* table). In this case, the fields must be set to *deprecated*, resulting in the data model shown in Figure 5-16.

Fig. 5-16
Calendar Sheet *Contains Data Information:* Deprecated *Fields*



The following steps are executed during this refactoring:

1. Add the new table *Calendar Sheet* to the data model.
2. Set the *Date* and *Employee ID* fields in the *Time Entries* table to *deprecated* (e.g. through an entry in the file *deprecated_db.txt*).
3. Rearrange the business objects' class structure.
4. Adapt the mapping functionality in the subsystem *Administration* so that it will also write the new table and its fields; if necessary using INSERT, should the respective set of data not yet exist in the *Calendar Sheet* table.
5. Step by step adjust all other write access instances in the system in such a way that old and new fields are written parallel.
6. When existing data shall be adopted: Copy data per SQL script from the existing tables into the new table.
7. Adapt the mapping functionality in the subsystem *Administration* so that none of the *deprecated* fields will be read any more.
8. Step by step delete all other read access to the *deprecated* fields.
9. Step by step delete all other write access to the *deprecated* fields – thus enabling reading from the new fields.
10. Delete *deprecated* fields.

Here it becomes clear that modifications of database structures can become quite tedious if access is not unambiguously channeled by few classes: In the beginning, *all* write access instances must be modified in such a way that they will write to old *and* new fields. Only then the read access instances can be adapted stepwise.

It is crucial not to deliver any releases to customers between single refactoring steps. Otherwise, there is a high risk that the fields won't be completely written to. Inconsistent data would be the consequence.

The Third Step

The third big step is extraction of the *Month* information from the *Calendar Sheet*. This step follows the same pattern as the second one and therefore isn't described here.

5.11 References

[AgileDB] Agile Datenbanken, mailing list: http://groups.yahoo.com/group/agileDatabases

This English language mailing list discusses database-related topics with a focus on agile methods.

[Ambler 2003a] Scott W. Ambler: The Process of Database Refactoring. http://www.agiledata.org/essays/databaseRefactoring.html. 2003.

An article that gives an overview of database refactorings, including descriptions of the development processes.

[Ambler 2003b] Scott W. Ambler: Catalog of Database Refactorings. http://www.agiledata.org/essays/databaseRefactoringCatalog.html. 2003.

A catalogue of frequently used refactorings of database schemas.

[Celko 1999] Joe Celko: SQL for Smarties – Advanced SQL Programming. 2nd ed., Harlekijn, 1999.

This book provides an introduction to advanced SQL concepts and presents suggestions for solutions for recurring modeling problems, such as the mapping of tree structures to relational databases.

[Fowler & Sadalage 2003] Martin Fowler, Pramod Sadalage: Evolutionary Database Design. http://www.martinfowler.com/articles/evodb.html, 2003.

This article explains the basic concepts of evolutionary database design. Refactorings of database schemas, the migration of data as well as refactorings of database access codes are examined.

[Sadalage & Schuh 2002] Pramod Sadalage, Peter Schuh: The Agile Database: Tutorial Notes. Presented at XP/Agile Universe 2002, www.xpuniverse.com.

Here, among other issues, the deprecated *marker of database elements is discussed.*

6 API Refactorings

In this chapter, we are going to examine the effects of refactorings on application programming interfaces (APIs) and the clients based on them. We will primarily focus on Java in this context. With justifiable effort, the results should be transferable to other object-oriented programming languages.

6.1 Subsystems

Subsystems

In each non-trivial software system, partitions can be found that are used by other partitions of the same system. Often this kind of structuring is specified: we talk about subsystems, class libraries, frameworks or components. They all have in common that they are clearly distinguished from other subsystems. A class always belongs to precisely one subsystem and is used by the rest of the system via an interface (API, i.e. application programming interface). To simplify matters, we will from now on summarize all these different partition types under the label *subsystems*.

The division into subsystems as well as the API's definition can either be implicit or explicit. For implicit subsystems, there is no specification which subsystems exist, what these subsystems are called, which classes belong to them, or which classes and methods constitute their API. The system's structure becomes much clearer when all these things are explicitly defined. For the explicit definition of subsystems, so-called component models are used, such as the Eclipse plug-in model, CORBA, COM, or a business/project-specific component model.

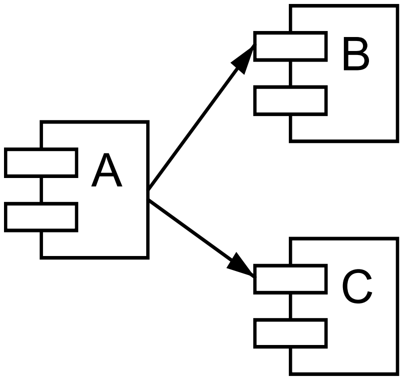


Fig. 6-1
Dependency between Subsystems

Subsystems for Reuse

Besides ensuring a clean structuring of the system, subsystems support reuse. First of all, they can be reused in a company's different projects. If a subsystem is rather common, it can either be commercially marketed or distributed as an open source component.

6.2 Problems of API Refactorings

Unfortunately, modifications of subsystems cannot always be limited to internal implementations. Occasionally, an API must be adapted as well in the course of a refactoring.

Anonymous Subsystem Users

If a subsystem is not only used for one project (internal reuse), but for various projects instead, maybe even in different companies (external reuse), the refactoring of APIs will become more difficult because the concrete code, which is based on the API, is unknown. This is the reason why code based on that subsystem cannot be changed instantaneously in the course of an API refactoring. If the API is broken, the dependent code has to migrate. A subsystem A depends on a subsystem B when interfaces or classes of subsystem B are used in subsystem A's source code.

For the purpose of API modifications, often a distinction between source code and binary compatibility is made. A modification is source code-compatible if the system can be compiled and its runtime behavior will still be the same after it has been modified. A modification is binary compatible if the system will be operable without prior new compilation. Interestingly, neither does source code compatibility imply binary compatibility, nor is this the case vice versa. However, we are concerned with source code compatibility, not binary compatibility in this chapter (the latter is discussed in [Rivières 01]).

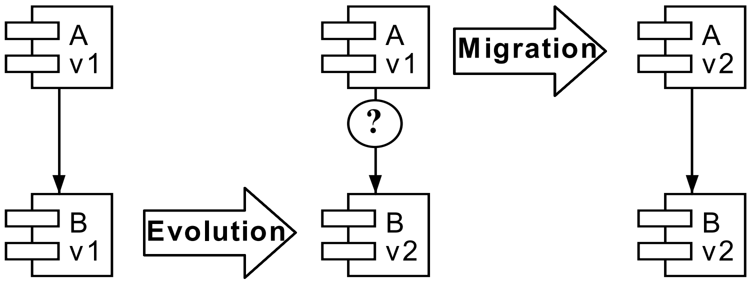


Fig. 6-2
Evolution and Migration

Stability of APIs

Sometimes demands not to change a subsystem's API any further after its publication are voiced. In practice, it soon becomes clear that meeting this demand would be purpose-defeating: On one hand, the API shall not be altered anymore. On the other hand, increased usage of the subsystem results in new requirements that can only be met through changing the API. Hence, we will try to build as stable APIs as possible although we know that we'll have to modify them sooner or later.

6.3 Compatibility Classes

Compatible and Incompatible Changes

Not every change of a subsystem's API will generate a demand for migration. At worst, compatible changes to the API will require a new compilation of the dependent code[1]. Regrettably, many more changes are incompatible than one would expect at first sight. Therefore, adding methods appears not to be a critical step. If an abstract method is added to an API class though, subclasses can be rendered invalid: they lack the implementation for the new, abstract method. The following tables convey an impression of compatibility classes[2] (we assume that non-constant fields are private and can thus be disregarded in this context). Non-private attributes can, if applicable, be adapted using the *Encapsulate-Field* refactoring.

1. In Java, usually not even a new compilation is necessary. There are a couple of interesting exceptions though, e.g. the changing of constant values, which is generated by the compiler in the client classes.
2. The comments on each compatibility class can be found in the descriptions of the different changes (see 6.5).

Changes to Interfaces:

No.	Change	Compatibility
1	*Adding an interface*	*Incompatible*
2	*Removing an interface*	*Incompatible*
3	*Renaming an interface (also: moving an interface into another package or renaming package)*	*Incompatible*
4	*Adding a superinterface*	*Incompatible, because dependent classes can become abstract*
5	*Removing a superinterface*	*Incompatible*

Changes to Classes

No.	Change	Compatibility
1	*Adding a class*	*Incompatible*
2	*Removing a class*	*Incompatible*
3	*Renaming a class (also: moving a class into another package or renaming package)*	*Incompatible*
4	*Changing a superclass*	*Incompatible*
5	*Adding an interface*	*Incompatible*
6	*Removing an interface*	*Incompatible*
7	*Expanding visibility*	*Compatible*
8	*Restricting visibility*	*Incompatible*
9	*Setting a class from final to non-final*	*Compatible*
10	*Setting a class from non-final to final*	*Incompatible*
11	*Setting a class to abstract*	*Incompatible*
12	*Setting a class from abstract to non-abstract*	*Compatible*

Changes to Constants in Classes and Interfaces

No.	Change	Compatibility
1	*Adding a constant*	*Compatible*
2	*Removing a constant*	*Incompatible*
3	*Changing a constant type*	*Incompatible*
4	*Changing a constant value*	*Compatible*

Changes to Methods in Interfaces

No.	Change	Compatibility
1	*Adding a method*	*Incompatible*
2	*Removing a method*	*Incompatible*
3	*Renaming a method*	*Incompatible*
4	*Changing a method's parameter list*	*Incompatible*
5	*Changing a method's return type*	*Incompatible*
6	*Adding an exception to a method in the interface*	*Incompatible*
7	*Removing an exception from a method in the interface*	*Incompatible*

Changes to Constructors in Classes:

No.	Change	Compatibility
1	*Adding a constructor*	*Incompatible, if the classes until now did not have an explicit constructor*
2	*Removing a constructor*	*Incompatible*
3	*Changing a constructor's parameter list*	*Incompatible*
6	*Expanding a constructor's visibility*	*Compatible*
7	*Restricting a constructor's visibility*	*Incompatible*

No.	Change	Compatibility
12	*Weakening a constructor's precondition*	*Compatible*
13	*Strengthening a constructor's precondition*	*Incompatible*
18	*Adding an exception to the constructor*	*Incompatible*
19	*Removing an exception from the constructor*	*Incompatible*

Changes to Methods in Classes:

No.	Change	Compatibility
1	*Adding a method*	*Incompatible, if the method is abstract; also incompatible, if the new method is final and "accidental" redefinitions take place in subclasses*
2	*Removing a method*	*Incompatible*
3	*Renaming a method*	*Incompatible*
4	*Changing a method's parameter list*	*Incompatible*
5	*Changing a method's return type*	*Incompatible*
6	*Expanding a method's visibility*	*Incompatible, if the method is redefined in subclasses*
7	*Restricting a method's visibility*	*Incompatible*
8	*Setting a method from final to non-final*	*Compatible*
9	*Setting a method from non-final to final*	*Incompatible, if the method is redefined in subclasses*
10	*Setting a method from static to non-static*	*Incompatible*
11	*Setting a method from non-static to static*	*Incompatible, if the method is redefined in subclasses*
12	*Setting a method to abstract*	*Incompatible*
13	*Setting a method from abstract to non-abstract*	*Compatible*
14	*Weakening a method's precondition*	*Incompatible, if the method is redefined*
15	*Strengthening a method's precondition*	*Incompatible, if the method is called*
16	*Weakening a method's postcondition*	*Incompatible, if the method is called*
17	*Strengthening a method's postcondition*	*Incompatible, if the method is redefined*

No.	Change	Compatibility
18	*Setting method to synchronized*	*Incompatible, if the method is used in a multi-threaded context*
19	*Setting method from synchronized to non-synchronized*	*Incompatible, if the method is used in a multi-threaded context*
20	*Adding an exception to a method in a class*	*Incompatible*
21	*Removing an exception from a method in a class*	*Incompatible*

Deprecated *Tag*

It turns out that most API changes are incompatible. In Java, the *deprecated* tag will provide some first assistance: it allows us to mark interfaces, classes and methods as obsolete. A class that shall be deleted will not be deleted right away but identified as *deprecated*. The class can still be used, although the compiler will generate a warning each time this happens. The dependent code can migrate step by step while staying compilable and executable at all times.

The following source code depicts how the *deprecated* tag denotes deletion of the class *MyClass*.

```
/**
  * @deprecated
  */
public class MyClass
{...}
```

Deferred-incompatible Changes

Use of the deprecated tag creates a new compatibility class. Such 'denoted' incompatible changes are called *deferred-incompatible*.

When we take a closer look at the table above, we will see that, in principle, the renaming of interfaces, classes and methods could be carried out automatically. One would merely need a machine-readable description of the changes to the API plus a program that reads in these descriptions and makes the necessary changes to the client. Especially for renamings, a simple mapping file in which the old as well as the new name is listed would suffice. Such a function for automated migration when package names are altered is already integrated in some development environments.

Changes that Can Be Automated

In this case, we speak of automatable changes and get the following compatibility classes:

No.	Compatibility Class	Migration
1	*Compatible*	*No migration required*
2	*Automatable*	*Automatic migration is possible in a single step; rather little effort needed*
4	*Deferred-incompatible*	*Stepwise migration is possible. The system stays compilable and executable all the time*
5	*Incompatible*	*Migration must be carried out completely. During migration, the system is neither compilable nor executable*

6.4 Refactoring Tags

To simplify the migration of dependent subsystems and enable *merciless refactorings*[3] also for published interfaces, we are going to introduce the concept of refactorings tags (see [Roock 04]), which serve to improve the compatibility of changes. Incompatible changes will become either deferred-compatible or even automatable.

In the following sections, we will show how the new meta tags affect the refactoring work on published APIs. Based on possible modifications of APIs, we will also show in detail how these can be executed in such a manner as to ensure compatibility.

6.4.1 The Future Tag

The *Future* tag demonstrates which form an interface, a class or a method will have in the future. If an API client uses an element with a *Future* tag, the developers can verify whether their usage of the element will still be valid in the future.

The simplest form of the *Future* tag specifies that the respective element will be deleted in the future. The following source code sample shows how the *Future* tag announces deletion of the class *MyClass*.

```
  * @future #undefined
  */
public class MyClass
{...}
```

3. The term *Merciless Refactoring* is derived from agile methods practice and emphasizes that here refactorings are a central part of everyday development work.

With the *deprecated* tag, Java offers a similar mechanism. The *deprecated* tag can be used as an acronym for the *Future* tag displayed above, supplemented with *#undefined*. The *deprecated* tag is interpreted by the Java compiler. Whenever an element marked deprecated is referenced, the compiler will give out a warning. The element will be compiled correctly though, and the system will remain completely operational.

However, no warning will be generated if the obsolete element as well as the referencing element can be found in one and the same class.

For classes, the *Future* tag can also denote changes to the modifier (visibility, *final*), as well as changes to the superclass and to implemented interfaces. For interfaces, this tag can also be used to mark changes to superinterfaces.

An Example

The inheritance relation between *Customer* and *Partner* can be marked as obsolete in the *Customer* class's comment. Here, the *Future* tag is used to denote that the inheritance relation will be deleted at some point.

```
/**
 * @future public class Customer
 */
public class Customer
  extends Partner
   ...
```

Thus, all direct uses of *Customer* and *Partner* stay valid. It is important that the client will be no longer allowed to make use of the inheritance relation between *Customer* and *Partner*, as it is the case with polymorphic assignments, for instance.

Even several changes can be described with the *Future* tag. In the following example, in the future the class *Customer* will no longer inherit from *Partner* and also no longer implement the *Serializable* interface. Instead, only the *Comparable* interface will be implemented.

```
/**
 * @future public class Customer implements Comparable
 */
public class Customer
   extends Partner
   implements Serializable, Comparable
   ...
```

For methods, changes of the modifiers can be described. Especially switches from non-final to *final* as well as changes of visibility can be elegantly expressed with the *Future* tag.

The following example shows how the *Future* tag denotes that the method *setName* will become final in the next version. Until the next subsystem version release, the client developers can eliminate all redefinitions of *setName*.

```
/**
 * @future public final void setName (String name)
 */
public void setName (String name)
    ...
```

Additionally, the tag can be supplemented with an informal description of what is to do now that the element cannot be longer used in the old way.

6.4.2 The Past Tag

Whereas the *Future* tag shows what an element will look like in the future, the *Past* tag describes what the element looked like in the past. This enables developers to see what the element's name was in the previous version. For classes and interfaces, it also contains information about in which packages the classes and interfaces were stored before.

The *Past* tag serves to visualize renamings and moves. In principle, the changes thus become automatable. The following example shows the *Past* tag for renaming a method *setName* to *setLastName*.

```
/**
 * @past public void setName (String name)
 */
public void setLastName (String name)
    ...
```

Migration can be accomplished in an even smoother manner not simply through renaming the method, but through duplicating it. The new version will refer to the old one via the *Past* tag, and the old version will be marked *deprecated*.

```
/**
 * @deprecated
 */
public void setName (String name) {
  setLastName(name);
}

/**
 * @past public void setName (String name)
 */
public void setLastName (String name)
    ...
```

In JDK, the succession method is often directly and informally appended to the *deprecated* tag.

```
/**
 * @deprecated Replaced by setLastName(String)
 */
public void setName (String name) {
  setLastName(name);
}

public void setLastName (String name)
    ...
```

6.4.3 Working with Refactoring Tags

The refactoring tags introduced here can also be usefully applied without the aid of special tools. The search options offered by modern development environments (e.g. Eclipse) are completely sufficient here.

Future *Tag*

First, the source code of a subsystem can be searched for all *Future* tags. Based on the elements found and supported by the development environment, one can determine in which places they are used. The developer must check the using elements and adapt them where necessary.

Past *Tag*

Similarly, *Past* tags in subsystems can be searched with the source code search function. The results will let developers conclude how these elements used were labeled before. The uses of the renamed elements can be roughly determined with a source code search, followed by a check of the detected uses and – if required – changes of their use.

Using the *deprecated* tag is even simpler. The compiler will point at the places in the client code where deprecated elements are used. These places must merely be analyzed. The more information has been added to the *deprecated* tag (e.g. 'replaced with'), the easier migration will be.

6.4.4 Tools for Migration

Specialized tools facilitate the handling of refactoring tags. Aided by the *Past taglet*, the subsystem developers can analyze the subsystem APIs' *Past* tags. The Past taglet will write the detected renamings to a file. Then this file is – together with the new subsystem version – delivered to the subsystem's clients. Here, the renaming file serves as input for the *Renamer*, which carries out the required renamings in the client code.

A first version of these open-source tools is available for download as an Eclipse-plug-in at: *https://sourceforge.net/projects/jmigrator.*

In addition, we plan to implement the *Future Warner*: It will check the client code for future invalid use of the API. Whenever such an invalid use is identified, a warning will be issued. Then the client developers could change the client code in such a way that it would function with a future subsystem version.

6.5 API Refactorings in Detail

If a subsystem's API is modified, two kinds of conflicts can emerge: structural conflicts and behavior conflicts. Structural conflicts prevent the system's compilability. In the case of a behavior conflict, the system will still be compilable, although its execution will be faulty. However, a clean test coverage will at least help to identify and systematically eradicate behavior conflicts.

In this section, we are going to explain for each API modification which conflicts it will create and how it can best be carried out compatibly.

The goal of changes to APIs is always to maintain compatibility with existing clients. One-hundred per cent security can hardly be ever reached. Many of the techniques for API refactorings presented here function based on copying a method and then pasting it with a new name. Admittedly, the generation of a method in turn will be incompatible. In practice though, such change does hardly ever lead to problems. Therefore, we accept that there is no such thing as one-hundred per cent security. We will content ourselves with a high compatibility probability.

Next, we will describe typical modifications. We always adhere to the premise that non-constant attributes in classes are always private.

This is why we won't consider the possible changes to attributes and their consequences any further.

New and Temporary Method Names

During the following refactorings, you will frequently encounter situations in which methods are not simply deleted or modified. Instead, they are copied and saved with a new name. Only later on the old version of the method will be deleted. The problem here is to find good, i.e. meaningful names for the methods. Let us assume that the old name was meaningful. Now we have to find an equally good, i.e. meaningful name to replace the old one. Alternatively, we can mark the new name as temporary, adhering to the respective convention (for example, the old name could be supplemented with the ending *_TEMP*) and change it back in the next version of the subsystem.

6.5.1 Changes to Interfaces

Adding an Interface

In most cases, adding an interface is compatible. The change will become incompatible though if an interface of the same name already exists in another package. Should the client import both packages with *, an ambiguity will be the result, and the client can no longer be compiled.

The change will not even become compatible when a subsystem's interface names are unambiguous without package names. Last but not least, an interface with an identical name can also be defined in another subsystem. Nevertheless, interface names should be unique for each subsystem. Thus the risk of ambiguities will not be entirely eliminated, but at least reduced.

Removing an Interface

The removal of interfaces is incompatible. If the interface is not immediately removed but set to *deprecated* instead, the change becomes deferred-incompatible.

Renaming Interfaces

The renaming of an interface is incompatible. One could copy the interface with the new name and set the old version to *deprecated*. However, this approach could easily create type problems, even if the new interface inherits from the old one or vice versa.

If the interface is renamed and the old name annotated with the *Past* tag, the change will become automatable.

Here is an example for renaming the *Customer* interface into *Partner*:

```
/**
 * @past public interface Customer
 */
public interface Partner
...
```

It should not go unmentioned that a change carried out with the *Past* tag is not always automatable. If an interface of the new name does already exist in another package, this can lead to an ambiguity (see also ‘Adding an Interface’).

Adding a Superinterface

When another interface is added to the list of interfaces that inherited, we will receive an incompatible change. Client classes that implement this interface will become abstract because they do not implement the methods of the new superinterface.

If the client class previously owned methods that now ‘coincidentally’ implement the superinterface’s methods, a behavior conflict can emerge.

The change will become deferred-incompatible when the interface is not directly added to the interfaces that inherited and the change is denoted only with the *Future* tag instead. The Future warner can detect those classes that must implement the new interface in the future. In this way, the client developers can adapt their code before the actual change is executed.

Here is an example of how the *Future* tag is used:

```
/**
 * @future public interface Customer
 *            extends Partner
 */
public interface Customer
...
```

Removing a Superinterface

If an interface is removed from the list of those interfaces that inherited, we are faced with an incompatible change. Client classes that use the interface for typing demand more methods than the interface will offer after the change.

This change will become deferred-incompatible if the interface is not directly removed from the list of interfaces that inherited, but the change is merely denoted with the *Future* tag instead. The Future warner can detect those classes that will expect methods which no longer exist in the future. The client developers can adapt their code before the actual change is executed.

An example of the *Future* tag's use:

```
/**
 * @future public interface Customer
 */
public interface Customer extends Partner
...
```

6.5.2 Changes to Classes

Adding a Class

As a rule, the addition of a class is compatible. The change will become incompatible though when a class of the same name already exists in another package. If the client imports both packages with *, an ambiguity will emerge, and the client can no longer be compiled. The only exception to this rule is the occurrence of a client coincidentally compiled with the wrong class. In such case it is very likely that a behavior conflict will emerge.

The change will not even become compatible if the class names of a subsystem are unique without being assignable to package names. After all, a class of the same name can also be defined in another subsystem. Nevertheless, class names should be unique for each subsystem. The risk of ambiguities will not be entirely eradicated, but at least reduced.

Removing Classes

The removal of classes is incompatible. If the class is not immediately removed but set to *deprecated* instead, the change will become deferred-incompatible.

Renaming Classes

The renaming of a class is incompatible. Theoretically, one could copy the class, assign it the new name and set the old version to *deprecated*. This can easily lead to type problems though, even if the new class inherits from the old one or vice versa.

If the class is renamed and the old name annotated with the *Past* tag, the change will become automatable.

Look at the example for renaming the class *Customer* in *Partner*:

```
/**
 * @past public class Customer
 */
public class Partner
...
```

Unfortunately, changes using the *Past* tag cannot always be automated. If a class with the new name already exists in another package, an ambiguity might be created (see also 'Adding a class').

Changing a Superclass

Changes of the superclass are incompatible. At first, polymorphic assignments will become invalid.

For example:

If *Customer* is a subclass of *Partner*, and *Partner* is exchanged as a superclass, all assignments of *Customer* to variables of the *Partner* type will become invalid.

Moreover, the subclasses of the modified classes will become abstract if the new superclass defines abstract methods. Should the subclasses 'coincidentally' define the abstract methods, a behavior conflict will be the outcome.

If the *Future* tag is used to denote changes, it will be rendered deferred-incompatible. The Future warner can identify those classes which will either expect methods that will no longer exist in the future or no longer implement defined abstract methods. This allows client developers to adapt their code before the actual change is made.

The following is an example of the *Future* tag's use:

```
/**
 * @future public class Customer
 *             extends Person
 */
public class Customer extends Partner
...
```

Adding an Interface

The adding of an interface to the list of interfaces that are implemented by the class is incompatible: existing subclasses will become abstract.

If the subclasses ‘coincidentally’ define the methods, a behavior conflict will emerge.

The change will become deferred-incompatible if the addition of the interface is denoted with the *Future* tag.

For example:

```
/**
 * @future public class Customer
 *           implements Storable
 */
public class Customer
...
```

Removing an Interface

The removal of an interface from the list of interfaces implemented by the class is incompatible: the class’s objects are no longer assignable to the remote type.

The change will become deferred-incompatible if the removal of the interface is denoted with the *Future* tag.

For example:

```
/**
 * @future public class Customer
 */
public class Customer implements Storable
...
```

Expanding Class Visibility

The expansion of class visibility is compatible.

Restricting Class Visibility

The restriction of class visibility is incompatible.

The change will become deferred-incompatible if the visibility restriction is denoted with the *Future* tag.

For example:

```
/**
 * @future class Customer
 */
public class Customer
```

```
...
```

Setting a Class from *Final* to Non-*final*

If a class that is declared *final* is set to non-*final*, the change is compatible.

Setting a Class from Non-*final* to *Final*

If a class is set from non-*final* to final, a structural conflict will be the result: existing subclasses will be rendered invalid.

The change will become deferred-incompatible if it is not executed directly, but denoted with the *Future* tag instead.

Here is an example of the *Future* tag's use:

```
/**
 * @future final public class Customer
 */
public class Customer
...
```

Setting a Class to *Abstract*

If a concrete class becomes abstract, we are faced with an incompatible change. When objects of this class are created, those create statements will become invalid.

The change will become deferred-incompatible if it is denoted with the *Future* tag.

For example:

```
/**
 * @future public abstract class Customer
 */
public class Customer
...
```

Setting a Class from *Abstract* to Non-*abstract*

If an abstract class becomes concrete, we are faced with a compatible change – at least as long as no new methods that were previously abstract must be added to the class. If that was the case, we could get a behavior conflict.

6.5.3 Changes to Constants in Interfaces/Classes

Adding a Constant

The addition of a constant is compatible.

Removing a Constant

If a constant is removed, a structural conflict will occur.

The change will become deferred-incompatible if the constant is not deleted, but marked as *deprecated* instead.

Changing a Constant Type

The changing of a constant type is incompatible.

The change will become deferred-incompatible if a new constant with the desired type is created while the old constant is set to *deprecated*.

For example:

```
interface Printer {
  /**
   * @deprecated
   */
  public static final int LASERPRINTER=1;
  /**
   * @deprecated
   */
  public static final int INKJETPRINTER=2;

  public static final String
    LASERPRINTER_TYP="laser";
  public static final String
    INKJETPRINTER_TYP="ink";
}
```

Changing a Constant Value

The changing of a constant value is – as a rule – compatible. However, if a number of constants constitutes the value range of an enumeration type, the change can create a behavior conflict. This is the case when the client's value range has been expanded by constants of its own and the values used there are in conflict with the new constant value.

6.5.4 Changes to Methods in Interfaces

Adding a Method to an Interface

If a method is added to an interface, a structural conflict will be created. Existing implementations of the interface will become abstract, because they don't possess an implementation of the new method. If, by chance, a suitable method already happens to exist in an implementation, this method will be implemented 'accidentally.' This can lead to a behavior conflict.

The change will become compatible if the interface is not directly implemented in the application, but application classes are derived from default implementations instead. Then the subsystem developers can provide a suitable method implementation in the default implementation.

For example:

```
public interface Window {
  public void setWidth(int w);
  public void setHeight(int h);

  // new method: setSize
  public void setSize(int width, int height);
}

public class DefaultWindow implements Window {
  private int width, height;

  public void setWidth(int w) {
    width = w;
  }

  public void setHeight(int h) {
    height = h;
  }

  // new method: setSize
  public void setSize(int w, int h) {
    setWidth(w);
    setHeight(h);
  }
}
```

Removing a Method from an Interface

The removal of a method from an interface is incompatible. The change will become deferred-incompatible if the method is not directly deleted, but set to *deprecated* instead.

Renaming a Method in an Interface

If a method in an interface is renamed, a structural conflict will be the result. Generally, the change will become automatable if the method's previous name is defined in the *Past* tag.

For example:

```
public interface Customer {
  /**
   * @past void setName(String name)
   */
  public void setLastName(String name);
}
```

Yet it is possible to experience situations where the changes will remain incompatible. This is going to be the case if a method of the same name and parameters but with a different return type does already exist in either a subinterface or an implementation of this interface. If a method with the same parameters and a matching return type exists in an implementation, a behavior conflict can occur because the renamed method will be automatically implemented by the method in that implementation. If a default implementation exists for the interface and classes are never directly implemented in that interface, but succeed the default implementation instead, the change can also be handled with the *deprecated* tag: In this case, the method must be duplicated in both the interface and the default implementation. Also, the old method must be set to *deprecated*.

Default Implementations of Interfaces

Changing the Parameter List of a Method in the Interface

If the parameter list of a method in an interface is changed, a structural conflict will emerge. The change will become deferred-incompatible unless it is executed directly. Instead, the method will be copied and the copy will be changed. The old method must be set to *deprecated*.

For example:

```
public interface Customer {
  /**
   * @deprecated
   */
  public void setName(String name);
  public void setName(String lastname,
                      String firstname);
}
```

Changing the Return Type of a Method in the Interface

If the return type of a method in an interface is changed, the change will be incompatible. Existing implementations of this method in the client will become invalid.

The change will become deferred-incompatible when a new method with a new name and the desired return type is created. The old method will be marked with the *deprecated* tag.

We will have to find a new name for the new method if the programming language (in this case Java) does not allow for defining a number of methods in one class that can only be distinguished by their return type.

For example:

```
public interface Customer {
  /**
   * @deprecated
   */
  public String getname();
  public Name getCustomername();
}
```

Adding an Exception to a Method in the Interface

The addition of an exception to a method is incompatible because the client code will have to catch this exception.

The change will become deferred-incompatible if the method is copied and generated together with the desired exception list under a new name. The old method must be set to *deprecated*.

For example:

```
public interface Printer {
  /**
   * @deprecated
   */
  public void print(Document d);
  public void printDoc(Document d)
    throws PrinterException;
}
```

Removing an Exception from a Method in the Interface

The removal of an exception from a method is incompatible because the client code is not allowed to catch this removed exception. In addition, redefinitions of the method will become invalid because they expand the exception list.

The change will become deferred-incompatible if the method is copied and generated with a new name together with the desired exception list. The old method will be set to *deprecated*.

For example:

```
public interface Printer {
  /**
   * @deprecated
   */
  public void print(Document d)
    throws PrinterException;
  public void printDoc(Document d);
}
```

6.5.5 Changes to Constructors in Classes

Adding a Constructor

When a new constructor is added to a class, incompatibilities will emerge, as long as no explicit constructor exists. In this case, the compiler will no longer generate the default constructor. Clients that until now have been using the default constructor will become invalid.

The change will become compatible if one always creates an explicit default constructor whenever the first constructor is inserted. This problem can be avoided right from the start when always at least one explicit constructor is created.

Removing a Constructor

The removal of a constructor from a class is incompatible. The change will become deferred-incompatible if the constructor is not directly deleted, but set to *deprecated* instead.

Changing a Constructor's Parameter List

If the parameter list of a constructor is changed, a structural conflict will occur. The change will become deferred-incompatible unless the change is made directly. Instead, the constructor will be copied and the copy will be changed. The old constructor will be set to *deprecated* and call the new constructor.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  public Customer(String name) {
    this(name, „“);
  }

  public Customer(String lastname,
                 String firstname)
  {..}
}
```

Expanding Constructor Visibility

If constructor visibility is expanded, the change will be compatible.

Restricting Constructor Visibility

If constructor visibility is restricted (e.g. from *public* to *protected*), an incompatible change will be the result. Clients that use the respective constructor will become invalid because the constructor is no longer visible to them.

The change will become deferred-incompatible if the restriction of the constructor's visibility is denoted with the *Future* tag.

For example:

```
public class Customer {
  /**
   * @future protected Customer(String name)
   */
  public Customer(String name)
  {..}
}
```

Weakening of a Constructor's Precondition[4]

If a constructor's precondition is weakened, a compatible change will be the outcome.

Strengthening of a Constructor's Precondition

If a constructor's precondition is strengthened, its uses will become invalid. Thus the change will be incompatible.

Adding an Exception to a Constructor

The addition of an exception to a constructor is incompatible because the client code has to catch this exception.

Removing an Exception from a Constructor

The removal of an exception from a constructor is incompatible because the client code is not allowed to catch this removed exception.

6.5.6 Changes to Methods in Classes

Adding Methods to a Class

If a new, non-private method is added to a class, incompatibilities will arise. If a method of the same name and parameters but with a different return type exists in a subclass, a structural conflict will emerge. Even if the method is defined with identical parameters and an identical return type in the subclass, the structural conflict will be inevitable if the method in this subclass is less visible. Should the method's signature happen to be the same as the signature of the new method, a behavior conflict is likely to result, because the new method will accidentally be overwritten by the subclass.

4. Pre- and postconditions refer to the contract model based on the design-by-contract principle (cf. [Meyer 92]).

Removing Methods from Classes

The removal of a method from a class is incompatible. The change will become deferred-incompatible if the method is not directly deleted but set to *deprecated* instead.

Renaming Methods in Classes

If a method in a class is renamed, a structural conflict will be the result. As a rule, the change will become automatable if the method's previous name is defined in the *Past* tag.

For example:

```
public class Customer {
  /**
   * @past void setName(String name)
   */
  public void setLastName(String name);
}
```

Alternatively, the old method can also be copied and saved with the new name. Then the old method must be set to *deprecated*. In this case, the change will not become automatable, but at least it will be deferred-incompatible. Yet it is possible to experience situations where the changes will remain incompatible. This is going to be the case if a method of the same name and parameters but with a different return type already exists. Should a method of the same name, the same parameters plus a matching return type exist in the subclass, a behavior conflict can emerge because the renamed method will be overwritten by the method in the subclass.

Changing Parameter List of a Method in a Class

If the parameter list or the return type of a method in a class is changed, a structural conflict will emerge. The change will become deferred-incompatible unless the change is not made directly. Instead, the method will be copied and the copy will be changed. The old method will be set to *deprecated* and call the new method.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  public void setName(String name) {
    setName(name, „“);
  }

  public void setName(String lastname,
                      String firstname)
  {..}
}
```

Changing the Return Type of a Method in a Class

If the return type of method in a class is changed, the change is incompatible. Existing redefinitions of this method in the client will become invalid. If the new return type is no subtype of the old one, the uses of the respective methods will also be rendered invalid.

The change will become deferred-incompatible if a new method with the new name and the desired return type are created. The old method will be marked *deprecated* and their implementation will refer to the new method.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  public String getName() {
    return getCustomername().toString();
  }

  public Name getCustomername()
  {..}
}
```

Expanding Method Visibility in a Class

If method visibility in a class is expanded (e.g. from *protected* to *public*), we will get an incompatible change. Existing redefinitions of this method will become invalid because they restrict visibility.

The change will become deferred-incompatible if a copy of the method with the desired visibility is generated and saved with another name. The old method will be set to *deprecated*. The new method will refer to the old method.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  protected String getName()
  {..}

  public String getCustomername() {
    return getName();
  }
}
```

Restricting Method Visibility in a Class

If method visibility in a class is restricted (e.g. from *public* to *protected*), an incompatible change will result. Clients of this method will become invalid, because the method will no longer be visible to them.

The change will become deferred-incompatible if a copy of the method with the desired visibility is generated and saved with another name. The old method will be set to *deprecated* and refer to the new method.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  public String getName() {
    return getCustomername();
  }
  protected String getCustomername()
  {..}
}
```

Setting a Method in a Class from *Final* to Non-*final*

If a method declared final is set to non-final, the change will be compatible.

Setting a Method in a Class from Non-*Final* to *Final*

If a method is set *final*, the change will be incompatible because existing redefinitions of that method have become invalid.

The change will become deferred-incompatible if the old method is copied and inserted under a new name. The new method will be declared *final* and call the old method.

For example:

```
public class Customer {

  /**
   * @deprecated
   */
  public String getName()
  {..}

  public final String getCustomername() {
    return getName();
  }
}
```

Setting a Method in a Class from *Static* to Non-*static*

If a method is set from *static* to non-*static*, the change will be incompatible. Calls via the class name will be rendered invalid through the change. Now, one instance of the class will always be required.

The change will become deferred-incompatible if a new, non-static method is generated under a new name. At the same time, a static variable that contains a default instance of the respective class will be introduced into the class. The old method will be set to *deprecated* and call the new method on this default instance.

For example:

```
public class Printer {
    private static Printer defaultPrinter =
                              new Printer();

    /**
     * @deprecated
     */
    public static void print(Document d) {
      defaultPrinter.printDoc(d);
    }

    public void printDoc(Document d)
    {..}
  }
```

Setting a Method in a Class from Non-*static* to *Static*

If a method is set to *static*, the change is incompatible. Redefinitions of methods in client classes will be rendered invalid through the change.

The change will become deferred-incompatible if a new, static Method is generated under a new name. The old method will be set to *deprecated* and call the new method.

For example:

```
  public class Printer {

    /**
     * @deprecated
     */
    public void print(Document d) {
      printDoc(d);
    }

    public static void printDoc(Document d)
    {..}
  }
```

Setting a Method to *Abstract*

If a method that was until now concrete is changed into an abstract method, the resulting change will be incompatible: existing subclasses will become abstract.

The change will become deferred-incompatible if the change is denoted with the *Future* tag.

For example:

```
public abstract class Printer {
  /**
   * @future abstract print(Document d)
   */
  public void print(Document d)
  {
    printDoc(d);
  }
}
```

Setting a Method from *Abstract* to Non-*abstract*

If an abstract method becomes non-abstract, we will get a compatible change.

Weakening of a Method's Precondition in a Class

If a method's precondition is weakened, uses of this method will remain valid. However, redefinitions of this method will become invalid, because they expect the old precondition. It is not permissible though to strengthen the precondition in a class. Thus the change is incompatible.

The change will become deferred-incompatible if a new method with the desired precondition is created under a new name.

For example:

```
public class Printer {

  /**
   * @deprecated
   * @require d != zero
   */
  public void print(Document d) {
    printDoc(d);
  }

  /**
   * @require true
   */
  public void printDoc(Document d)
  {..}
}
```

Strengthening of a Method's Precondition in a Class

If a method's precondition is strengthened, redefinitions of this method will remain valid. However, uses of this method will become invalid. Thus the change is incompatible.

The change will become deferred-incompatible if a new method with the desired precondition is created under a new name.

For example:

```
public class Printer {

  /**
   * @deprecated
   * @require true
   */
  public void print(Document d)
  {..}

  /**
   * @require d != zero
   */
  public void printDoc(Document d) {
    print(d);
  }
}
```

Weakening of a Method's Postcondition in a Class

If a method's postcondition is weakened, redefinitions of this method will remain valid. However, uses of this method will become invalid. Thus the change is incompatible.

The change will become deferred-incompatible if a new method with the desired postcondition is created under a new name.

For example:

```
public class Printer {

  /**
   * @deprecated
   * @ensure d.hasbeenprinted()
   */
  public void print(Document d)
  {..}
```

```
    /**
     * @ensure true
     */
    public void printDoc(Document d) {
      print(d);
    }
  }
```

Strengthening of a Method's Postcondition in a Class

If a method's postcondition is strengthened, uses of this method will remain valid. However, redefinitions of this method will become invalid. Thus the change is incompatible.

The change will become deferred-incompatible if a new method with the desired postcondition is created under a new name.

For example:

```
  public class Printer {

    /**
     * @deprecated
     * @ensure true
     */
    public void print(Document d) {
      printDoc(d);
    }

    /**
     * @ensure d.hasbeenprinted()
     */
    public void printDoc(Document d)
    {..}
  }
```

Setting a Method in a Class to *Synchronized*

If a method that is declared non-*synchronized* is set to *synchronized*, there will be the rare case in which this change is incompatible. It can trigger deadlocks, and a behavior conflict will be the consequence.

The change will become deferred-incompatible if the method is copied and inserted as *synchronized* under a new name. The old method will be set to *deprecated*.

For example:

```
public class Printer {

  /**
   * @deprecated
   */
  public void print(Document d)
  {..}

  public synchronized void printDoc(Document d)
  {..}
}
```

In contrast to the usually applied duplicating of methods, the original method will not be simply delegated to the new method. If this was done, the aforementioned deadlock situation would occur. Instead, the new method can either call the old one or the implementation itself will be copied.

Setting a Method in a Class from *Synchronized* to Non-*synchronized*

If a method that declared *synchronized* is set to non-*synchronized*, the change will be incompatible. Multi-threaded applications can display an aberrant behavior after this change has been made. A behavior conflict will emerge.

The change will become deferred-incompatible if the method is copied and inserted as non-*synchronized* under a new name. The old method will be set to *deprecated*.

For example:

```
public class Printer {

  /**
   * @deprecated
   */
  public synchronized void print(Document d)
  {..}
```

```
    public void printDoc(Document d) {
      print(d);
    }
  }
```

Adding an Exception to a Method in a Class

The addition of an exception to a method is incompatible because the client code must catch this exception.

The change will become deferred-incompatible if the method is copied and generated under a new name with the desired exception list. The old method will be set to *deprecated*.

For example:

```
  public class Printer {

    /**
     * @deprecated
     */
    public void print(Document d)
    {..}

    public void printDoc(Document d)
      throws PrinterException {
      print(d);
    }
  }
```

Removing an Exception from a Method in a Class

The removal of an exception from a method is incompatible because the client code is not allowed to catch this removed exception. Moreover, the method's redefinitions will become invalid because they expand the exception list.

The change will become deferred-incompatible if the method is copied and generated under a new name with the desired exception list. The old method will be set to *deprecated*.

For example:

```
public class Printer {

  /**
   * @deprecated
   */
  public void print(Document d)
    throws PrinterException {
    printDoc(d);
  }

  public void printDoc(Document d)
  {..}
}
```

6.6 Converter

The refactoring tags described here clearly aim at keeping the new interface of the modified subsystem temporarily backwards-compatible with the old version. Thus the interface gets 'polluted' with methods that are not needed by the new client in the subsystem.

Converters that convert object structures between different versions are an alternative here. If they are used, the subsystem's API will not be altered. The subsystem is copied instead, so that the old as well as the new version of the subsystem can be used parallel. Often the new version of the subsystem will receive the suffix '2.'

Since in many cases it is not possible to adapt the entire application at once, some parts of it will continue to work with the subsystem's old version for the time being, while other parts are already using the new subsystem version. If the different parts of the application have to communicate with each other, the object structures of the old and the new subsystem versions must be bidirectionally convertible.

For this purpose, developers of a duplicated subsystem can supply one converter or more. In this way, the system's 'pollution' will be limited, and the convention of adding the suffix '2' will make it sufficiently clear to anyone that the old subsystem version will soon be history. Moreover, all classes of the old subsystem version will be set to *deprecated*, of course.

Converters do display definite limitations though, if classes of the modified subsystem have been inherited by other subsystems. In such case, it will no longer be feasible to construct a general converter with simple means.

6.7 Application Migration with Incompatible Subsystem Changes

Unless the subsystem developers alleviated their modifications of the subsystem API by using the aforementioned tags, the application developers will be in for an unpleasant surprise: after the subsystem's new version has been installed, the application will be no longer compilable. The compiler will generate countless error messages. Unfortunately, the number of error messages provides little valuable information. Several of them will be sequence errors, so that less changes must be made than the mass of messages at first suggests. But single migration steps can in turn produce new sequence errors – for example, because a developer notices that a parameter list of a method in the application must be adapted. All in all, the demand for adaptation can hardly be precisely projected. This creates a lot of insecurity for further project planning. Thus the migration to a new subsystem version becomes a relevant risk.

In addition, the application will remain uncompilable during the entire migration period. This means that neither the application itself nor tests can be executed. Whether all single parts of the application migrated correctly or not will only become clear at the very end of the migration process.

Migrating an Application Layer by Layer

This problem can be countered with a stepwise, new construction of the application. Please proceed as follows:

1. Install the new version of the subsystem.
2. Create a new, empty version of the application project, including references to the new subsystem version.
3. Copy the application's lowest layer into the new project.
4. Make the copied application classes compilable again.
5. Change the copied application classes so that they will pass the tests again. The tests will also let you discover and resolve semantic conflicts (which requires a good test coverage, of course).
6. Copy the next application layer. Proceed with step 4.

Figure 6-3 visualizes this procedure. However, of course the migration effort will not become smaller when these instructions are being followed, but at least the already migrated parts of the application will be compilable and run the according tests. This migration method will significantly reduce the risks involved.

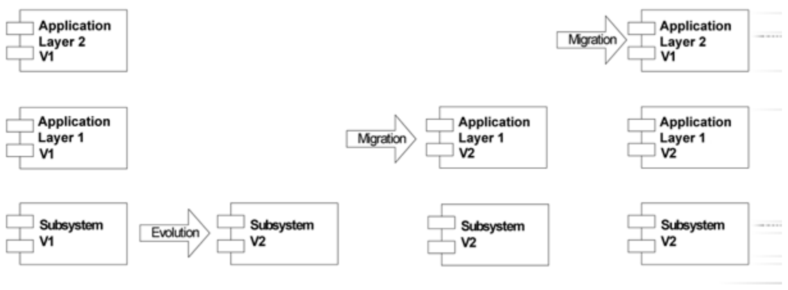


Fig. 6-3
New Construction of the Application for Migration

6.8 Tips for Designing APIs

Only a few, simple tips will help to design APIs in such a manner that they will be stable regarding changes.

Design Tip 1: Planning Inheritance

Inheritance must be planned. If inheriting from an API class hasn't been explicitly planned and scheduled beforehand, the class should be set to *final*. Methods that are not explicitly meant for overwriting should be declared *final* or *private*. Methods that in principle inherit from subclasses but do not belong to the class's normal API, can be labeled *protected*.

The keyword *final* constitutes a very powerful restriction for clients and can thus lead to problems. Especially when writing tests with JUnit, often scenarios will be created where a referenced class must be replaced by a specific version (e.g. Mock or Dummy). If the referenced class has been declared *final*, the testability via Mock or Dummy classes (which, as a rule, inherit from the class) will be impaired. Interfaces come in handy here: the actual implementation class is declared *final*, whereas the interface can be utilized for Mock and Dummy implementations.

Design Tip 2: Avoiding Inheritance

If inheritance between API classes can be avoided, it should be avoided: otherwise API clients can build on these inheritance relations.

Design Tip 3: Abstract Implementations for Interfaces

API clients that implement an API interface will become more stable regarding changes to the interface if they do not directly implement the interface, but inherit from an abstract class. If a new method is added to an interface, a default implementation will be defined in the abstract class, so that API clients must not be adapted.

Here, we are facing an area of potential conflict: In the chapter about architecture smells, we argued that list-like inheritance hierarchies point at speculative generalization. Now, in this chapter, we are suddenly suggesting use of exactly these list-like inheritance hierarchies. The reason for this suggestion is easily explained: In the case depicted here, the list-like inheritance hierarchy has deliberately been applied in order to smooth the way for modifications of subsystem interfaces. This is an ostensive example of the fact that not every smell automatically signals the existence of a problem.

Of course the abstract implementations for interfaces will function only as long as the client classes implement only a single interface. If several interfaces are supposed to be implemented, inheritance from several abstract implementations will prove impossible (at least in languages with single inheritance).

A way out of this dilemma is provided by the *Adaptable pattern* used in Eclipse (see [Gamma & Beck 03]).

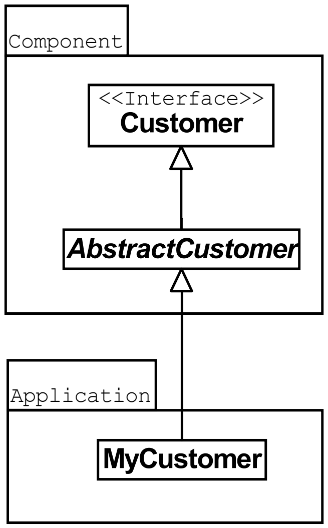


Fig. 6-4 *Abstract Implementations for Interfaces*

Design Tip 4: Small APIs

The smaller the API is, the lower the probability will be that the API must be changed incompatibly. Subsystems should be designed in such a way that as many classes as possible are hidden behind an as small as possible API.

All elements should have as little visibility as possible.

Design Tip 5: API in Its Own Packages

If the API as well as the implementation of a subsystem are respectively organized in packages of their own, changes to the implementation can be easier kept separate from the API.

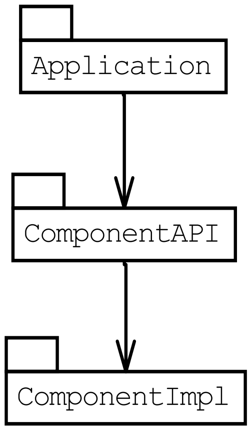


Fig. 6-5
Division between API and Implementation

Design Tip 6: Unambiguous Class Names

Class names in a subsystem API should be unambiguous, even without package names. Otherwise, problems can arise during migration when classes are moved into other packages.

Design Tip 7: Explicit Default Constructor

Each class should have at least one explicit constructor to prevent the compiler from generating the default constructor. Otherwise, the compiler-generated default constructor can be deleted 'accidentally' when a parameterized constructor is added.

6.9 An Example

This section aims at clarifying the previously explained principles of API refactoring, using the framework of a more comprehensive example. To this end, we will choose the example of a time recording system for IT consultants. The consultants can access the system via a web-based interface to feed in their work hours. The values fed into the system form the basis for payroll accounting as well as for generating invoices to customers[5].

5. This example is also used in the chapter on database refactorings. It is described here in its entirety so that one can read both chapters independently from each other.

6.9.1 Our Starting Point

Subsystems in the Time Recording

The subsystems are displayed in Figure 6-6. The consultants access the system via the subsystem *Web*. The accounting department uses the subsystem Report to produce the required print lists and analyses.

Subsystem Web

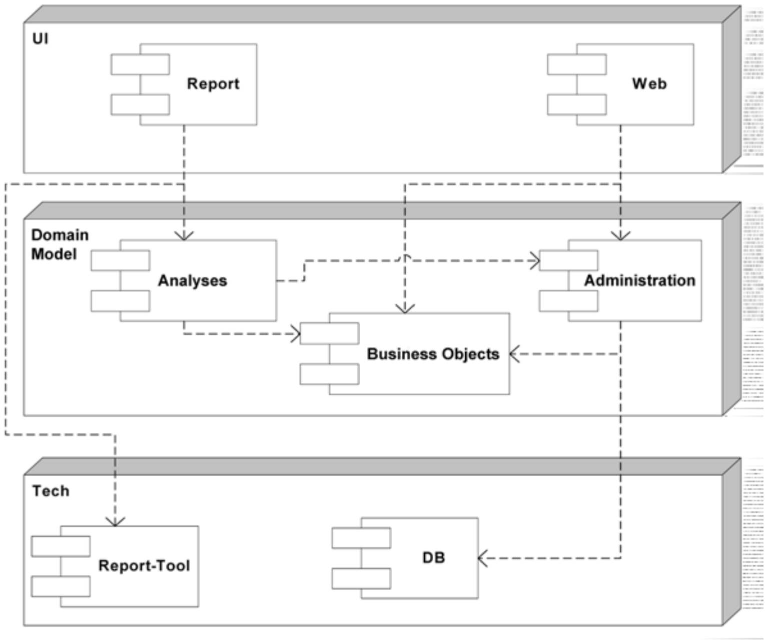
The subsystem *Web* works with the subsystem *Business Objects* which provides concepts such as *Employees* and *Time Entries*. These business objects are saved in the database and reconstructed from the database with the subsystem *Administration*. To this end, a purchased subsystem *DB* is utilized.

Subsystem Report

The subsystem *Report* employs the subsystem *Analysis* in order to carry out all necessary analyses for the print lists (e.g. all hours for each employee and for one project respectively). Of course, the subsystem *Analysis* uses the subsystems *Business Objects* and *Administration* to access the persistent business objects. The print lists are created with the aid of a commercially available report tool (subsystem *Report Tool*).

Of course, the subsystem *Analysis* uses the subsystems *Business Objects* and *Administration* to access the persistent business objects. The print lists are created with the aid of a commercially available report tool (subsystem *Report Tool*).

Fig. 6-6
Subsystems of the Time Recording Example

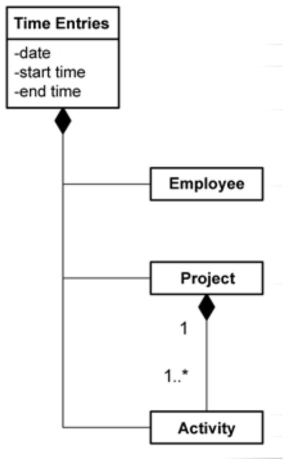


The subsystems are arranged in three non-strict layers: user interface (UI), domain model and technology.

The Business Objects

At its core, the time recording system builds on the business objects displayed in Figure 6-7: *Time Entries* holds a central position. Besides date, start and end time, it contains references to the *Project*, the project's *Activity* and the *Employees*.

Fig. 6-7
Business Objects



The subsystem *Business Objects* has a special role in our example. It is used by many other subsystems, and its API contains all classes of the subsystem. Consequently, we are confronted with the architecture smell 'subsystem API too large' here. In our example, this smell is wanted though. The subsystem *Business Objects* shall provide the vocabulary for the other systems' communication among each other.

Based on this vocabulary, the API of the subsystem *Administration* looks as follows:

API of the Subsystem Administration

```
public interface Administration {
  public void enter (Time Entries z);
  public void cancel (Time Entries z);
  public Time Entries[] getEntry
    (Project p, int year, int month);
  public Time Entries[] getEntries
    (Employee m, Employee);
  public Time Entries[] getEntries
    (Employee m,
     int year, int month, int day);
  ...
}
```

6.9.2 The Reasons for this Refactoring

The modeling of the subsystem *Business Objects* strongly influences the subsystem *Administration*'s API and thus also the interaction of *Analysis* and *Administration*.

Basically, we have to implement the respective low-level functions for most analyses in *Administration*. The business objects are too 'stupid' to allow the subsystem *Analysis* to execute complex functions on them. Theoretically, it is also possible for the subsystem *Analysis* to directly access the database. However, this would also mean that the subsystem *Administration* no longer encapsulates the database, thus making modifications of the database schema more difficult.

Therefore, the subsystem *Business Objects* should be restructured in such a way that it becomes 'smarter' and the API of the subsystem *Administration* does not inflate so strongly.

6.9.3 The Goal of this Refactoring

The New Object Model of the Business Objects

This object model of the subsystem *Business Objects* shall now be modified in such a manner that the model of the core business objects will look as follows: each one of the *Employees* has got a *Month Folder* for each month with a *Calendar Sheet* for every work day. On the *Calendar Sheet* all *Time Entries* are recorded, including start and end time, *Project* and *Activity* in the project (see Figure 6-8).

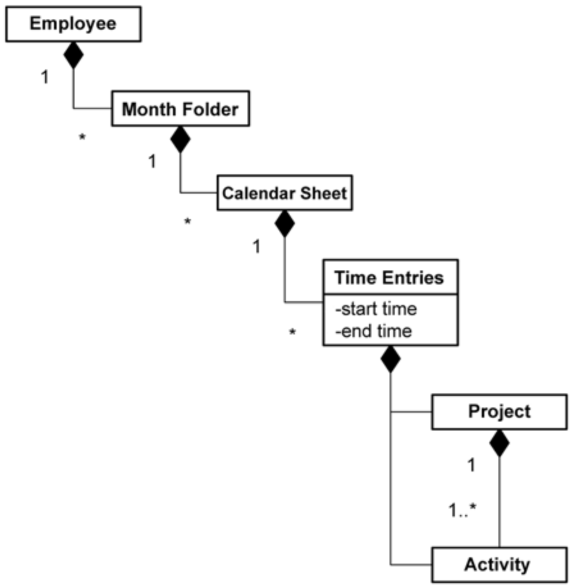


Fig. 6-8
Business Objects after Restructuring

With this restructuring of the subsystem *Business Objects*, we venture deep into the whole system's vocabulary. We should expect a demand for comprehensive restructuring measures of the entire system.

Here, we are going to focus on the refactoring's impact on the API of the *Administration* subsystem. We can imagine the new API as follows:

The New API of the Administration Subsystem

```
public interface Administration {
  public void save (Employee m);
  public Employee getEmployee
    (String name);
  public Employee[] getProjectEntries
    (Project p);
  ...
}
```

In principle, it would be possible to leave the old methods in *Administration* and simply set them to *deprecated*.

Backwards-compatible API of the Subsystem

```
public interface Administration {
  public void save (Employee m);
  public Employee getEmployee
    (String name);
  public Employee[] getProjectEntries
    (Project p);

  /**
   * @deprecated
   */
  public void enter (Time Entry z);

  /**
   * @deprecated
   */
  public void cancel (Time Entry z);

  /**
   * @deprecated
   */
  public Time Entry[] getEntries
    (Project p, int year, int month);

  /**
```

```
         * @deprecated
         */
        public Time Entry[] getEntries
          (Employee m, int year, int month);

        /**
         * @deprecated
         */
        public Time Entry[] getEntries
          (Employee m,
           int year, int month, int day);
        ...
      }
```

Regrettably, this does not solve the problem of how changes of the subsystem *Business Objects*' API should be dealt with. There, not only the interfaces of the classes have changed, but also the relations between the classes themselves were completely rearranged.

6.9.4 Refactoring Procedure: Business Objects Omnipotent

Different Teams for Each Subsystem

We will proceed based on the assumption that the single subsystems in our example are developed by different teams. In the course of a subsystem's refactoring, it is therefore not feasible to reconstruct all this subsystem's dependent subsystems. We must find a way to execute the refactoring step by step, and in such a manner that the subsequent migration will cause as little effort as possible.

Omnipotent Business Objects

The evident refactoring course would be to model the subsystem *Business Objects* in such a fashion that it can be used like the old subsystem. To this end, all classes with their respective methods would be kept and the new methods merely added to them. Naturally, the old methods would be set to *deprecated* to indicate that migration must be directed at the new methods. The outcome would be the class structure that can be seen in Figure 6-9.

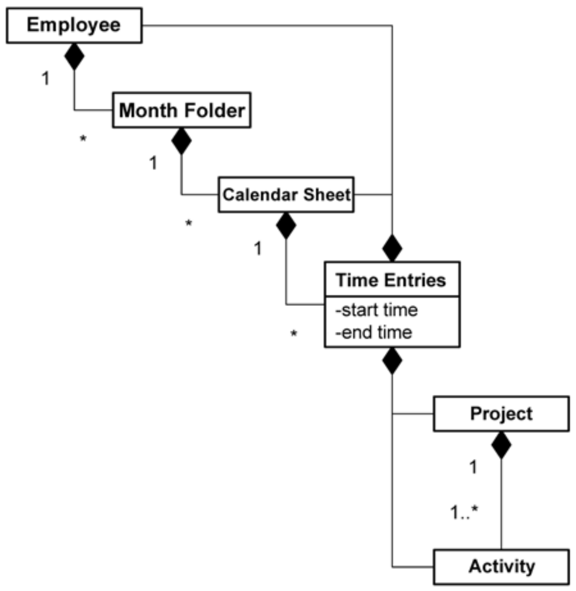


Fig. 6-9
Backwards-compatible Subsystem Business Objects

This example too proves that a large refactoring will often lead to an initial deterioration of the system's structure. Here, we have provoked two extremely bad smells because we created two cyclical relations. For this reason it is pivotal that we finish our refactoring: In the next version of the subsystem *Business Objects*, the cyclical references will be gone, as will be the *deprecated* methods.

Anyhow, the 'deteriorated' version of the subsystem will enable us to adapt the remaining application to the business objects' new structure. Likewise, we will let the API of the subsystem *Administration* 'deteriorate,' so that both the old and the new structure can be processed.

Afterwards, the deprecated methods are removed from the API. We will get the targeted API of the subsystem *Administration*.

New API of the Subsystem Administration

```
public interface Administration {
  public void save (Employee m);
  public Employee getEmployee
    (String name);
  public Employee[] getProjectEntries
    (Project p);
  ...
}
```

6.9.5 Refactoring Procedure: Duplicating Business Objects

Duplication of the Subsystem Business Objects

Another option for a step by step restructuring is duplication of the subsystem *Business Objects*. First, a new subsystem labeled *Business Objects2* is created in the same location as the subsystem *Business Objects*.

In this subsystem *Business Objects*, all classes are then set to *deprecated*. The new subsystem *Business Objects2* contains – besides the new business object classes – a converter that is capable of transferring object structures of the subsystem *Business Objects* into object structures of the subsystem *Business Objects2* and vice versa. This gives us the opportunity of restructuring the remaining parts of the application step by step. Once the application has been completely restructured, both the converter and the subsystem *Business Objects* will be deleted and the subsystem *Business Objects2* renamed into *Business Objects*.

Preconditions for Programming a Converter

To enable programming of such a converter for object structures while keeping the effort to this end acceptable, the object graphs must be rendered convertible without information loss. In our example, the conversion of a time entry of the old model into *Employee* with his or her *Time Entries* is only feasible if access to the database is possible. There, all further time entries for *Employee* must be determined. During reverse conversion, the opposite can easily occur: too much information is provided for one class *Time Entries*. In that case, several *Time Entries* for one *Employee*, based on the old model, must be generated.

This problem can be addressed by creating a 'streamlined' model of the new business objects, i.e. one that allows conversion without information loss (see Figure 6-10).

In this 'streamlined' model, one employee is always assigned one calendar sheet, and each calendar sheet only one time entry. In effect, the time entry's date information has been moved to a class of its own: the class *Calendar Sheet*. Additionally, the uses relation between *Time Entries* and *Employee* has been reversed. Now, the business objects' new structure can relatively simply be changed with the help of a converter.

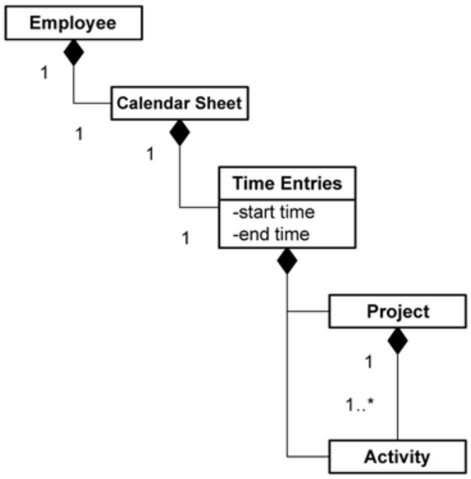


Fig. 6-10 *The 'Streamlined' Model of the New Business Objects*

As soon as we deployed the new object structure, we can install the new API of the *Administration* system:

New API of the Subsystem

```
public interface Administration {
  public void save (Employee m);
  public Employee getEmployee
    (String name);
  public Employee[] getProjectEntries
    (Project p);
  ...
}
```

In the next step, the refactoring of the business objects will be completed. The new class *Month Sheet* and the transition from 1:1 relations to 1:N relations is relatively easily accomplished.

6.9.6 Evaluation of Both Approaches

Option 1

The application of the first refactoring option did not require any particular amount of creativity. It did have the disadvantage of degenerating the system structure though. In a very large system, this can have significant negative consequences. Therefore, this kind of 'pollution' must be eradicated as soon as possible.

Option 2

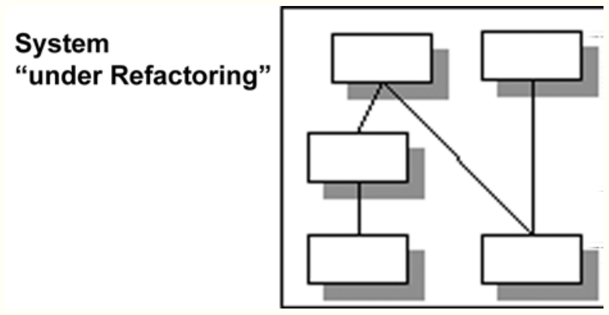
The second option created far less 'pollution' of the system due to use of the converter. Especially the dependent system's APIs did not inflate. The duplication of the subsystem *Business Objects* definitely constitutes a smell (code duplication), but both units are strictly kept separate. Of course, it is important to finally delete the old version of the subsystem as soon as possible.

The second option seems to be more appealing for migration, but unfortunately it is not universally valid. If classes from the subsystem *Business Objects* had inherited from other subsystems, writing a universally applicable converter would have become impossible. Converters will work well if it is guaranteed that no subclasses exist. This is the case if at least all dependent subsystems are available for analysis, or if constructive subclasses have been excluded, for example through declaring the classes of one's subsystem *final*[6].

Excursion: Black Box Refactoring

A contribution by Jens Uwe Pipka
(jens-uwe.pipka@daedalos.com, Daedalos Consulting)

One often-made implicit assumption during a refactoring process is that the entire code basis of a software system is at the programmers' disposal. If this is the case, all relations and uses of a subsystem to be refactored are known. They can be analyzed and adapted accordingly. This procedure constitutes a so-called *White Box Refactoring*, because the changes are made to the whole system. After the whole system has been adapted, the refactoring is complete. No further steps are required. This scenario is shown in the following figure:

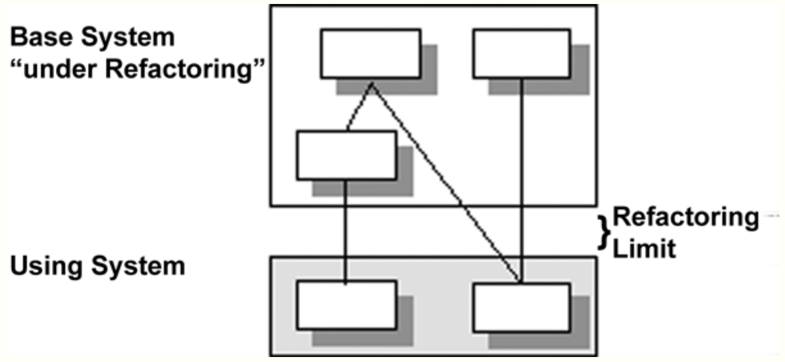


White Box Refactoring

6. See comments regarding testability in chapter 6.8.

We are facing a more difficult situation if a refactored system has a number of potential clients that are not familiar, as it is e.g. the case when object-oriented libraries or a framework are applied. In such a situation, different systems use the common code basis. Therefore, we speak of a base system. An external subsystem that uses and specifies this base system is called the using system (see next figure).

If a refactoring is carried out in the base system, it is in most cases not feasible to identify all potential effects on the using systems, because multiple couplings between systems are possible. Thus, not all options can be fully taken into account during refactoring processes concerning the base system: A so-called *Black Box Refactoring* takes place, since there is no guarantee that each combination of the refactored base system and the using system will display the same semantic behavior in the future.



Black Box Refactoring

The complexity of identifying possible effects on a using system during the refactoring process is caused by the fact that object-oriented systems in principle allow two orthogonal ways of base system usage and specialization:

1. Call of those methods that are accessible via the interface of a class; also called *Black Box Usage.*
2. Specialization of a class through inheritance; also known as *White Box Usage.*

In the first-mentioned case, a refactoring in the base system can be executed with relatively few problems, because the internal structure is to a large degree encapsulated and changes are thus transparent. Only changes to the public interface of a class are critical, such as e.g. the removal or addition of a method. However, the use of refactoring tags enables proactive communication of these modifications and even – as described in chapter 6.4 – mostly automated adaptation in the using system.

The second case, however, has yet another aspect that must be considered when refactoring the base system: changes in the system structure can have additional semantic effects on the using system, because the latter is implemented based on a specific syntactical structure. Inheritance causes the structure of the code basis to lie open and thus forms the basis for developing the using system. As a consequence, changes to this structure can affect the conditions present at implementation and therefore directly affect the program's semantics. Modifications of the program's semantics cannot be completely handled with refactoring tags alone, because refactoring tags are only meant to deal with changes of a class's interface. The complete, implicit interaction of base system and using system is not covered by the capabilities of refactoring tags.

Thus it is necessary to establish some additional procedures to ensure that a refactoring will not alter the semantics of the whole system: To this end, information for changes to the public interface of a class must be checked – not only in regard to its callers, but also in the context of the inheritance hierarchy, while keeping the using system in mind. Possible conflict situations have already been addressed in the section about compatibility classes in chapter 6.3. It should not go unmentioned though that conflicts in the inheritance hierarchy cannot be sufficiently solved by 'mechanically' changing all references, because such an action may create significant semantic conflicts.

In effect, not only the isolated migration efforts in the base system must be considered, but moreover their impact as well as the required migration steps in the using system. In the following, we will discuss this problem and illustrate it using two examples. We will examine in how far usage and specialization must be distinguished in the refactoring process and whether a semantic conflict analysis is required or not.

Semantic Conflict Scenarios through Black Box Refactoring

The difficulties encountered in Black Box Refactoring can be demonstrated with a simple example. In our example, the base system implements an accounting system that generates invoices when one or more products are purchased. One component of the accounting system is the discount module: This module allows calculation of a certain discount based on a product price. In our example, the discount functionality is implemented as follows:

```
public class DiscountOffer {
  int individualDiscount;
  public DiscountOffer() {
    individualDiscount = this.initialDiscount();
  }
  public int initialDiscount() {
    return 0;
  }
  public void setDiscountUser(Customer customer) {
    this.individualDiscount =
      this.initialDiscount() + 0;
  }
  public int getDiscount(int price, Customer customer) {
    this.setDiscountUser(customer);
    return price * individualDiscount / 100;
  }
  public int getDiscount(int price) {
    return price * individualDiscount / 100;
  }
}
```

The customer discount is composed of an initial discount granted to all buyers, whether they are registered customers or not. This percentage is determined by the method *initialDiscount()*. Persons who are already registered customers can be given an additional discount. This percentage must be determined with the method *setDiscountUser(Customer)*. Within the generic functionality of the base system, both values are set to zero percent.

This base system is distributed to various users, who can define different specifications for the generic discount function provided by the base system, depending on the kind of discount offer they wish to create. The concrete discount offer is implemented with the aid of the inheritance mechanism in the using system.

A company wishes to give all its buyers an additional discount of five percent as part of their summer discount offer. Regular customers (who are already registered) are not supposed to get this extra discount. Accordingly, only the initial discount must be adapted and set to five percent. This requirement will be implemented in the using system:

```
public class SummerDiscountOffer extends DiscountOffer {
  public int initialDiscount() {
    return 5;
  }
}
```

A test of the using system shows that the requirement is met as desired. The call *summerDiscountOffer.getDiscount(100)* returns the result that a discount of five Euro has been given based on a price of 100 Euro.

Independent from the tests, the developers of the base system continue to work on the accounting system's implementation. To prepare the implementation and usage of the class *DiscountOffer* for extensions, they move the predefined values into two new instance variables: *initialDiscount* and *customerDiscount*. They apply the *Inline Method* refactoring in the constructor to achieve a more efficient and clearly-structured implementation. Since the public interface shall not deviate from that of the earlier version, the method *initialDiscount()* is kept and adapted to the new structure. For better legibility, only changes to the class *DiscountOffer* are listed below.

```
public class DiscountOffer {
  ...
  int initialDiscount = 0;
  int customerDiscount = 0;
  public DiscountOffer() {
    individualDiscount = initialDiscount;
      //Inline Method
  }
  public int initialDiscount() {
    return initialDiscount;
  }
  public void setDiscountUser(Customer customer) {
    this.individualDiscount =
      initialDiscount + customerDiscount;
  }
}
```

The behavior of the class *DiscountOffer* in the base system is identical to the class's behavior in the previous version, i.e. all tests return the same results as before. The new version of the base system is now ready for distribution.

The first step of integration with the system that implements the summer discount offer initially appears to be successful, since no syntactical errors occur and the whole system can be compiled without problems. Nevertheless, the program semantics have changed, so that the requirement is no longer met correctly. The call *summerDiscountOffer.getDiscount(100)* now generates the result that no discount will be given.

The reason for that error is that – due to the Inline Method refactoring – the specialization of the method *initialDiscount()* is no longer considered in the system, thus leading to a faulty initialization of instances of the class *SummerDiscountOffer* and generating the incorrect program behavior.

This altered behavior is caused by the fact that the call graph has been changed during refactoring: the method *initialDiscount()* which was originally called in the constructor is no longer called on. Therefore, the specialization in the class *SummerDiscountOffer* is disregarded. This conflict will only be recognized when the base system and the using system are analyzed together. As part of the refactoring process on the base system level, this change poses no problem.

Even the application of refactoring tags will not help in this case: the public interface of the class *DiscountOffer* has stayed the same. In effect, no refactoring tags are defined and thus no changes are triggered in the using system.

This example clearly demonstrates that an exclusively proactive approach for describing the impact of a refactoring in the base system is neither sufficient for the using systems, nor will an analysis and test of the base system alone suffice. Only a comprehensive function test of the whole system will bring certainty that an integration of basic and using system was truly successful. But even this approach will only be partially helpful for error analysis: in order to analyze the cause of the system's semantic misbehavior, comprehensive knowledge of the entire system is required. For that reason, integration of an automated, semantic analysis in the refactoring process is advisable to offer programmers additional support during development of a distributed system.

Gray Areas in Black Box Refactoring

Besides direct changes to the call graphs, especially so-called 'Big Refactorings' will lead to constellations that literally provoke semantic conflicts. The combination of different changes that are made in the course of a refactoring will often lead into 'gray areas' that cannot be resolved in the base system and that will only become visible in combination with the using system.

Particularly modifications that are meant to prepare the ground for extending the base system are often dangerous. To illustrate such a situation, we will now return to our accounting system example. Based on the now familiar implementation, first a new discount offer for the coming Christmas season is implemented in the user system, granting a five-percent discount to all buyers as well as an extra five percent to regular customers.

```
public class ChristmasDiscountOffer
  extends DiscountOffer {
  public ChristmasDiscountOffer() {
    individualDiscount = initialDiscount = 5;
  }
  public void setDiscountUser(Customer customer) {
    customerDiscount = 5;
    super.setDiscountUser(customer);
  }
}
```

In the base system a method *getDiscount(price, customer)* does already exist to calculate a customer's discount in a single step. For a price of 100 Euro a customer will accordingly receive a discount of ten Euro.

The introduction of the class *Person* as superclass of *Customer* shall prepare the base system for future extensions. This plan is also reflected in the implementation of the class *DiscountOffer*: the interface for the method *getDiscount(price, customer)* is altered and the parameter of the type *Customer* is changed to a parameter of the superclass's *Person* type. No change takes place for the callers, since the generalization of a parameter for using this method is transparent.

```
  public class Person { … }
  public class Customer extends Person { … }
  public class DiscountOffer {
    ...
    public void setDiscountUser(Person person) {
      this.individualDiscount =
        initialDiscount + customerDiscount;
    }
    public int getDiscount(int price, Person person) {
      this.setDiscountUser(person);
      return price * individualDiscount / 100;
    }
  }
```

This change is in itself uncritical, but in connection with the user system it results in a changed discount calculation because the specialization for *setDiscountUser(customer)* from the class *ChristmasDiscountOffer* will no longer be called on. Thus customers will receive the wrong discount amount of five percent instead of the offered ten percent.

This example clarifies that gray areas which cannot always be eliminated inherently exist in refactoring processes. The application of refactoring tags can help in this case: the *Past* tag is implemented for the method *set-DiscountUser(Person)* here. Unfortunately, this approach also has a disadvantage. All callers of this method must execute an extra type cast of the object *Customer* after *Person*. This contradicts the original idea of utilizing the options object-orientation has to offer and of keeping modifications of the base system transparent for all callers through generalization of this method, while also creating as little as possible (ideally no) demand for migration within the using systems.

Support of the Black Box Refactoring through Semantic Conflict Analysis

Both examples underline that distributed development in Black Box Refactoring harbors the danger of creating semantic conflicts that stem from the combination of base system and using system. These conflicts can hardly be avoided by documenting the changes alone.

Here we must differentiate between two scenarios: In the first scenario, we are dealing with changes of a class's interface that have to be communicated to the using system. This can be done via refactoring tags. Changes to certain categories of classes, their effects on the using system as well as possible measures for avoiding or removing conflicts have been discussed in detail in chapter 6.5. In this respect, this scenario is clearly structured and can be handled.

Things are different for the second scenario, where changes occur in the inheritance interface itself: the using system must be informed of these changes. The number of potentially possible conflicts exceeds that of those possible in the public interface: Adding to them are the effects of changes to the internal structure and the implementation of the base system itself. For example, changes to the call graph can lead to errors in the combination of base system and using system. The same is true for generalizations or specializations on the class and method level that are not entirely alleviated by classic documentation concepts for a class's interface. Their effects will vary strongly, depending on which specializations are implemented in the using systems.

Since the number of possible effects of changes in the inheritance interface is fairly large – even for simple refactorings – a reactive innocuousness check is required besides a documentation of the changes: On the one hand, this can be achieved by running an as comprehensive function test as possible of the using system; on the other hand, a semantic error analysis for certain error classes following the integration of the modified base system will ensure that the system's behavior has remained semantically unaffected.

Semantic Error Analysis in the Refactoring Context

In order to recognize the semantic conflict situations discussed here, information about inheritance structures as well as the call graph are required – information also needed by refactoring tools to enable a mostly automated execution of refactorings. One peculiarity of semantic error analysis is that changes between the status quo prior to a refactoring and the status quo after a refactoring do matter. Without this differentiation, it is not possible to identify the conflict situation depicted here.

This prerequisite proves critical particularly for the execution of a refactoring in a base system without direct relationships to possible using systems: the data needed for recognizing the conflict must be obtained from both systems. Therefore, the call graph of the system plays a pivotal role that goes beyond the scope of the actual refactoring. In our first example, the conflict originated from the fact that a call was removed from the base system and thus a specialization from the using system was no longer observed.

Yet another obstacle must be taken into account: It does not suffice to consider the current combination of the system consisting of the refactored base system and the using system. Moreover, the condition created by their combination must be compared for the base system as well as for the using system before and after refactoring. These prerequisites make it clear that recognition of a conflict based on a concrete misbehavior can only take place on the using system's side. In addition, both variants of the base system must be available to allow a comparison.

Stepping over the system boundary between basic and using system has yet another consequence: the release and thus integration of the base system is not restricted to a single, atomic refactoring. To enable an efficient development cycle on the one hand and a rewarding integration scenario on the other, a new version of the base system must contain a series of changes. Here, we can distinguish between an exclusive refactoring, exclusive continuous development and a blend of the two. All variations have in common that modifications of the source code will bring about a large number of changes in the program structure.

This distinction does hardly matter for semantic conflict analysis, because occurrences of the described conflict situations are thinkable for all variations. However, evaluations showed that the number of possible semantic conflicts in a refactoring cycle is particularly high, because in a refactoring cycle most work takes place in existing classes and methods which already supply their functionality to using systems. The realization of new functionalities will rather lead to the implementation of new classes and methods that are not yet in use – a fact that reduces the danger of semantic conflicts. This confirms that semantic conflict analysis, especially for large refactorings, is an indispensable means of support for controlling and limiting the number of a refactoring's possible consequences.

Further evaluations prove that a separation between pure refactoring cycles on the one hand and development cycles on the other makes sense when developing a universal base system with a series of possible clients. The integration of base and using system can be treated in different ways in the base system, depending on the type of change: Suitable integration steps (depending on the type of change), for example the execution of a semantic conflict analysis exclusively in the course of the refactoring cycle, can lower the risk of running headfirst into an unrecognized misbehavior of the whole system without losing the option of quickly integrating new functionalities.

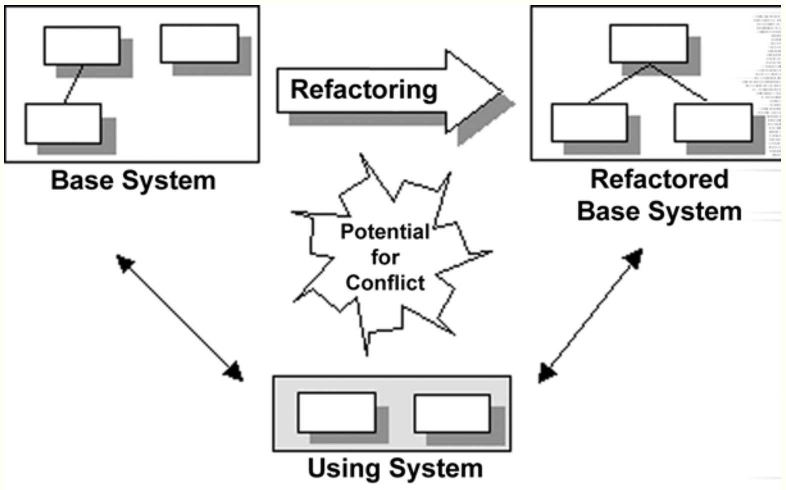
Tool-based Recognition of Semantic Conflicts

Changes in the call graph of the base system can hardly be monitored manually. Especially when the execution of a complete refactoring or development cycle is examined, it is impossible to document all changes in the form of additional tags. This is also the case for necessary evaluations that must be carried out in the using system during integration of the refactored base system.

Therefore, it is necessary to automatically record the data, for example by extracting the relevant data from the source code and transferring it to a suitable meta model or by using the existing meta information offered by a refactoring tool. In the next step, the actual states before and after a refactoring must be evaluated to identify possible conflict situations.

This procedure can algorithmically be realized as follows:

1. Determination of the potential for conflicts as the amount of all changes of the base system before and after refactoring or, respectively, the development cycle.
2. Analysis of the dependencies between the potential for conflicts and the using system.
3. Identification of concrete conflicts based on the identified dependencies while applying adequate rules for conflict recognition.



Potential for Conflicts through a Refactoring of the Base System

In this context, the definition of a rule depends on the specific error scenario. For our first example that does disregard method specialization, a suitable rule looks like this:

1. Determine the potential for conflicts consisting of all method calls that have been removed from inside the modified basic module.
2. Find all methods that have been overwritten by the basic module in the using system and therefore have been specialized.
3. Identify the potential for conflicts as the amount of all methods found in step 2 that are also part of the potential for conflicts determined in step 1.

Analogously, rules for the recognition of further conflicts can be created and integrated in the recognition algorithm. Here, most rules operate based on a straightforward number of basic operations and structural information provided by the software system. If these are available, the definition of new recognition rules is in most cases feasible with calculable effort.

This principle of conflict recognition is realized in combination with additional semantic analyses in JaMB, the Java Migration Browser. JaMB is a tool for the detection of semantic conflicts that occur during the further development and migration of object-oriented systems. It obtains its required information directly from the Java byte code and stores it in a suitable meta model. More information about JaMB can be found in [Pipka & Mezini 00].

Another approach to recognizing conflicts and making them transparent during the development process is the backward compatibility tester. Here, the interfaces of the contained classes are analyzed and incompatible changes are pointed out. However, this analysis is exclusively syntax-based. More information about this tool can be found on IBM's Alphaworks website (http://www.alphaworks.ibm.com).

6.10 References

[Bloch 01] Joshua Bloch: *Effective Java*. Addison-Wesley, 2001.

Bloch argues that inheritance must be planned. If inheritance from a class has not explicitly been provided for, it should rather be prevented (for example through setting the class to final*).*

In this book you will find – besides a number of very useful tips for Java programming – an instruction for realizing typesafe and expandable enumeration types (typesafe enums) in Java.

[Fowler 99] M. Fowler: *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts, Addison-Wesley, 1999.

Not only does Fowler describe basic refactorings here, he also introduces the distinction between public and published interfaces.

[Gamma & Beck 03] Erich Gamma, Kent Beck: *Contributing to Eclipse – Principles, Patterns, Plug-Ins*. Addison-Wesley, Eclipse Series, 2003.

In this standard work about Plug-in development with Eclipse the authors depict very diverse mechanisms realized in the Eclipse framework. Among them are patterns that simplify evolution between subsystems, such as, for example, the Adaptable mechanism.

[Gosling et al. 97] J. Gosling, B. Joy, G. Steele: *Java: Die Sprachspezifikation*. Bonn, Germany: Addison-Wesley, 1997.

A language specification for Java.

[Havenstein 03] Andreas Havenstein: *Werkzeuge für die Migrationsunterstützung von Anwendungen auf neue Rahmenwerksversionen*. Diploma thesis, Software Engineering Group, Dept. of Informatics, University of Hamburg, Germany, 2003.

Here, tools for the handling of refactoring tags as well as implementation methods for these tools are presented. Also discusses how much help converters can provide for migration.

[Havenstein & Roock 02] Andreas Havenstein, Stefan Roock: *Refactoring Tags for Automatic Refactoring of Framework*. In: Proceedings of Extreme Programming Conference 2002, Villasimius, Cagliari, Italy, 2002

This article is the first one to introduce the concept of refactoring tags.

[JavaSoft 00] JavaSoft: *How and When to Deprecate APIs.*
Part of the Java2 Documentation, 2000.

Describes work with the deprecated *tag.*

[Lippert et al. 01] Martin Lippert, Stefan Roock, Henning Wolf, Heinz Züllighoven: *JWAM and XP – Using XP for Framework Development.*
In: [Succi & Marcesi 01]. Pp. 103-117.

Illustrates various experiences in the use of XP techniques (such as refactoring) for Frameworks.

[Meyer 92] B. Meyer: *Eiffel: The Language*. Englewood Cliffs: Prentice-Hall, 1992.

Meyer introduces the reader to the language Eiffel*, which offers an option for marking classes and methods as deprecated with the keyword* obsolete. *Just like with using the* deprecated *tag in Java, the compiler will generate warnings for obsolete elements.*

[Pipka & Mezini 00] Jens Uwe Pipka, Mira Mezini: *Weiterentwicklung objektorientierter Softwaresysteme: Risiken und deren Vermeidung.*
In: Informatik 2000, Springer-Verlag, 2000.

This work provides an overview of important migration conflicts in literature and practice, which are further scrutinized using Java. The focus is on conflict scenarios that can emerge during the integration of more complex subsystem versions. To support the integration process, a tool-based solution for the recognition of such conflicts is introduced as an example: the Java Migration Browser JaMB.

[Rivières 01] Jim des Rivières: *Evolving Java-based APIs*. http://www-eclipse.org/eclipse/development/java-api-evolution.html. 2001.

Depicts possible API changes and their compatibilities. The focus is only on binary compatibility, not on source code compatibility though.

[Roock 01] Stefan Roock: *eXtreme Frameworking – How to Aim Applications at Evolving Frameworks*. In: [Succi & Marcesi 01]. Pp. 71-82.

Analyses of problems related to API changes and some ideas for solutions.

[Roock 04] Stefan Roock: *Unterstützung für die Evolution und Migration objektorientierter Systeme*. Ph.D. thesis Software, Engineering Group, Dept. of Informatics, University of Hamburg, Germany. To be completed in 2004.

In this thesis, the conceptual and basic principles of refactoring tags (called migration tags *here) are developed and further elaborated on.*

[Succi & Marchesi 01] G. Succi, M. Marchesi (Hrsg.): *Extreme Programming Examined*. Reading, Massachusetts, Addison-Wesley, 2001.

Conference publication of the XP-2000 Conference.

[Szyperski 97] C. Szyperski: *Component Software*. Harlow, England, Addison-Wesley. 1997.

Standard work about components that also offers insights on some of the aspects discussed here.

7 Tool-based Detection and Avoidance of Architecture Smells

By Walter Bischofberger and Henning Wolf

Introduction

Architecture smells are difficult to detect 'manually,' since to this end information from the entire source code must be collected and condensed. As we already pointed out in chapter 3, IDEs cannot be used for architecture analyses because they visualize relations between classes and packages. For an analysis of architectural aspects, developers must work on the levels of class, package and subsystem relations and not on the method call level.

In this chapter, we will demonstrate how architecture smells can be found with tool support and how large refactorings can be closely monitored. Our tool of choice will be *Sotograph*, a product of Software-Tomography GmbH. We decided to focus on a commercially available tool because as far as we know there are currently no other tools in existence that support both architecture analysis and visualization for object-oriented systems in a similarly comprehensive manner.

7.1 Specifications of an Analysis Tool

Levels of Abstraction

Chapter 3 discusses that architecture smells can be found on different abstraction levels (class, package, subsystem and architecture). In consequence, tools for the detection of architecture smells must be able to handle these or related abstraction levels. Depending on the abstraction mechanisms offered by the supported programming language, more or less high-level abstraction levels must be user-definable.

Types of Analyses

Based on the abstraction levels at the developers' disposal, the following analyses can be executed:

- *Architecture Analysis*, i.e. an analysis of how well the source code observes the restrictions specified in the architecture model (who is allowed to use whom, and who is allowed to inherit from whom), and where these restrictions are not observed.

- *Cycle Analysis*, to analyze the cyclic relations between artefacts. More precisely, this means the search for classes, packages and subsystem groups that are strongly coupled in cyclic relations.
- *Metrics-based Analysis*, in order to identify potential architectural problems using metrics.

Interpretation of Analysis Results

All three analysis types deliver distinct indications of a multitude of problems that will range from those that can be neglected to the ones that are quite serious. To assess the seriousness of a problem as efficiently as possible, it is important that the tool provides explanations, or at least the basic data of its analysis. In this context, visualization – for example with class diagrams or package graphs – is a major issue. Particularly since architectural problems, after all, become manifest in unexpected and unwanted cooperation of artefact groups, graphs can make it much easier to understand such cooperation structures.

It is very laborious to analyze a version of a software system in detail. In most cases, the members of a project team who possess the required knowledge to conduct such an analysis will be highly in demand. This is another reason why it is unrealistic to continuously monitor a software system's quality without adequate tool support, even though continuous monitoring would be the most useful approach in such a situation.

Continuous Monitoring

Tools that are suited for the continuous monitoring of large software systems will only issue information that has changed since the last analysis was conducted and will also filter it based on its relevance. A metrics tool, for instance, will only show those metrics values that were already bad a week ago and have gotten either worse or better since. An architecture tester will merely display new architecture violations. In an ideal case, the weekly time exposure for interpreting the obtained data can be reduced to under half an hour.

Ad-hoc Analyses

In big projects, frequently questions will come up that cannot be answered with the aid of pre-defined analyses. Therefore, a tool for large-scale software analysis should offer users an easy-to-learn mechanism to enable them to formulate new metrics and ad-hoc queries.

Support of the Refactoring Process

Depending on the kind of restructuring that is needed, the execution of a large refactoring can take place over a rather long period. Without careful organizing it will soon become difficult to determine what parts of the refactoring have already been carried out, and to what extent the architecture's status quo already approximates the targeted architecture. The latter question can be answered precisely with the right tool. Furthermore, a tool can provide a lot of useful information about a refactoring's present stage if it has monitored the development of a software system over a longer period. For example, it can point out what classes have been modified in which subsystems, and which methods of these classes have been newly created, deleted or changed.

Analysis of Distributed Systems

Typically, modern software systems consist of distributed systems which are tied into different processes on various machines and managed at runtime by application servers, for instance.

If one considers only the static relations of such a system exclusively from a language point of view, one will see a quantity of isolated parts, e.g. Enterprise Java Beans or .Net components. Apparently these are not cooperating, because they don't use each other directly, but solely communicate via the application server. Architectural analyses of such systems will only make sense if the tool for the analysis of distributed systems can handle various interprocess communication mechanisms and component models. Then distributed systems, including their relations beyond processes, can be analyzed and visualized as a whole.

7.2 Architecture Analysis with Sotograph

In this chapter, we will elaborate on how to use the *Sotograph* as a tool for analysis and scrutinize its role as an analyzing software system, that is, we will apply Sotograph to versions 0.90, 0.95 and 0.96 of Sotograph's source code. We will use relatively old versions because from version 0.96 onward Sotograph has been used to further its own development. Thus the newer versions will deliver less interesting results for architecture testing.

The Underlying Concept

The underlying concept of the Sotograph is derived from computerized tomography as it is applied in medicine, i.e. it extracts – just like a computer tomography – as much information as possible about the system it examines, before it commences actual analysis. For software systems, its analyses comprise byte and source code, which will yield information about references, artefacts on the method level, fields, classes, packages and relations between these elements. Sotograph stores this information in a relational database and then proceeds to make available a number of closely integrated tools in order to analyze and visualize the gathered information. At present, Sotograph possesses analyzers for Java, C++ and C.

In addition, Sotograph can manage information that encompasses a series of a software system's versions in a database, thus enabling analysis of a system's changes over time.

Moreover, information about the (desired) architecture model can be fed into Sotograph's database. This extra information will serve as a basis for analyzing the system's architecture.

Abstraction Support

With classes, files, packages, subsystems and tier architectures, Sotograph supports five abstraction levels on which analyses can be conducted. Whereas in object-oriented languages classes are surveyed on the lowest abstraction level, for procedural languages – such as C – it is important to have the abstraction level 'file' at hand. Moreover, files (and directories) constitute the entities that physically structure the code (e.g. several classes in one file).

For systems implemented in Java, the packages defined in the source code are used directly as an abstraction level. Since only very few programming languages offer an abstraction level that is equivalent to the Java package, Sotograph will aggregate all files located in the same directory in a package of the same name as the directory for these languages.

Packages can be combined to form subsystems using a subsystem description language. For each subsystem, it can be defined where its explicit API is located (if applicable). For example, a subsystem should only be used via certain interfaces that are placed either in a subpackage 'interface' or directly in the subsystem's root directory. Other references from outside this subsystem constitute an architecture violation. The subsystems form the abstraction level on which the system's architecture can be reviewed. For small and medium-sized software systems, one subsystem model will normally suffice. For large software systems, often several subsystem models are needed for the modeling of architectural levels that are relevant for analyses.

Building on one subsystem model each, tier architectures can be specified. These tier architectures serve the purpose of limiting the amount of relations that are permissible between subsystems.

Identifiable Architecture Smells

Architecture testing with Sotograph allows identification of the following architecture smells:

- Subsystem-API bypassed (3.5.5)
- Upward references between layers (3.6.2)
- Strict layering violated (3.5.3)
- Relations between product-specific subsystems of product line architecture (briefly mentioned in 3.5.5)

The following graphic (see Figure 7-1) exemplifies these architecture smells in a 3-tier architecture.

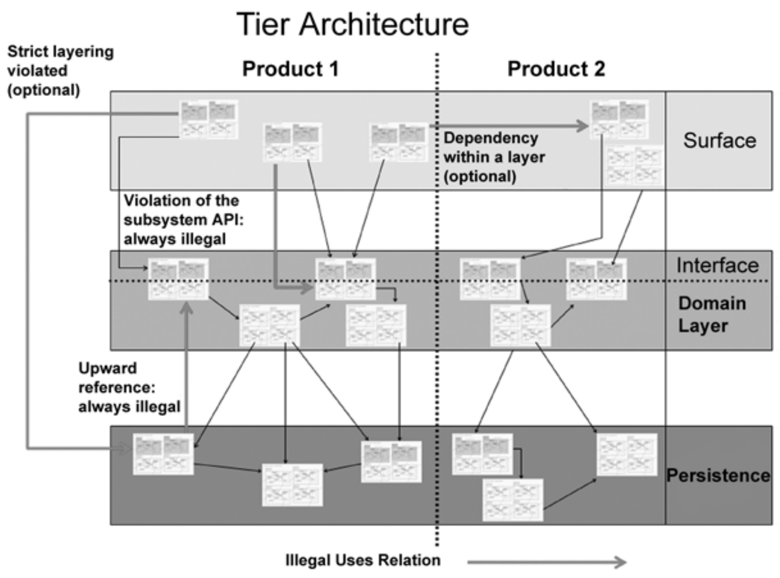

Tier Architecture
Strict layering violated (optional)
Product 1
Product 2
Dependency within a layer (optional)
Surface
Violation of the subsystem API: always illegal
Interface
Domain Layer
Upward reference: always illegal
Persistence
Illegal Uses Relation

Fig. 7-1
Architecture Violations

Definition of a Subsystem Model

Before we can begin with the actual architecture analysis, we must first define at least one subsystem model and one architecture model.

Sotograph defines subsystem models with a simple subsystem description language. Basically, there are two ways of describing a subsystem:

- There are rules that specify how package trees will be aggregated in subsystems. The following rule, for instance, leads to the generation of one subsystem for each of the three package trees.

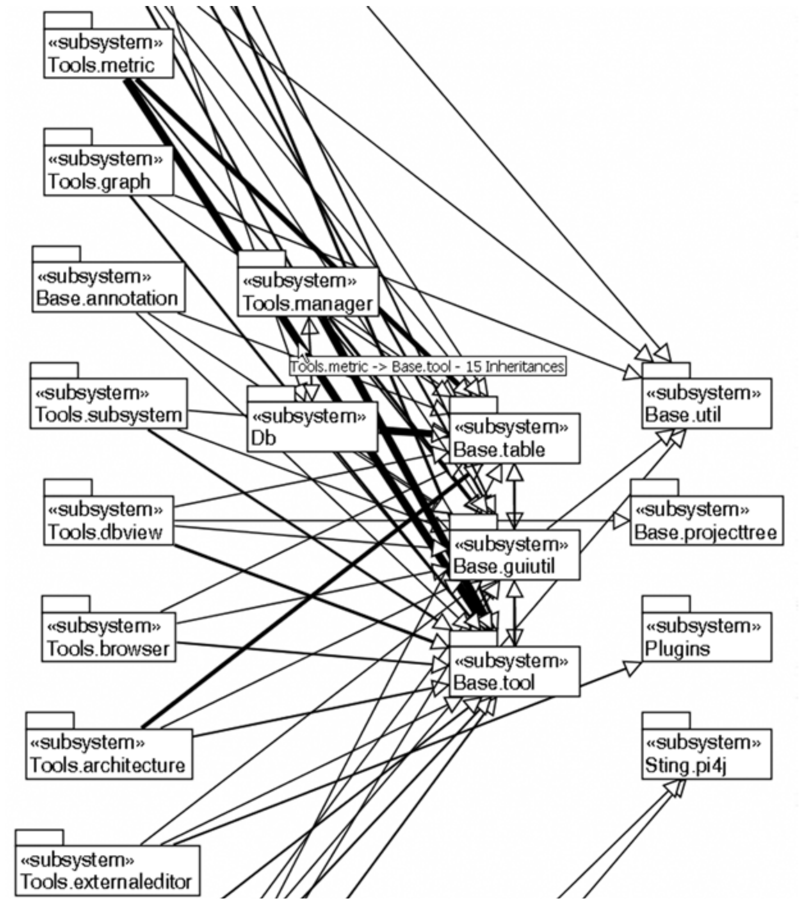
```
RuleBasedSubsystem Public {
    InterfacePath "";
    Packages "com.sotogra.'(util|guiutil|plugins)'";
}
```

  - Using a regular expression, the packages statement defines the root packages of the package trees that are combined into subsystems.
  - The name of a generated subsystem will be derived from the rule's name and the name of the package tree's root package, e.g. *Public.util*.
  - The interface path statement defines the path from the package tree's root to the package that contains the subsystem's API classes. In our example this is the root package.

- Through counting the packages contained in it. This sort of definition is mainly used in generated subsystem models or in such exceptional cases where a subsystem cannot be defined by rules.

Once a subsystem model has been defined and set as an effective subsystem layer, the subsystems defined in this layer can be employed analog to classes, files or packages in all tools offered by Sotograph. For instance, all inheritance relations between subsystems can be visualized (see Figure 7-2). The thicker the lines in the image, the greater the number of inheritance relations existing between the subsystems.

Fig. 7-2
A Section of a Subsystem Graph


«subsystem» Tools.metric
«subsystem» Tools.graph
«subsystem» Base.annotation
«subsystem» Tools.manager
«subsystem» Tools.subsystem
«subsystem» Db
Tools.metric -> Base.tool - 15 Inheritances
«subsystem» Base.util
«subsystem» Base.table
«subsystem» Tools.dbview
«subsystem» Base.projecttree
«subsystem» Base.guiutil
«subsystem» Tools.browser
«subsystem» Plugins
«subsystem» Base.tool
«subsystem» Tools.architecture
«subsystem» Sting.pi4j
«subsystem» Tools.externaleditor

Definition of an Architecture Model

Sotograph defines tier architectures with a simple architecture description language. In this language, one

- defines the underlying subsystem model;
- assigns subsystems or packages to single layers;

- defines if the subsystems or packages of a layer are allowed to mutually use each other, and whether the layer model is subject to strict interpretation or not.

For comprehensive software systems, developers will often define several architecture models that focus on various aspects as well as separate architectures for large or complex structured subsystems.

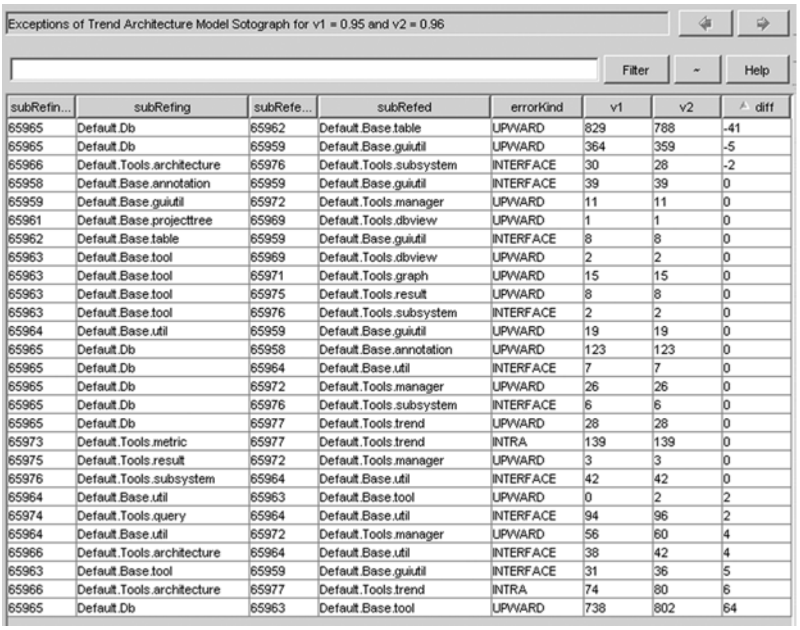
The following is an excerpt from Sotograph's architecture model description:

```
ArchitectureModel Overview {
  Uses Default; // used subsystem model
  ArchitectureLayer Manager {
    // layer may use all deeper layers
    InterLayerUsage = True;
    // the layer's subsystems may
    // mutually use each other
    IntraLayerUsage = True;
    // subsystems contained in the layer
    Subsystem Tools.manager;
    Subsystem Access;
  }
  ArchitectureLayer ToolsAndServices {
    InterLayerUsage = True;
    // the layer's subsystems may
    // not mutually use each other
    IntraLayerUsage = False;
    // selection of subsystems belonging to the layer
    // with a regular expression
    Subsystems "Tools.([^m]|met).*";
  }
  ArchitectureLayer ToolInfrastructure {
    InterLayerUsage = True;
    IntraLayerUsage = True;
    Subsystem Base.annotation;
    Subsystem Base.dbupdate;
    Subsystem Base.migration;
  }
  ArchitectureLayer Frameworks {
```

Analysis of Architecture Violations

Once an architecture model has been defined and evaluated, the illegal relations that were found can be examined. First, we will take a look at the list of illegal relations between subsystems (see Figure 7-3). The table shows the number of architectural deviations between subsystem pairs, sorted based on architecture smells. The last three columns contain the values for Sotograph versions 0.95 and 0.96 as well as changes that occurred between these versions[1].

Fig. 7-3
Overview: Architecture



Exceptions of Trend Architecture Model Sotograph for v1 = 0.95 and v2 = 0.96

subRefin...	subRefing	subRefe...	subRefed	errorKind	v1	v2	diff
65965	Default.Db	65962	Default.Base.table	UPWARD	829	788	-41
65965	Default.Db	65959	Default.Base.guiutil	UPWARD	364	359	-5
65966	Default.Tools.architecture	65976	Default.Tools.subsystem	INTERFACE	30	28	-2
65958	Default.Base.annotation	65959	Default.Base.guiutil	INTERFACE	39	39	0
65959	Default.Base.guiutil	65972	Default.Tools.manager	UPWARD	11	11	0
65961	Default.Base.projecttree	65969	Default.Tools.dbview	UPWARD	1	1	0
65962	Default.Base.table	65959	Default.Base.guiutil	INTERFACE	8	8	0
65963	Default.Base.tool	65969	Default.Tools.dbview	UPWARD	2	2	0
65963	Default.Base.tool	65971	Default.Tools.graph	UPWARD	15	15	0
65963	Default.Base.tool	65975	Default.Tools.result	UPWARD	8	8	0
65963	Default.Base.tool	65976	Default.Tools.subsystem	INTERFACE	2	2	0
65964	Default.Base.util	65959	Default.Base.guiutil	UPWARD	19	19	0
65965	Default.Db	65958	Default.Base.annotation	UPWARD	123	123	0
65965	Default.Db	65964	Default.Base.util	INTERFACE	7	7	0
65965	Default.Db	65972	Default.Tools.manager	UPWARD	26	26	0
65965	Default.Db	65976	Default.Tools.subsystem	INTERFACE	6	6	0
65965	Default.Db	65977	Default.Tools.trend	UPWARD	28	28	0
65973	Default.Tools.metric	65977	Default.Tools.trend	INTRA	139	139	0
65975	Default.Tools.result	65972	Default.Tools.manager	UPWARD	3	3	0
65976	Default.Tools.subsystem	65964	Default.Base.util	INTERFACE	42	42	0
65964	Default.Base.util	65963	Default.Base.tool	UPWARD	0	2	2
65974	Default.Tools.query	65964	Default.Base.util	INTERFACE	94	96	2
65964	Default.Base.util	65972	Default.Tools.manager	UPWARD	56	60	4
65966	Default.Tools.architecture	65964	Default.Base.util	INTERFACE	38	42	4
65963	Default.Base.tool	65959	Default.Base.guiutil	INTERFACE	31	36	5
65966	Default.Tools.architecture	65977	Default.Tools.trend	INTRA	74	80	6
65965	Default.Db	65963	Default.Base.tool	UPWARD	738	802	64

If you analyze a system for the very first time, it is recommended that you get an idea of how badly the system is afflicted by architecture violations. This will be accomplished by marking the illegally referenced subsystems in a subsystem graph (see Figure 7-4). Here, you can see clearly that about one third of all subsystems of Sotograph are used in a manner that is not permitted. At first sight, this might come as somewhat of a shock. However, our practical experiences have proven that this result ranks rather low on the scale of software systems that were developed without architecture testing.

1. One prominent feature of tables displayed in Sotograph is that they often contain ‘Id’ columns. Based on these Id columns, commands can be given, evaluations be started and graphs generated. This feature allows programmers to use the same generic visualization infrastructure for the results of pre-configured and user-specific evaluations.

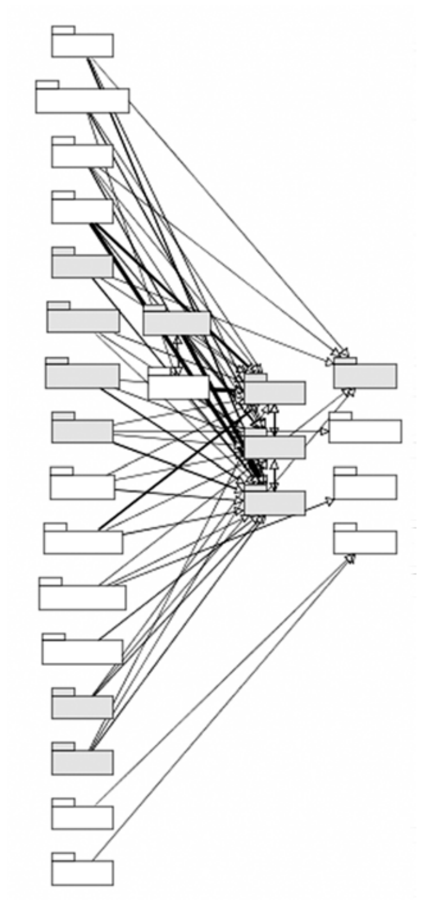
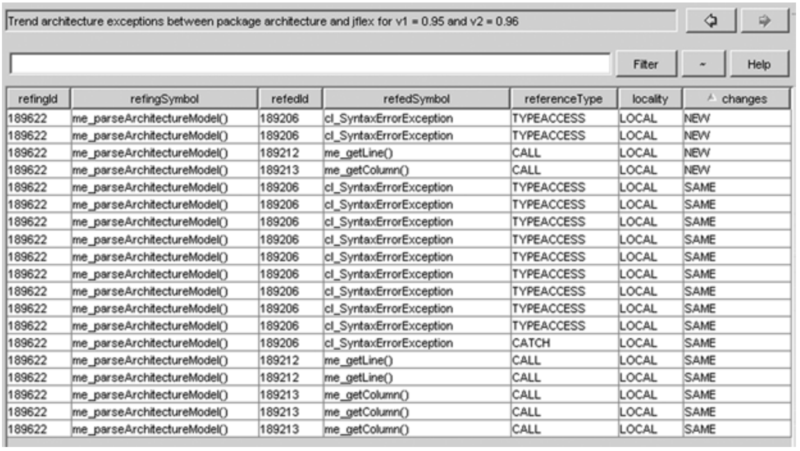


Fig. 7-4
A Marked Subsystem Graph

In the next step of your architecture analysis, we will take a close look at the list of architectural deviations to identify the sources of the detected problems (again, see Figure 7-3). Usually we will encounter quite a variety of problems that can easily be solved by moving classes into another package or by moving packages into another subsystem. In most cases, more than enough architecture smells will remain that can only be cured with more complex refactorings.

Architecture testing should always take place parallel to other project-related work. Therefore, it is important that the ongoing monitoring process isn't too time-consuming – at least not after the first comprehensive analysis. Thanks to Sotograph's Trend support this demand is also realistic. The table's last column (see Figure 7-3) displays those changes that took place between the last two selected versions of the system. In the context of a continuous monitoring it will do to scrutinize the new ones as well as the eliminated architecture violations. Double-click on a table row to zoom in the package level first, the level of basic references next, and last into the source code. The following table (see Figure 7-5) lists all basic references for illegal relations between the packages *architecture* and *jflex* of the subsystems *Tools.architecture* and *Base.util*. You will immediately recognize those illegal relations that were already present in version 0.95 as well as the four illegal relations that have crept into the system in version 0.96.



Trend architecture exceptions between package architecture and jflex for v1 = 0.95 and v2 = 0.96

refingId	refingSymbol	refedId	refedSymbol	referenceType	locality	changes
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	NEW
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	NEW
189622	me_parseArchitectureModel()	189212	me_getLine()	CALL	LOCAL	NEW
189622	me_parseArchitectureModel()	189213	me_getColumn()	CALL	LOCAL	NEW
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	TYPEACCESS	LOCAL	SAME
189622	me_parseArchitectureModel()	189206	cl_SyntaxErrorException	CATCH	LOCAL	SAME
189622	me_parseArchitectureModel()	189212	me_getLine()	CALL	LOCAL	SAME
189622	me_parseArchitectureModel()	189212	me_getLine()	CALL	LOCAL	SAME
189622	me_parseArchitectureModel()	189213	me_getColumn()	CALL	LOCAL	SAME
189622	me_parseArchitectureModel()	189213	me_getColumn()	CALL	LOCAL	SAME
189622	me_parseArchitectureModel()	189213	me_getColumn()	CALL	LOCAL	SAME

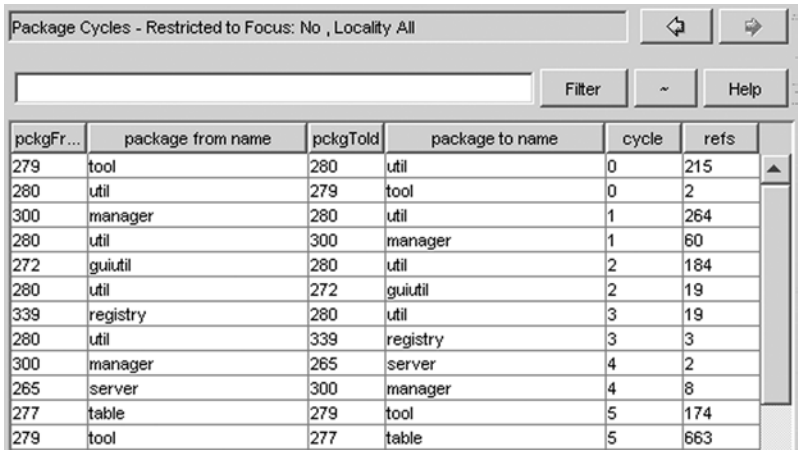
Fig. 7-5
Architecture Violation on the Package Level

7.3 Architecture Analysis Based on Cycles

Sotograph allows the identification of cyclic relations between classes, files, packages and subsystems. The related architecture smells and their negative effects are depicted in sections 3.1.3, 3.3.2 and 3.5.2.

One fundamental problem of cycle analysis are combinatorial explosions. Most of the commercial systems we have analyzed until now are so closely coupled that thousands of cycles will be detected. Thus Sotograph will search all cycles of the length 2 in a first step, then eliminate those for further searching, and scan for continuously longer cycles. This procedure enables interpretation of the cycle analysis' results within a reasonable time frame without losing relevant information.

At the beginning of any cycle-based architecture analysis, the most sensible approach is to search for package cycles across subsystem boundaries. The table below (see Figure 7-6) shows the results of that query. One line represents a relation in a cycle, and all lines of the same *cycle* Id represent the entire cycle.



Package Cycles - Restricted to Focus: No , Locality All

pckgFr...	package from name	pckgTold	package to name	cycle	refs
279	tool	280	util	0	215
280	util	279	tool	0	2
300	manager	280	util	1	264
280	util	300	manager	1	60
272	guiutil	280	util	2	184
280	util	272	guiutil	2	19
339	registry	280	util	3	19
280	util	339	registry	3	3
300	manager	265	server	4	2
265	server	300	manager	4	8
277	table	279	tool	5	174
279	tool	277	table	5	663

Fig. 7-6
Analysis of Cycles on the Package Level

The first two lines point at a cyclic relation between the *tool* and *util* packages. Since only two references exist from *util* to *tool*, but 215 references in the other direction, it is safe to assume that the references from *util* to *tool* should be eliminated. For further analyses, you can zoom in the basic references level with a double-click. A second double-click will display the corresponding source code. This view works well if you start to break up cycles. Since Sotograph will break down long cycles into short ones during analysis to avoid combinatorial explosions, you must make a survey of where critical amassments of cycles are present.

For this purpose, you must sort the table, listing the detected cycles by package name. When this is done, you'll recognize at a glance which packages are involved in what number of many cycles. The tangle surrounding such critical packages can be viewed best in a graph. Figure 7-7 shows the graph that contains all packages which maintain cyclic relations with the package tool. This graph exemplifies very well how software systems start to 'lump' due to their constantly growing number of cycles in the course of their evolution. It is pretty obvious that it will take new developers quite an effort to familiarize themselves with such a chaotic dependency graph.

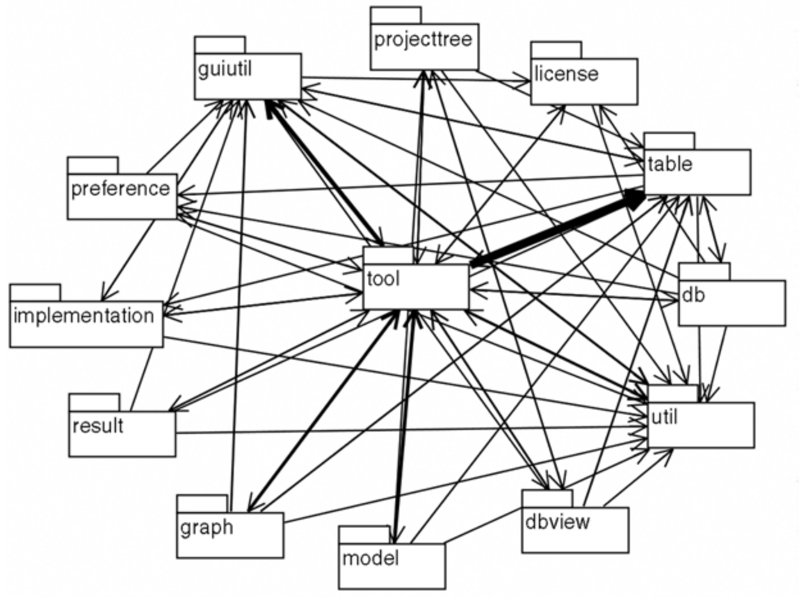


Fig. 7-7
Packages in Relation to the Package Tool

In practice, it is not always wise to eliminate all cyclic relations from a software system. On one hand, it can be advantageous to have two classes using each other on the class level. On the other hand it often takes too much effort to eliminate all unwanted cycles in systems that were not developed with the aid of cycle analyses from the very beginning. This is why support of a cycle analysis is indispensable: it helps to distinguish existing cycles from newly introduced ones. This kind of information is provided by Sotograph's metrics tool.

7.4 Metrics-based Architecture Analysis

The central problem of any type of metrics-based software analysis is the huge amount of metrics values that will be generated even for medium-sized software systems. All this information must be examined and interpreted.

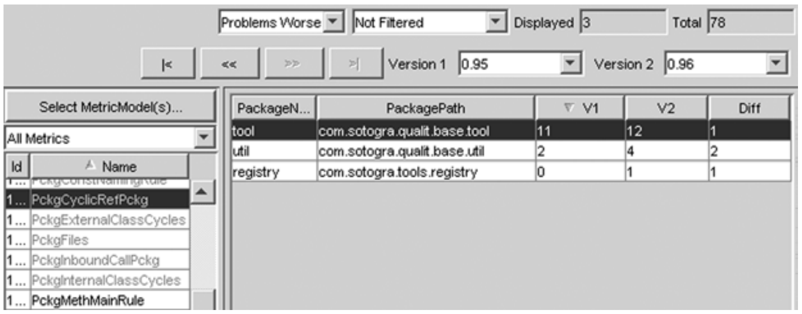
The fundamental prerequisite for an efficient metrics-based first analysis is sophisticated tool support. Here are examples of what such tool support can do:

- Fast elimination of metrics results for parts of the system that have already been recognized as being irrelevant or basically problematic.

- Explanation of a metrics value. Without an explanation, one has to search the reason for each suspicious metrics value in the source code and the IDE.
- Visualization of metrics values. Particularly for high number of couplings the graph is best suited for determining what a certain value means.

For a regular analysis that proceeds parallel to other project work, it is first and foremost important that developers review only 'relevant' values. In our experience, these will normally be metrics values that were bad before and have now gotten worse, or metrics values which were bad but have improved.

Figure 7-8 shows a part of the Trend metrics tool of Sotograph with its 'Problems Worse' filter activated. Metrics underlaid in dark color contain values that have not been filtered out. The selected PckgCyclicRefPckg metrics calculates the number of packages with which a single package maintains a cyclic relation. In this example, we can see that the package *base.tool* that already caught our attention in the cycle analysis described in the previous section has been cyclically coupled with a new package between releases 0.95 and 0.96.


Problems Worse | Not Filtered | Displayed 3 | Total 78

Version 1 0.95 | Version 2 0.96

All Metrics

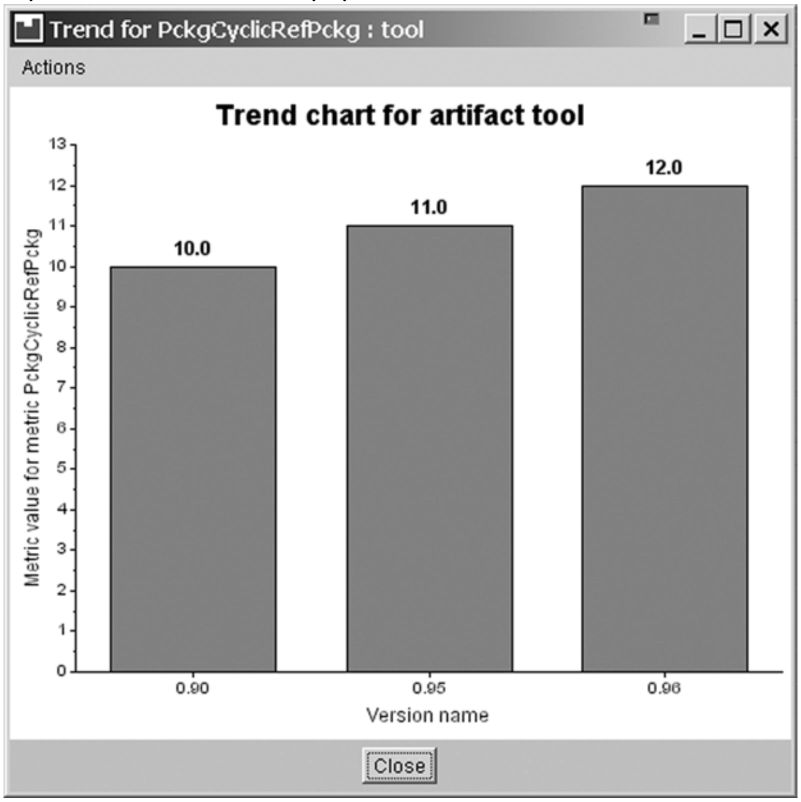
Id	Name
1...	PckgCyclicRefPckg
1...	PckgExternalClassCycles
1...	PckgFiles
1...	PckgInboundCallPckg
1...	PckgInternalClassCycles
1...	PckgMethMainRule

PackageN...	PackagePath	V1	V2	Diff
tool	com.sotogra.qualit.base.tool	11	12	1
util	com.sotogra.qualit.base.util	2	4	2
registry	com.sotogra.tools.registry	0	1	1

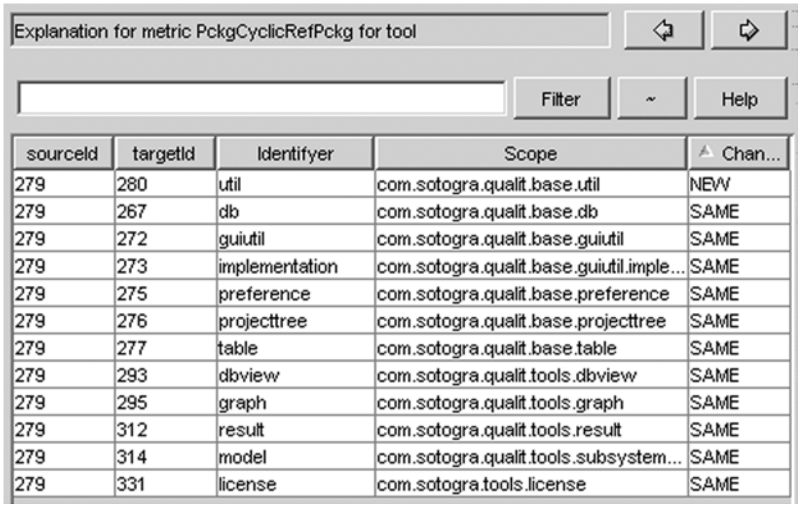
Fig. 7-8
A Section of the Trend Metrics Tool

In this case, it would be interesting to learn if the number of packages which have cyclic relations with *base.tool* constantly increases. This information can best be obtained from the diagram in Figure 7-9. This diagram proves that the increase occurred already between releases 0.90 and 0.95 – another indication that the package *base.tool* and the packages closely coupled with it should be examined much closer in the future, prior to any changes and expansions. In this way we can prevent that new problems are introduced to the system, and eventually establish a more orderly system.

Fig. 7-9
Trend Chart


Trend for PckgCyclicRefPckg : tool
Actions
Trend chart for artifact tool
Metric value for metric PckgCyclicRefPckg
0
1
2
3
4
5
6
7
8
9
10
11
12
13
10.0
11.0
12.0
0.90
0.95
0.96
Version name
Close

After the diagram window has been closed, it makes sense to carefully analyze what exactly has changed in *base.tool*'s vicinity. A double-click on the metrics value will take us to the explanation in Figure 7-10. Here we can see that a cyclic relation with package *base.util* has been newly inserted. The package *base.util* could now be highlighted in the last generated package graph, or one could look at the relations between *base.tool* and *base.util* in detail.



Explanation for metric PckgCyclicRefPckg for tool

sourceId	targetId	Identifyer	Scope	Chan...
279	280	util	com.sotogra.qualit.base.util	NEW
279	267	db	com.sotogra.qualit.base.db	SAME
279	272	guiutil	com.sotogra.qualit.base.guiutil	SAME
279	273	implementation	com.sotogra.qualit.base.guiutil.imple...	SAME
279	275	preference	com.sotogra.qualit.base.preference	SAME
279	276	projecttree	com.sotogra.qualit.base.projecttree	SAME
279	277	table	com.sotogra.qualit.base.table	SAME
279	293	dbview	com.sotogra.qualit.tools.dbview	SAME
279	295	graph	com.sotogra.qualit.tools.graph	SAME
279	312	result	com.sotogra.qualit.tools.result	SAME
279	314	model	com.sotogra.qualit.tools.subsystem...	SAME
279	331	license	com.sotogra.tools.license	SAME

Fig. 7-10
Explanation of Cycle Metrics

After the cycles have been analyzed, one can proceed to survey the critical values of other metrics. The procedure is quite similar to that used for the PckgCyclicRefPckg metrics.

Other Smells that Can Be Detected with Metrics

Basically, the following architecture smells can be detected automatically with metrics or specific analysis queries.

- Unused artefacts (3.4.1)
- Too small artefacts (3.4.3, 3.5.3)
- Too big artefacts (3.4.4)
- Treelike dependency graphs between classes (3.2.2)
- Type queries (3.3.1)
- Listlike inheritance hierarchy (3.3.2)
- Too deep inheritance hierarchy (3.3.6)
- Packages too deep or unbalanced (3.4.5)
- Packages not clearly named (3.4.6)
- Too many subsystems (3.5.4)
- Subsystem API too big (3.5.6)

In contrast to the results yielded by architecture-based and cycle-based analysis, the values calculated for these metrics will only point at potential problems, each of which must be examined in detail. Artefacts that are recognized as not being in use are not necessarily unused. For instance, it is possible that objects created via reflection at runtime and used polymorphically are not recognized as being in use. For many metrics the main question concerns the upper and lower boundaries: at which point does a metrics value begin to indicate a problem, e.g. at which point does a class or a package become too big?

To usefully apply metrics in projects, it is in our experience recommendable to let the team decide which metrics are beneficial for this specific project and thus should be further pursued in the course of the project. Then project-specific, commonly accepted upper and lower boundaries must be defined for the selected metrics. The calculation of, e.g., too small and too big artefacts does only make sense if the team manages to agree on upper and lower boundaries. Otherwise, the determination of such metrics values will lead to futile discussions, or the metrics will simply be ignored.

7.5 Support for the Preparation of Large Refactorings

As the name already indicates, large refactorings can have a noticeable effect on large parts of a software system. Therefore, it is all the more important to understand which code sections will be affected in what manner before beginning with a refactoring. Especially where restructurings of libraries are concerned, this sort of information can hardly be obtained with the tools that are available today. Sotograph can make it much easier to understand software systems in general, as well as specifically the effects of changes. This is demonstrated here using an analysis of the internal API of Sotograph's tool framework. For lack of space, we will focus exclusively on the inheritance interface. The call interface can be analyzed analog.

Before the actual analysis of the interface is carried out, we have to get an idea of how broadly the framework will be used. In our example, we are dealing with the subsystem *Base.tool*. Here, we can let the *Crossreferencer* identify all classes that use *Base.tool* and have them displayed in a subsystem graph. Likewise, it is possible to generate a graph that shows which classes inherit from classes in *Base.tool*. Figure 7-11 depicts two sections of each of the two graphs. Right away it will become clear that changes of *Base.tool* can affect significant parts of the system. It is remarkable that most subsystems do not merely use *Base.tool*, but also inherit from it. However, it is not surprising that the central tool framework is widely used for the implementation of software analysis tools.

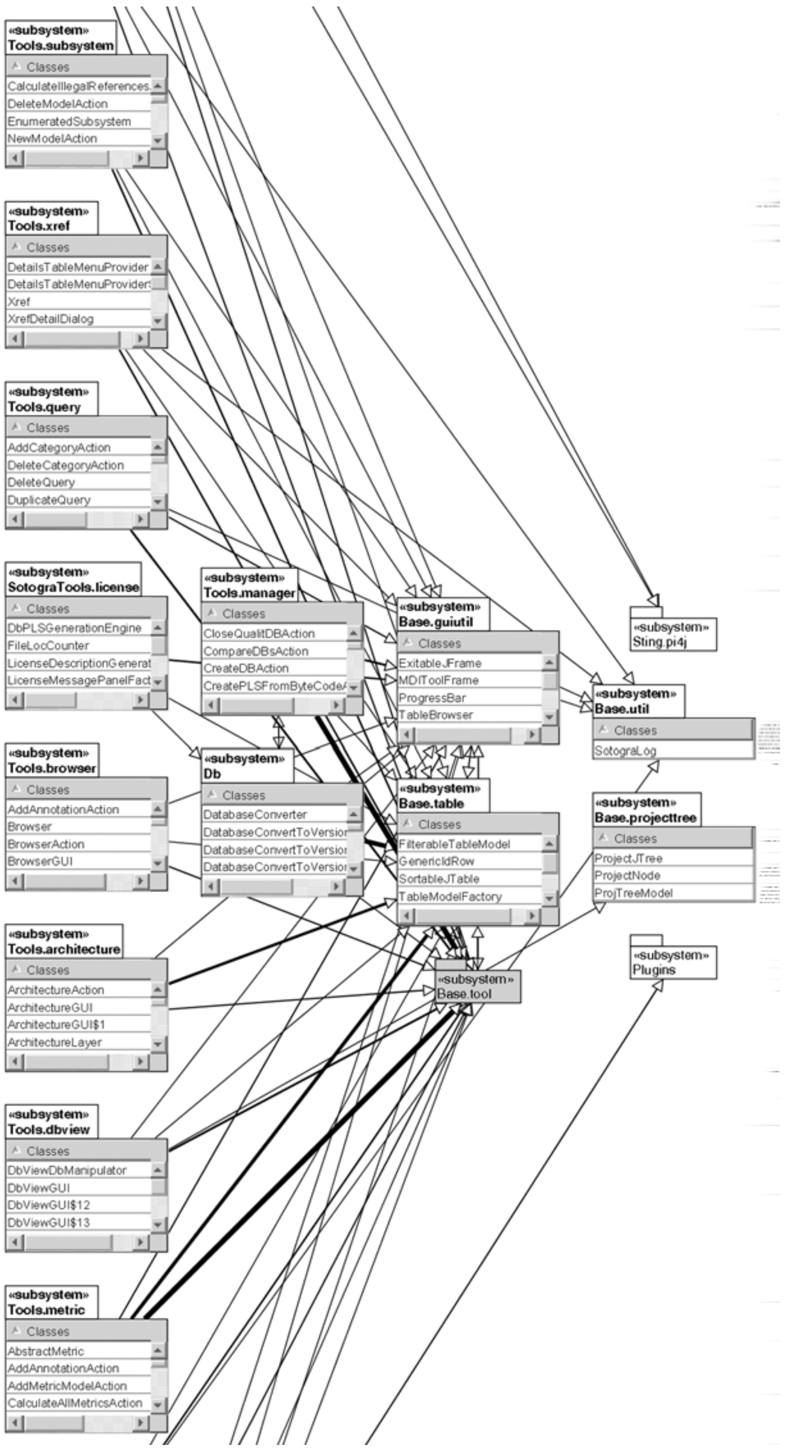


Fig. 7-11
Marked Subsystem Graph 1

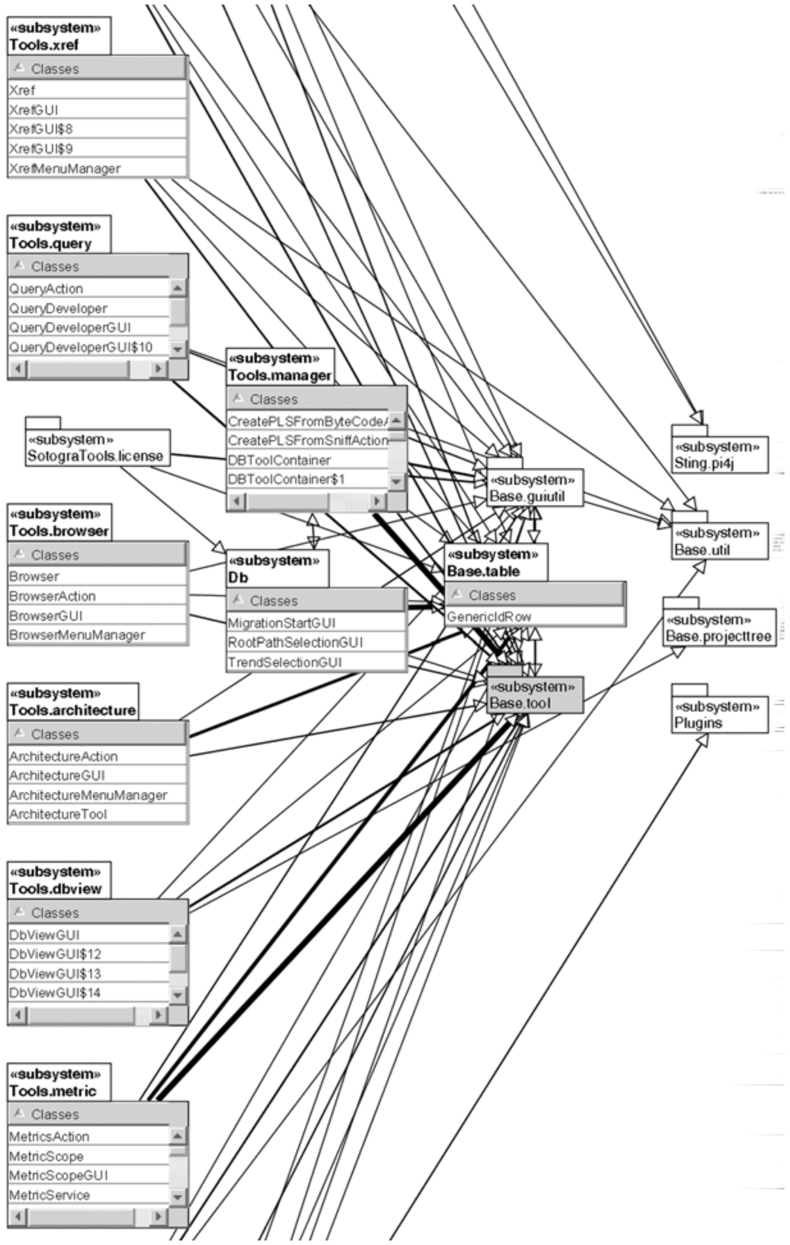


Fig. 7-12
Marked Subsystem Graph 2

In the next step, we are going to call up a display of the inheritance hierarchy of the subsystem *Base.tool*' classes. In this graph, we will then highlight the classes which were overwritten outside the subsystem and insert the overwritten methods, as depicted in Figure 7-13. The disadvantage of this form of presentation is that no difference between often and rarely overwritten classes and methods is visible.

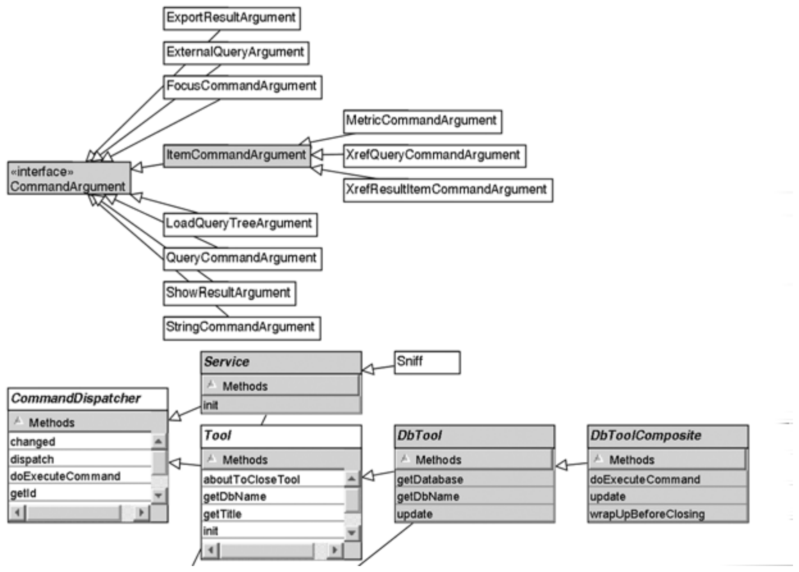


Fig. 7-13
Marked Inheritance Hierarchy

In this way Sotograph also offers the option of identifying all those classes and methods of a subsystem or package which are typically overwritten, i.e. those that are overwritten by most of N most important clients. Figure 7-14 depicts the resulting Graph, which shows far more clearly than the previous image what changes should rather be avoided if one doesn't wish to provoke severe side-effects. It also proves clearly that – in spite of generous use of the tool framework – only few methods are overwritten by a relevant part of the clients, at least in the section of the graph displayed here.

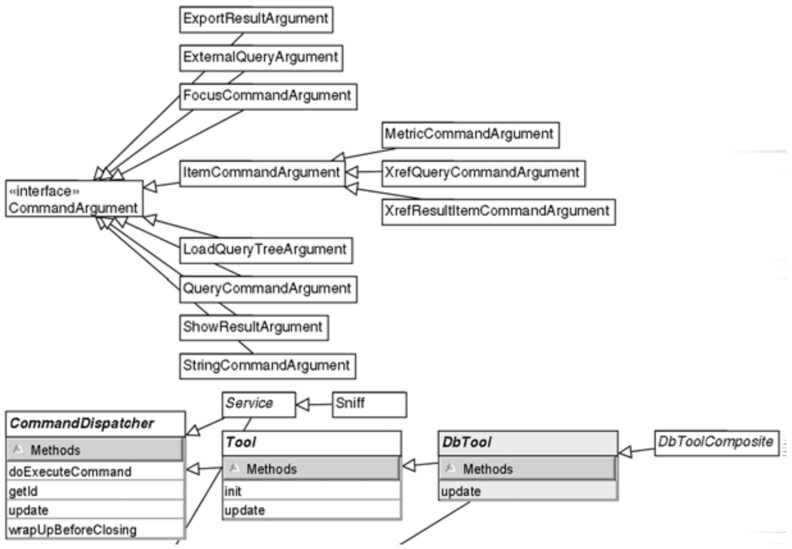


Fig. 7-14
Marked Hierarchy, based on Frequency

Based on such graphs, a rough assessment of the effects of a refactoring on a part of the tool framework can be made. Prior to the actual refactoring, one should proceed further to get an in-depth impression of the analysis done with Sotograph and examine various aspects in greater detail.

7.6 Support of the Refactoring Process

A fundamental problem of large refactorings is that often clients of the restructured code are also affected by changes. This will have quite an impact on much-used libraries, whose users are no longer in touch with the developers, and for which the developers are unable to directly adapt the customer code in the course of their refactoring. In this case, it is pivotal that the users learn what has changed in between the different versions of the respective library.

Sotograph possesses a variety of query options which help developers to find out which artefacts were generated, deleted or modified between two versions. The overview for *Base.tool* will result in the following figures on the class and method levels for differences between versions 0.90 and 0.96:

New classes	3
Deleted classes	0
Modified classes	49
New methods	21

New public methods + public methods with changed signature	11
Deleted methods	0
Modified methods	88
Modified public methods without changed signatures (new + modified)	69

The data on which these figures are based can then be visualized. Figure 7-15 pictures all new public methods and all public methods with changed signatures in the inheritance hierarchy's context. All modified classes are marked.

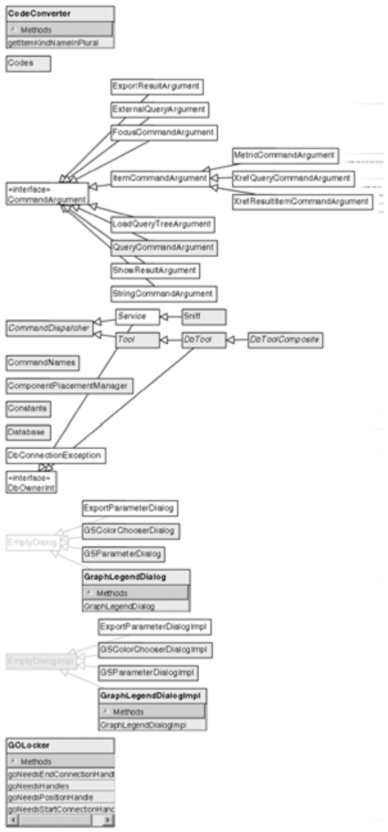


Fig. 7-15
Changed Classes, Marked

7.7 Conclusion

This chapter shows that a tool such as Sotograph allows continuous monitoring of a software system's architectural quality during development without much effort. Thus it is feasible to cure many architecture smells before they become so firmly rooted in the system that they can only be eradicated with very sophisticated refactorings.

For several years now, the Sotograph has been in use for professional analyses of large software systems. Apart from a few exceptions, a high number of architecture violations and cycles were found in the examined systems. These analyses clearly proved that architectural decay in most cases begins with the first code lines and not later on, in the maintenance phase. This held also true for the implementation of Sotograph itself, as this chapter showed.

Furthermore, it is an interesting experience that architecture analyses and large refactorings contribute to enabling economic maintenance of such projects which have been declared not maintainable, and this with only a few man months of work. Of course, these experiences are only transferable to projects that concern software systems of a decent technical quality.

In this chapter, the Sotograph was used as a vehicle for illustrating the technological possibilities of architecture analysis as well as for those of supporting the preparation and execution of large refactorings. Further information about Sotograph can be found at: www.software-tomography.com.

8 Conclusion

Object Orientation for Large Projects

Object-oriented programming has been around for a couple of decades. In its early days, it was quite difficult for this approach to become established. Lack of tool support as well as performance concerns often led to the continued use of 'classic' programming languages such as Cobol or C, in spite of the propagated superiority of object-oriented concepts.

Over the years, object-orientation succeeded in entering the world of commercial software development; venturing there from small, not business-critical systems. Today, object-oriented programming languages offer – besides exclusively object-oriented concepts – everything that makes them perfect for application in extremely comprehensive projects:

- A standardized programming language
- Platform independence
- Performance
- Support from popular manufacturers
- Libraries and frameworks for all significant technologies, such as databases, network communication etc.
- Powerful tools plus highly integrated development environments
- Products for application with high transaction rates, transaction monitors and application servers
- Developers with the necessary know-how

Moreover, a substantial part of the available tools and libraries is open source software. Now, we can finally roll up our sleeves and get to work on switching the no longer maintainable systems from the good old days of Cobol and C to the seemingly superior object-oriented technologies.

Object-oriented Legacy Systems

But wait a second here! The already existing object-oriented systems should serve as a warning: a considerable number of these object-oriented systems falls in the category of legacy software, which is difficult to service. These systems quite clearly prove that not much will be gained with object-oriented programming languages and technologies alone. The newly won flexibility will not automatically result in easy-maintenance systems. If this flexibility is wrongly applied, it can even make system maintenance harder than it would have been using classic, non-object-oriented technologies.

Recognizing & Solving Architecture Problems

We hope that this book will contribute to making object-oriented systems easier to service. We don't pursue the goal of delivering a perfect system design at an as early stage as possible – we think this approach is illusional anyway, particularly for large systems. Instead, we hope that the contents of this book will help to point at ways for recognizing and solving architecture problems in systems with the aid of various refactoring techniques.

Architecture Smells

Architecture smells indicate where architectural issues might be present. Especially the 'lumping' problem drastically reduces the maintainability of large systems. While we are still smiling mildly at programs written in Basic, which – thanks to the *goto* statement – happily & frequently turn into spaghetti code, similarly critical spaghetti structures are not rarely present in more complex systems these days. This phenomenon will not occur on a single method's or statement's level, but on higher ones, such as classes, packages and subsystems. Here, clearly defined structures are often lost. Since recently though, adequate tools – like, for example, Sotograph – are available that can identify these smells in a system.

Whereas finding potential architecture smells with the aid of available tools is mostly a rather menial task, evaluating smells requires a lot of architecture *experience*. Whether there really is a problem or not strongly depends on the system context.

Refactoring Plans

Minor structural weaknesses can be eliminated in the course of our everyday development work. We use small refactorings, preferably aided by a suitable IDE, to keep the structure clean and easily changeable. However, should architectural problems that call for more comprehensive code restructurings arise, creativity is needed more than anything else. In such situations, we must look for ways to solve the architecture problem on one hand and modifying the system in small steps on the other. We can create refactoring plans that will guide us in solving the problem. These plans must constantly be adapted to reflect the refactoring's progress. During a large refactoring, we will always learn something new that will lead us to further adapting our plan.

Database Refactorings

Large refactorings are rarely limited exclusively to the program code. Most commercial systems work with (relational) databases. Therefore, data structures too must often be modified as well in the course of a refactoring. Problems will arise because relational databases hardly offer any options for concealment: the effects of changes to the data model can hardly be restricted to merely one partition. In addition, not only must the data structures of an already running system be altered, but the existing persistent data must migrate to the new data schema.

In the respective chapter of this book, we came a step closer to our goal of improving database structures in the course of an evolutionary development process. We also gained and discussed some expert knowledge that shows how evolutionary changes to an object-oriented system can affect the database connection, and how this task too can be solved using an evolutionary approach.

API Refactorings

Subsystems are important instruments for structuring large systems: they hide their internal realization behind a published interface (published API). Other subsystems access this subsystem exclusively over the API. When architecture problems exist on the subsystem level or even in layers, the subsystems' interfaces usually must be adapted as well.

This poses special challenges for refactoring. After all, the subsystems have entered into *contracts* with each other via the interfaces that govern their collaboration. These contracts cannot be changed by one side alone: The client subsystems must migrate to the altered interface. With a few simple tricks and tools, the developers of the subsystem to be changed can make life (that is: migration) much easier for the developers of the using subsystems. The techniques discussed here can also be applied for restructuring frameworks, for example, without provoking a high migration demand for the applications.

The Architect's Role

Agile methods negate the validity of large architecture designs (*Big Upfront Design*). Consequently, this also implies some criticism of those who create such big architecture designs: the software architects. We do not believe that agile methods do away with the need for software architects. Of course, systems developed with agile methods do have a software architecture too, and of course this architecture must meet present requirements. In an agile project, the architecture can develop in the course of the project period, but in a more complex project somebody must monitor this development and alert others to emerging problems, i.e. architecture smells. Problems on the architecture level can not be efficiently found by simply reading the code. Here, we clearly see the task of software architects in agile projects: they should not merely define the architecture, but first and foremost provide their architectural experience as a service to other team members.

Outlook

The discussion about large refactorings certainly won't end with this book. The concepts and procedures presented here are derived from our project experiences. Their application in further projects will create new incentives for the future discussion in the field of large refactorings.

9 Glossary

Acceptance Tests

Tests assuming the user's perspective that describe the system's acceptance criteria. Ideally, an as large number of these tests as possible shall be automatically executable. Of course there are limitations to this approach, particularly where the system's ergonomics is concerned. See Chapter 4.

Architecture Smell

An *architecture smell* is a smell that indicates a problem in the software architecture. Whether such a problem does exist or not must be verified through detailed testing. See also *Code Smell.* See Chapter 3.

Automated Refactoring

Automated refactorings are refactorings which are supported by an IDE and therefore can be executed automatically. The IDE guarantees that the system's behavior will remain unchanged. In consequence, automated refactorings are also always *safe refactorings*. Moreover, automated refactorings can be carried out in a very short time, regardless of the system's size.

Basic Refactoring

Refactorings that are primarily based on elementary, object-oriented constructs. Most refactorings introduced in [Fowler 99] are *basic refactorings*.

Code Smell

A *code smell* is a smell that indicates a problem in the code. Whether such a problem does exist or not must be verified with detailed testing. See also *Architecture Smell.* See Chapter 3.

Detailed Refactoring Plan

The *detailed refactoring plan* specifies details of a refactoring plan. It breaks down single refactoring steps into basic refactorings wherever this is feasible and analyzes the remaining modifications in detail. See Chapter 4.

Function Test

In the context of this book, the term *function test* is used synonymously with *acceptance test*.

Large Refactoring

Refactorings are considered *large refactorings* if the following criteria are met: they last longer than a day; they alter significant parts of the system; and they become visible to all developers involved in the project even while the refactoring is being executed. See Chapter 4.

Manual Refactoring

Manual refactorings are not supported by the IDE, which means that developers have to conduct them manually. They are the opposite of *automated refactorings*.

Merciless Refactoring

Merciless refactoring reflects a particular attitude and practice in software development: developers will not wait with refactorings until a system structure has degenerated. Instead, even minor flaws will be eradicated at once. See Chapter 2.

Merge

Merging is the incorporation of parallel changes to one and the same class. One can only partly automate this process with *Merge tools*. See Chapter 4.

Merge Tool

Merge tools support the merging of two simultaneously altered versions of a class. They serve to point out differences between both classes and thus enable developers to either manually incorporate changes to one class in the other one or to let the merge tool automatically integrate these changes.

However, automatic *merging* is not safe. It is possible that merging results in a non-compilable class. See Chapter 4.

Public Interface

The public interface of a class. The public interface includes all public methods and attributes. We must distinguish between *public* and *published API*. See Chapter 3.

Published API

The *published interface* of a component, which allows us to use the components' services. See Chapters 3 and 6.

Refactoring

Refactoring means changing the internal structure of a software in such a manner as to make it easier understandable and changeable without affecting its visible behavior at runtime. See Chapter 2.

Refactoring Plan

A *refactoring plan* sketchily lists the single steps required in the course of a large refactoring. The plan is discussed by all members of a team. It should fit onto a flip chart and be posted publicly, i.e. clearly visible to all those involved in the project. The large refactoring's progress will be visualized on the refactoring plan (by checking off the single steps). The refactoring plan is further specified by the *detailed refactoring plan*. See Chapter 4.

Safe Refactoring

Safe refactorings are refactorings that can be executed without risking changes to the system's behavior or creating errors. If, for example, a tried step-by-step instruction for a refactoring is available (such as the *Mechanics* in [Fowler 99]), the refactoring can be carried out with no risk of creating new errors.

Save Point

A *save point* is one stage of a large refactoring at which the system structure is definitely better than prior to refactoring or, respectively, better than before the previous save point. See Chapter 4.

Smell

A *smell* hints at a potential problem in the system. See Chapter 3.

Unit Test

A *unit test* tests the components on which a system is based. In non-object-oriented imperative programming, single procedures and functions are tested. In object-oriented systems, classes and sets of classes are tested.
There are open source tools available that allow developing and running unit tests for almost every programming language. See Chapter 2.

Unsafe Refactoring

Refactorings for which no tried step-by-step instructions are available that would allow their safe, incremental execution. One example of an *unsafe refactoring* is the renaming of a class.