Refactoring
Benefits and Disadvantages of an Amazing Technique

Michael Hunger

25th October 2000

Contents

List of Figures

Intention of this paper

With the Book "Refactoring - Improving The Design Of Existing Code" by MARTIN FOWLER [Fowler] the topic was raised from the realm of Smalltalk to a broader community that shares one very important problem.

Everyone knows of and has experienced the decay of existing software. Refactoring intends to work against this decaying.

This paper discusses the basics of refactoring and its uses for real-world projects. It reflects the surprisingly long (in terms of computer science) history of refactoring and the evolution and development of tools which allow the user to apply refactorings in a more comfortable way. Some of them already automate the application of whole refactorings as the Refactoring Browser [Refactoring Browser] and others offer considerable possibilities for evolving. The tools are tested with a small set of classes but due to the fact that most of the tools offer only a small subset of refactorings the tests were quite incomplete.

The use of refactoring within the development process is considered as well with a special focus on the methodology which embraces refactoring as one of its basic principles Extreme Programming [Beck,Xp].

The main references for this paper were the book mentioned above and the web sites [Wiki] which have specialized in refactoring.

In the third chapter of the paper one of my own projects - the Metaworks Framework is exemplarily and partially refactored to illustrated the benefits and problems which arise when refactoring legacy code.

Because refactoring has become an interesting topic within the programmers community there is much material to be found on the web, where people report of their own experiences when doing refactoring. This paper should also be a kind of such a report but not as intriguing as theirs as I am new to the field of refactoring and have to learn and experience it much more.

Chapter 1

Introduction to Refactoring

1.1 Introduction

You know bad code when you see it. It tends to have large classes with lots of very long methods, which actually look more like a piece of code of a traditional procedural approach, doing much too much. It contains duplicate code everywhere framed by large switch statements and complex conditionals. All of this is written in a very dense style with many comments keeping it from really being dense code.

This is an vision which no one wants to encounter. But as reality goes it is very hard to keep code consistent to the original design of a system when it evolves within or after the project. Requirements are changed, features are added, programmers change and all these things tend to happen under a very tense schedule. These factors imply that the code is not maintained but only extended resulting in hard to read and to understand, complicated code which has only little to do with the design which once upon a time represented the system, the code stands for.

If one is asked to add functionality to such a kind of system (most likely one in production, if the project went that far) the first reaction should be to refuse. After a second thought there is the possibility to throw away the current system (which tends to be scarcely documented as well) and restart from scratch. The third thing one can do about something like that is taking a break for a time (depending on the size of the project) to refactor the code to make it understandable and maintainable again. And after that it should be no problem at all to implement the wanted functionality.

The best way to solve this problem would of course be never to let something like this happen. If taken care of, the code would never had got the chance to decay and to loose its structure and design.

That can be done by looking at the code before extending or modifying it. If the code is in a good shape, it should be no problem at all to add the functionality. But if it will take a lot of time to understand the code and extending it would require some hacks which violate the intended design, the code should be refactored first to regain the attributes needed for easy maintenance. Only after that it can be extended.

1.2 What is Refactoring?

As many people were involved in the publication of refactoring there are also a lot of definitions around. I've found this one of Ralph Johnson [Wiki, ReFactor] quite suited.

> Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probably the most important.

The author of the reference book [Fowler, p.53] for refactoring MARTIN FOWLER also provides a more general definition in two parts:

> Refactoring(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
>
> Refactor(verb): to restructure software by applying a series of refactorings without changing its observable behavior.

The most important aspects of these definitions are that refactoring must not change the observable behavior of the software system (not observable is for instance the runtime aspect which may change due to refactoring) and the changing of the structure of the software towards a better design and more understandable and reusable code.

> Refactoring provides a technique for cleaning up code in a more efficient and controlled manner. [Fowler, p.54]

The consistency of the observable behavior must be guaranteed when doing refactoring. Either you are able to use an automated tool to perform refactorings. Then you can rely on the operations the tool performs as the basic refactorings which can composed to larger ones are proved to be semantics preserving ([Opdyke, Thesis], [**?**]).

The second best way to be sure the behavior is not changed, is the use of automatic testing. This is a technique which is also used as a principle of Extreme Programming [Beck,Xp] (like refactoring as well) which implies that for each aspect of the software which is not simple enough to contain no bugs (e.g. accessors), there has to be a test. Most likely these test should be designed that way that they try to break the software, e.g. expose errors.

1.2.1 Testing using Unit Tests

As those test run automatically it is a very convenient way to check if something that was changed did corrupt the system (at least the test). Then the change done should be observed very carefully, and if necessary undone and repeated in smaller steps. Although this is not a way to catch any error hidden inside the software but to find a lot of them. Automatic testing has the advantage of not requiring user interaction. Those tests can be run after each change made to the system. Bugs that were produced when implementing the change can be spotted immediately.

Much of help for automatic testing is the JUnit [Gamma, JUnit] testing framework that was developed by KENT BECK and ERICH GAMMA in 1997. It is based on white-box, regression testing strategies which are implemented by the developer of the concerned code within their own test classes which are called by the JUnit testing environment.

The importance of testing is stressed in the following quote:

> The fewer tests you write, the less productive you are and the less stable you code becomes. The less productive and accurate you are, the more pressure you feel. ... You would see the value of the immediate feedback you get from writing and saving and rerunning your own unit tests. [Gamma, JUnit]

The following excerpts were taken from an article at JavaWorld [Nygard] where two Sun developers expose their enthusiasm for unit testing and show possibilities for extending unit testing to check distributed components such as Enterprise Java Beans.

> We can never over test software, but we seldom test it enough. ... Unit testing is a critical, but often misunderstood, part of the software development process. Unit testing involves individually testing each small unit of code to ensure that it works on its own, independent of the other units. In object-oriented languages, a unit often, but not always, equivocates to a class. If developers knew for certain that each piece of the application works as it was designed to do, they would realize that problems with the assembled application must result from the way the components were put together. Unit testing tells developers that an application's pieces are working as designed.
>
> Process of testing:
>
> 1. Decide what the component should do.
> 2. Design the component. This can be done formally or informally, depending on the complexity of the component.
> 3. Write unit tests to confirm that behavior. At first, the tests will not compile, since the code they test is not yet written. Your focus here is on capturing the *intent* of the component, not on the implementation.
> 4. Start coding the component to the design. Refactor as needed.
> 5. When the tests pass, stop coding.
> 6. Consider other ways the component can break; write tests to confirm and then fix the code.
> 7. Each time a defect is reported, write a test to confirm. Then fix the code.
> 8. Each time you change the code, rerun all tests to make sure you haven't broken anything.

Important attributes of unit tests are:

- they cover each small unit of code (e.g. a class)
- they are organized into groups or suites of tests
- they have to execute 100
- it should be run often (e.g. each 10-15 minutes) covering only that amount of changes done
- they are regressive tests, i.e. the changes since the last test are responsible for breaking the test
- they raise confidence in the system as a whole, because the components are thoroughly tested
- they don't focus on code coverage but on vulnerable parts which are most likely to break
- they allow developers to write tests first and code tested by them afterwards

- it is like welding requirements in code, thereby gaining clarity in purpose, and preserving them (the requirements) as long as the code lives
- they beware of over-design
- they allow changes without second thoughts, as the tests will point on any failure made
- they help when maintaining as the tests more clearly display the intended use of the class than the class itself

Problems of testing distributed components, or those heavily relying on their (complicated) context:

- it would be necessary to build a test stub to test the components directly
- this test stub would almost be as complicated as the real context (e.g. Application Server, Database)
- therefore the testing environment has to run within the context to test the individual objects, (e.g. as a servlet on a application server)

William Wake has written some essays regarding unit testing, and in one of them [Wake, JUnit] statet the following point:

> Unit tests can be tedious to write, but they save you time in the future (by catching bugs after changes). Less obviously, but just as important, is that they can save you time now: tests focus your design and implementation on simplicity, they support refactoring, and they validate features as you develop.

Martin Fowler [Fowler, p.91] tells about testing practice:

> By writing the test you are asking yourself what needs to be done to add the function. Writing the test also concentrates on the interface rather than the implementation (always a good thing). It also means you have a clear point at which you are done coding - when the test works. ... Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.

The authors of JUnit say what kind of test should be written [Gamma, JUnit]:

> You can always write more tests. However, you will quickly find that only a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they should fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.

1.2.2 Refactoring Consists of Small Steps

All of that leads to another principle of refactoring - small steps. When changing a lot of the system at one time it is most likely that also a lot of bugs were introduced in doing this. But when and where these bug were created is no longer reproduceable. If a change is implemented in small steps with tests running after each step the bug would occur in

the test after introducing it into the system. Then the step could be examined or after undoing the step it could be split in even smaller steps which can be applied afterwards.

These steps are taken when refactoring a software system:

- find the place where a refactoring has to be applied. This can be done by having problems when trying to understand, extend or restructure the system or by looking for the code smells (see 1.6, p.20) in the actual code.
- if a unit test for the code under consideration exits, run the test to see if it is completed correctly
- otherwise write all necessary unit tests and get them running
- locate a refactoring that can be sensibly applied either by searching your mind or by looking at the catalog presented in the book of MARTIN FOWLER [Fowler].
- follow the step by step instructions to implement the refactoring
- run the tests between each step to ensure that the behavior has not changed
- if necessary adapt the test code to changed interfaces
- when the refactoring is successfully used to restructure the code, run the tests again, integrate and run the complete unit tests and functional tests
- if no problem shows up you are done

The steps taken when applying the refactoring should be:

- small enough to oversee the consequences they have,
- reproduceable to allow others to understand them and perhaps get rid of them,
- generalized the way, that they are more a rule that can be applied to any structure that has a special appearance,
- written down to allow sharing these steps and to keep a reference, completed with scope, problems, examples and most useful with step by step instructions how to apply them.

If such a set of rules is available, the application of these steps is more a craft than creative programming. That "stupid" work is more likely to be error free than free thinking about how to reorganize the system structure in order to get a better design. And it offers another possibility that many programmers dream of. The creation of a tool that supports refactoring in applying these boring steps by itself.

But stating there is nothing creative in refactoring is wrong. The interesting part lies in finding the bad structured parts of the system which are in need of refactoring. As there is no metric for good design, it is a skill which has to be developed by practicing refactoring. Another important aspect is to decide whether to refactor or not, there are factors that make it harder or even impossible to refactor a certain piece of code.

A set of rule like the one described above evolved when those people which have been refactoring for years started to gather their experience and wrote it down. It is like everything that is contained in the knowledge of experts. If it is written down it can be shared with a great number of people who can rely on this knowledge and reference it when they are in need of such information.

Although a lot of people encounter deja vu's when reading such repositories, it is a good sign that many people have done likewise. But they have done without explicitely thinking of it as an a refactoring but just intermingled with the normal day to day programming. That bears the danger of mixing up implementing new functionality and at the same time modifying the structure of the design. This is very error-prone as it is not clear which activity comes first and which supports the other one.

It is not easy to constrain oneself when doing a refactoring. Often the temptation is there to include this or that functionality at the current point of refactoring. But as the rules say you either refactor or add functionality. KENT BECK references this rule as the swapping of hats. At one time you can only wear one hat. Either the hat of refactoring or the one of extending. You have to complete your current task (or to abandon it) to do the other thing. And it is recommended to do the refactoring first as it contributes to making the system far easier to extend with new features. Extending a system should only add new capabilities not change existing code.

The goal of refactoring is to restructure the design of a software system to meet certain criteria which contribute to an general design and appearance that easily allows it to extend the functionality of the system.

1.3 The Benefits of Refactoring

Those criteria do not only represent the goals of refactoring but also imply a request to do refactoring when missing.

KENT BECK [Fowler, p.60] states that refactoring adds to the value of any program that has at least one of the following shortcomings:

- Programs that are hard to read are hard to modify.
- Programs that have duplicate logic are hard to modify
- Programs that require additional behavior that requires you to change running code are hard to modify.
- Programs with complex conditional logic are hard to modify.

1.3.1 Refactoring helps Understanding Code

There are also people dealing with with the program code of any system, not only machines. For them it is most important that they are enabled to understand the system easily. Therefore any system that is bound rather to be extended by people than machines has to be maintained that way. Apart from documentation and design papers (which are often lost during the lifetime of a software) the source code is one of the most important documentation form that exists for software. And no, not the comments. If the comments do only state what is done within the code they are useless, because what is done is already stated through the code. And if the code does not communicate its doing it has to be refactored to do so (*Rename Method, Rename Class, Extract Method, Consolidate Conditional Expression,...*).

The code of software should be written the way that reading it explains what is done and comments should be used to describe why this is done (the intention). Any well structured and well named program is in no need of a lot of commenting.

> When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous. [Fowler, p.88]

Software that is written with the future readers in mind (who can be anyone including oneself) who have to understand it from scratch has to be constantly updated and cleaned i.e. refactored up to keep communicating its purpose and intention. Then even without any other documentation the clear structure and the still visible design collaborating with proper naming allow an easy understanding of the whole system.

Refactoring is also of certain use when its becomes necessary to understand unknown code. With its help a piece of code that one thinks to have a particular understanding of, can be transformed (refactored) to reflect that personal understanding. If this transformed code is executed without errors when testing it, the personal understanding is validated.

1.3.2 Refactoring as a Tool to Improve the (Non)Existing Design

Most software engineers see programming as a necessary but not so important evil. It is done in the phase of implementation and most bugs are produced there. Finding these bugs then or even later is very costly (see 1.2, p.18). Most methodologies try to prevent producing bugs by paying much attention to analysis an design and define the implementation mostly as writing down the design to code.

But this is a mere theory. As the customers of most software projects don't know exactly what they want, the requirements given at the beginning of the project are only a first hint in which direction the project should evolve. As the project continues, the requirements are refined, changed or removed. With an upfront design it is hard to cope with these changes. The design must be adapted each time the requirements change. When the project has already gone into the implementation phase, it's necessary to adapt the design and then update the code to reflect these changes.

Therefore it's helpful if the used methodology supports incremental iterative development [UML+Patterns, Larman], which allows it to analyze and design only as much as needed in the current step of development and then work through the implementation and test phase to get the scope of the current iteration step working correctly.

As reality goes, it is very difficult to keep design and implementation consistent. The code that is written by the programmer tends to decay. It slowly departs from the original design, as short-term goals or wanted changes are implemented without considering the design of the whole system. The farther the code departs from its origins the harder these are to be seen in the code and the easier it decays further.

Refactoring helps to take the depreciated code back into a useful and good design, by applying the necessary refactorings step after step with a lot of testing in between.

Bad code often includes lots of duplicate code, therefore many refactorings deal with eliminating duplicate code. If a system with duplicated code has to be modified the modification must be implemented in each place the duplicate code occurs. If some code is forgotten the way is open to many bugs due to the different execution of the duplicate code. Therefore any refactoring reducing duplicate code improves the design of the system making it easier to extend it.

As bugs often hide within the depths of bad code, clearing up the design as well as the code takes them out to the shining light. By using refactoring to reestablish the structure of the design, the code gets easier to be read and code reviews are more successful. Because refactoring relies heavily on testing, it is very useful to catch bugs which were not found earlier due to little testing.

By considering all these aspects, it can be stated that refactoring not only improves the quality of the code, but takes it back to the intended or an even better design and allows others to understand it without difficulty, but it also makes programming more

efficient. This is accomplished by not allowing the software to decay, keeping its structure clear, lowering the effort needed to extend the functionality of the system and exposing bugs early by testing before and after each refactoring.

Project Management aspects of refactoring (When doing it) Except when refactoring a large project that is in urgent need of it, refactoring should not be treated as separated activity which has to be scheduled for itself. Instead refactoring should be an integrated part of program development that is done every time it needs to be done - either to adapt the design to ease adding functionality or to understand some unintelligible code or to expose a bug wrapped in the software.

The Rule of Three by Don Roberts

> The first time you do something you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor. [Fowler, p.58]

1.3.3 Code Reviews

Regular code reviews are a very useful tool in development as people tend to overlook mistakes that they have implemented themselves. Code reviews also help to spread expert knowledge throughout a development team. The suggestions and ideas which evolve during review sessions often take the process big steps forward.

Even there refactoring can help a lot. It can be used to make the reviewed code easier to understand and this broader comprehension leads to even more useful propositions. When taking these suggestions as a source for an immediate refactoring to be implemented by the reviewers, code reviews can deliver concrete results.

As this kind of review is a very intensive work, it is suggested that only two people - the author and one reviewer - do this work together to get the highest efficiency. This kind of work is even taken farther by KENT BECK as an basic principle of the Extreme Programming methodology [Beck,Xp] known as Pair Programming. This principle demands that anytime any pair of two team members is working together to solve a programming task. Because Extreme Programming also relies on the principles of *Refactoring* and *Unit Testing* as an integral part of the whole process, it is a very good demonstration of how these principles are working together to maximize the efficiency of software development.

1.3.4 Introducing Refactoring to the Management

is not that simple because refactoring is seen first as an activity that neither increases the functionality of the software nor modifies the behavior, but may take a lot of time. The investment into the future is like all long-term goals difficult to sell to any manager.

But there are certain aspects of refactoring that can impress even project managers. First of all is quality. Refactoring definitely improves the quality of software by restructuring it to reintroduce a design to the code. It also lowers the efforts needed to add functionality afterwards.

If the manager is convinced of the positive effects of technical or code reviews refactoring can be introduced as technique to immediately implement the suggestions for improvement which are the result of the review process.

As any programmer wants to create effective software as fast as possible, he should even use the benefits of refactoring if the manager does not comply. The time invested will repay itself in the whole development process afterwards.

1.4 Insufficiencies of Refactoring

MARTIN FOWLER [Fowler, p.62] devotes a section of his book to the problems arising when doing refactoring, which is for most of us a new technique.

> When you learn a new technique that greatly improves your productivity, it is hard so see when it does not apply. Usually you learn it within a specific context, often just a single project. It is hard to see what causes the technique to be less effective, even harmful.

When working with relational databases refactoring that move the data to another place are very costly to implement as the database schemas have to be modified as well. This can be partially avoided by separating the database and the application by an indirection layer, which can be modified to reflect the changes without touching the database. This layer can be introduced as you go.

If object databases help with the migration of object versions it is only an question of the additional time needed. Otherwise the migration has to be done manually.

Many refactorings deal with changing names and places of methods thereby changing the interface of the concerned class. As long as this interface is only used within your own system you have to find and change all users of the interface, there the collective ownership of code [Beck,Xp] is very helpful.

If the interface is already used by classes outside your scope it has evolved beyond an public interface to become a published interface [Fowler, p.64]. This introduces the need of keeping the old interface alive while already publishing the refactored new interface. This might be accomplished by delegation from the deprecated methods (which should be marked as such if the system allows it, an example for this is the Java Development Kit [JDK]) to the new ones.

Therefore it should be avoided to publish interfaces prematurely. Let them evolve during the development process without restraining the flexibility - needed to allow refactoring - by publishing them.

1.4.1 When it Does Not Pay Off to Refactor

Refactoring can get a great deal out of decayed software but has no magic powers. If the concerned code is neither able to compile or to run in a stable manner, it might be better to throw it away and rewrite the software from scratch. This time using refactoring to avoid the mistakes made earlier. Perhaps the system can be divided into strong encapsulated loose coupled parts (subsystems) for which the decision of rewriting or refactoring can be made independently.

Another reason not to refactor at a given moment is when a deadline is very close. Then it would take more time to do the refactoring than the deadline allows. It will be much more sensible to deliver the system as is and to delay the refactoring after the deadline. The cost of delivering a system in such a state may be very high in terms of maintenance and stability.

MARTIN FOWLER [Fowler, p.66] sees at this point no other reason not to refactor. The notion of missing time to refactor is not an argument from his point of view:

> Other than when you are very close to a deadline, however, you should not put off refactoring because you haven't got time. Experience with several projects has shown that a bout results in increased productivity. Not having enough time usually is a sign that you need to do some refactoring.

1.4.2 The Costs of Refactoring

But the costs of refactoring are differing in environments which do or do not support some basic principles used for refactoring.

On the language/environment side it depends on how well the operations on the source code are supported (see 1.7.4, p.21). But in general the cost of applying the basic text modifications should be bearable.

As refactoring relies heavily on testing after each small step having a solid test suite of unit tests for the whole system substantially reduces the costs which would be implied by testing manually (if this is possible at all).

The costs of updating documentation of the project should not be underestimated as applying refactorings involves changes in interfaces, names, parameter lists and so on. That results in the change of the overall design of the system. All the documentation concerning these issues must be updated to the current state of development. Better off are development practices like XP which don't rely that much on external documentation but instead use the source code as main input for that.

The tests covering the system need an update as well as the interfaces and responsibilities change. These necessary changes can contribute to higher costs as tests are mostly very dependent on the implementation. Because between the steps of refactorings old interfaces are first kept and later gradually extended by the new ones, all tests should run without modification within the time frame of the refactoring. But afterwards when the old interfaces are removed in favor of the new ones the test have to be updated as well.

1.4.3 Risks Involved When Doing Refactoring

As every technique that changes a running (perhaps even working) system refactoring is not immune to introducing errors. Although there are several techniques which should enable the programmer to avoid them or at least catch them early, introducing a failure with refactoring can have serious consequences.

If refactoring is not used as part of the Extreme Programming process, where it is applied throughout the development, but if it is used to clean up the code of a software system that is already in production, before adding new functionality, the consequences of introducing bugs without catching them are very severe.

Therefore refactoring should not be treated lightly, but instead done with care and the possible problems in mind. The principles of refactoring like small steps, testing after each step, doing changes in a predictable way and not mixing up restructuring and adding new functionality, should make it impossible to introduce bugs while refactoring. Nonetheless programmers are not infallible. After applying refactorings to a system and possibly before adding the new functionality, the system should not only pass the unit tests required but also a regressive functional test which should show no difference (except perhaps speed) to the last run.

Even if there were errors introduced while refactoring the system it will be much easier to track them (once they showed up) as the system is well structured with no duplicate code, clean hierarchies and small units.

1.5 Refactoring vs. Design

It may sound unfamiliar but MARTIN FOWLER puts refactoring as a kind of complement to design, not the design in general but the generally accepted upfront design which has to

- occur before the implementation phase
- to cover all eventualities
- to provide the most possible flexibility

and which is based quite a lot on predictions.

Upfront design has its flaws. People are not very well at predicting the future especially when the requirements might not be that clear. The only possible way to deal with these problems is to design very flexible systems which can be extended in every imaginable direction. These systems tend to be very complicated implying they are difficult to understand and to maintain. If a future change happens to be in the right direction all goes well, but if not it's often a horror to modify the complex structure. And what about all of the changes that simply don't happen? A lot of flexibility created for what? Nothing.

Upfront design has its benefits as well. One can create a design and writing the system gets "just" being a craft. Design can provide a general first, accepted solution which covers the available requirements. Using that design to implement the software provides you with a running application that is already able to generate revenues for the customer. Although it is not flexible enough to incoming future requirements or changes it has the ability to grow.

A benefit of a distinct design phase is the possible division of work. Professional system analysts and designers (e.g. consultants) can be asked to design the system with their experience and abilities. But even those experts don't have the skill of clairvoyance. Therefore a thorough system analysis is needed which must be based on a complete catalog of requirements.

1.5.1 Iterative, Incremental Development

In contrast to the traditional processes (like "Waterfall") the incremental, iterative development process executes all the phases (analysis, design, etc.) more than once. Craig Larman introduced such a process in his book "Applying UML and Patterns". As the author has written a very focused introductory section and I can't say it better than himself, I'd like to quote the lines from his book [UML+Patterns, Larman].

> An iterative life-cycle is based on successive enlargement and refinement of a system through *multiple* development cycles of analysis, design, implementation and testing.
>
> The system grows by adding new functions within each development cycle. After a preliminary *Plan and Elaborate* phase, development proceeds in a *Build* phase through a series of development cycles.
>
> Each cycle tackles a relatively small set of requirements, proceeding through analysis, design, construction and testing (see 1.1, p.17). The system grows incrementally as each cycle is completed.
>
> This is in contrast to a classic waterfall life-cycle in which each activity (analysis, design, and so on) is done *once* for the entire set of system requirements.

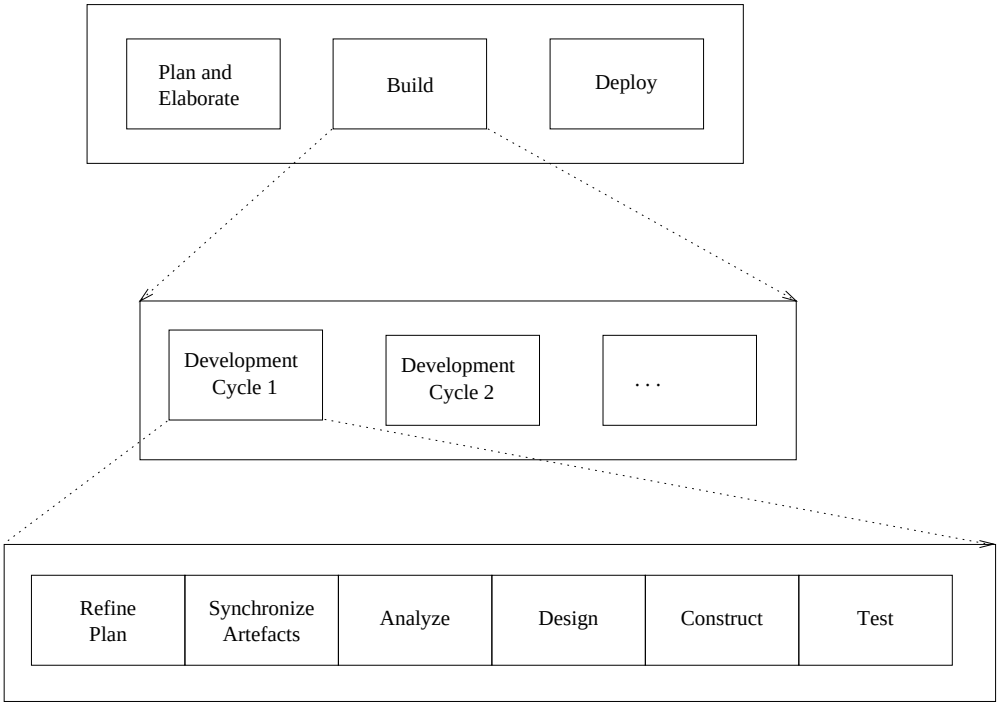

Figure 1.1: Iterative Development Cycles

Advantages of iterative development include:

- The complexity is never overwhelming.
- Early feedback is generated, because implementation occurs rapidly for a small subset of the system.

The software development process that is described by Craig Larman is a kind of intermediate between the traditional Waterfall model and the Extreme Programming method.

It already focuses on incremental analysis and design to avoid the high complexity that is introduced by a large upfront analysis and design phase and to allow the adoption to changing requirements.

He also shows the possibility to defer the consideration of requirements to later phases. This allows the reduction of the scope that has to be handled in the current iteration cycle with its fixed time-frame. The Exteme Programming method takes this approach even further.

> The recommended middle-ground strategy is to quickly create a rough conceptual model where the emphasis is on finding obvious concepts expressed in the requirements while deferring a deep investigation. Later, within each development cycle, the conceptual model is incrementally refined and extended for the requirements under consideration within that cycle.

This first conceptual model shall cover the architectural questions as the development of the system must be based on a solid foundation of architecture.

But he also stresses the importance of a detailed analysis and design to reduce the risks involved in developing a software system. He uses the exponential cost curve (see 1.2, p.18) to explain the higher costs of making changes in later phases of development (e.g. implementation) - “software is ‘harder’ than it sounds”.

The addition of functionality is done by adapting and changing the design and later modifying the code to resemble the actual design.

The application of refactorings to the code created and modified during the *construction* phase doesn't seem to be necessary, because the updated design should incorporate the changes needed for the current development cycle. But Craig Larman does not forget that "during programming and testing myriad changes will be made and detailed problems will be uncovered and resolved." [UML+Patterns, Larman, p.297] When implementing these changes refactoring can be used to provide a smooth transition from the existing design to a better suited one.

The changes made during the *construction* phase are integrated in the design of the next cycle while the *synchronize artifacts* phase

The disadvantage of the incremental process are the too high expectations of the customer and managment because of early visibile (user interface) prototypes or releases created.

1.5.2 Extreme Programming

The approach taken by Craig Larman is even taken further by the Extreme Programming method Kent Beck devised. The main difference to the incremental process discussed above is that the burden of a extensive upfront design is reduced to the most basic needs.

The real "design" occurs when writing the code to fulfill the Engineering task at hand which is a part of a greater user story (much like an use case). Then refactoring is used to establish the design needed for implementing the functionality at issue. Implementing the functionality after refactoring adds further details to the design implicitly created.

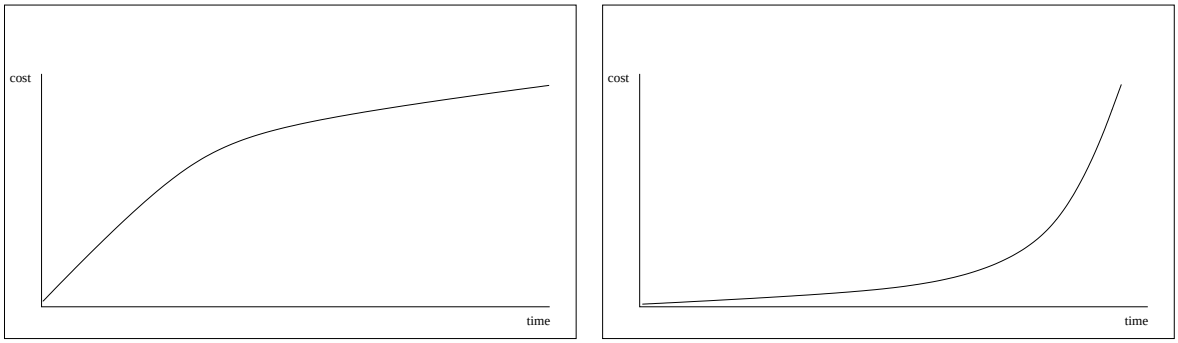

Figure 1.2: Comparison of Cost Curves of Extreme Programming and Conventional Development Process

The Extreme Programming [Beck,Xp] methodology uses the notion of a flat cost of change curve vs. the exponential cost of change curve used in traditional software engineering. Therefore it does not matter that much if decisions are delayed if they are not important enough or if they are to costly to implement right now.

With that advantage in mind, it's possible to start with a very simple solution. This first result has to be only as flexible as necessary, because it can be extended without a big effort. Refactoring helps further leveling the expenditure by constantly keeping the software in a clear structure reducing the hard work needed for the future extension.

Extreme Programming uses a initial small upfront design of the simplest solution for the basic requirements to create a running system. Using the principles of this methodology, the programmers work (in pairs) with the code to improve the design by using refactoring

when its necessary for adding functions, gaining simpler solutions or just cleaning up the code.

As refactoring does not change any behavior of the system, it must not affect the running version which is often already useful for the customer included at the development site. As he chooses the next most valuable functions/requirements/stories which have to be added, he steers the direction in which the project does evolve in small steps - every time able to change the necessary direction.

A Metaphor for XP KENT BECK [Beck,Xp] also introduces the metaphor of car driving when explaining why it is more useful to use an incremental approach instead of heavy upfront design.

> Driving is not about getting the car going in the right direction. Driving is about constantly paying attention, making a little correction this way, a little correction that way.
>
> This is the paradigm for XP. There is no such thing as straight and level. Even if things seem to be going perfectly, you don't take your eyes of the road. Change is the only constant. Always be prepared to move a little this way, a little that way. Sometimes maybe you have to move in a completely different direction. That's life as a programmer.

1.5.3 Refactoring Is Analyzing

I found this very interesting statement by WALDEN MATTHEWS on the relationship of refactoring to the development process on the WiKi Pages:

> When you are refactoring, you are analyzing. The right time to do it cannot come from some process manual or methodology - it's when you see an opportunity to express something more powerfully and simply. The project phase and the medium are unimportant. The people, their skills, and the project context are important. And it's far more important to do it when you can than to quibble about the ideal time to do it....
>
> I fancy I'm following my own advice here, by pointing out the commonality between code refactoring and other forms of analysis. This is the essence of systems development, no matter what name you give it.
>
> [Wiki, RefactorMercilessy,WaldenMathews]

1.5.4 Refactoring Adds Flexibility

Flexibility is one thing that has to be payed for in traditional development by anticipating the flexibility needed and incorporating it in the design of the system. On the other hand the flexibility of not being constraint in the direction of development comes with refactoring. MARTIN FOWLER [Fowler, p.68] puts it this way:

> Refactoring can lead to simpler designs without sacrificing flexibility. This makes the design process easier and less stressful. Once you have a broad sense of things that refactor easily, you don't even think of the flexible solutions. You have the confidence to refactor the time comes. You build the simplest thing that can possibly work. As for the flexible, complex design, most of the time you aren't going to need it.

1.6 When its Necessary to Refactor

As refactoring should be seen as a basic principle of programming, it does not require a special methodology. It can be used in the implementation phase every time it is necessary. But it can be supported in various ways. Basically experience in programming helps a lot (especially in object-oriented-programming). With that experience you are able to see the places where refactoring would make the difference. KENT BECK and MARTIN FOWLER name these parts of the code where refactoring is urgently necessary "bad smells in code" [Fowler, p.75]. A number of the smells they refer to are listed below:

Duplicate Code main reason to refactor

Long Method inherited from procedural programming

Large Class too much done by a single class

Long Parameter List are no longer necessary when working with objects

Divergent Change one class is commonly changed in different ways for different reasons

Shotgun Surgery changes influence to many classes and methods

Feature Envy too much interest in other objects data

Data Clumps data that is used together everywhere would make a great class of its own

Primitive Obsession use classes in addition or instead of primitive data types

Switch Statements object orientation has other ways to deal with actions depending on types

Parallel Inheritance Hierarchies sometimes useful but often unnecessary

Lazy Class A class that isn't doing enough to pay for itself should be eliminated.

Speculative Generality don't invest to much in flexibility for the future

Message Chain hard couple the client to the structure of navigation

Middle Man should be removed if delegation is all he does

Inappropriate Intimacy restricting the knowledge of the internals of other classes

Incomplete Library Class must sometimes be extended to add wanted functionality

Data Class should get additional tasks which deal with its data to raise its importance

Refused Bequest if subclasses use only very little of what they are given by their parents

Comments a comment is a good place to say *why* you did something not what you did

1.7 The Craft of Refactoring, or what helps doing it

But experience is not everything, often your mind is clouded with the wisdom you have and you don't even smell these bad smells. Then you are fortunate if you have a programming partner that helps you by opening your eyes or by doing it himself.

1.7.1 Patterns

The use of refactoring is also eased if you are familiar with design patterns. As patterns reflect expert knowledge - solutions for design problems in a particular context, they point to a direction which a refactoring can take. For instance the *Replace Conditional with Polymorphism* refactoring is supported by the strategy or state patterns [Gang of Four]. Other design patterns embody more the goal to which the refactoring should strive. Like the *Composite* pattern which can be (accidentally) reached by using *Extract Superclass* and *Pull Up Method* on a set of classes with common methods.

1.7.2 Extreme Programming, XP

Having understood the principles of Extreme Programming by KENT BECK [Beck,Xp] does also help a lot, even when you are not able or allowed to develop with this demanding and impressively simple methodology. Refactoring is one of the principles of his approach because it allows to improve the design of a software gradually without compromising its functionality. This supports the notion of incremental design, with only the simplest solution actually in place, as this costs as less as possible and generates the highest value. Moreover it allows changes in any direction if wanted or needed.

1.7.3 The Hard Work

When using refactoring, unfortunately a lot of the work that is needed to be done are stupid search and replace operations and cut'n'paste with some corrections applied afterwards. Although you have to decide which refactoring could improve your code at this place and time, implementing the refactoring is quite boring work. In order to reduce the errors created when doing refactoring, it is the best way to introduce the changes in a ordered manner using step-by-step instructions, which should be each followed by the run of the automatic unit-test-suite.

Using an IDE that supports enhanced search and replace functionality is very useful but even that does not make refactoring easy enough to really use it often and regularly. It takes too much time to rename a field or method with a manual search and replace operation as you have to find all calls or references to the item. If the method is overloaded, i.e. the same method name is used with various parameter lists, it has to be decided which occurrences have to be replaced and which not.

1.7.4 Language Advantages

If the programming language is not based on source code text files as the basic place to keep the code but instead uses a database or a repository to hold the tokens, it gets easier to apply refactorings. In the repository each element of the programming language can be addressed by itself. There is no danger of mixing up different elements with the same name. The references can also be updated immediately because they are represented as data structures within the repository. Smalltalk uses such an approach, other languages support it as well. For C++ and Java it is possible to use the IBM Visual Age development environment which is based on the principle of keeping source code elements in a repository.

1.7.5 Refactoring vs. Rewriting

If rewriting is the hard way of changing a system, refactoring is the soft one. It even makes rewriting superfluous for some parts of the code as it uses the available code to engineer a better version:

> Refactoring is the moving of units of functionality from one place to another in your program. It encompasses things as simple as renaming methods, and as complex as adding a helper class, moving methods to classes where they better belong, and creating subclasses and superclasses to reduce the overall amount of code in the system. Refactoring has as a primary objective, getting each piece of functionality to exist in exactly one place in the software. Once that has happened, if there is something wrong with a particular method, there's just one method to "rewrite". What emerges is that continuous refactoring makes most rewriting unnecessary. [Wiki, RefactoringAndRewriting]

Chapter 2

The Development Of Refactoring

2.1 Looking at the History

The history of refactoring starts a lot earlier than one would think. Refactoring is a common practice of programmers who have to deal with bad code. Well structured code is easier maintained and extended, there is no way out of this. But like Patterns it is one of the hidden practices or better implicit knowledge that has to be digged out and named explicitely to take its value to other developers.

The first people who seem to have explicitely caught its importance were Ward Cunningham and Kent Beck who worked with Smalltalk from the 1980s onward [Fowler, p.71]. Those two eventually worked on creating a software development process which integrates refactoring as a basic principle [Beck,Xp].

Another man with strong interests in refactoring, especially of frameworks, was Ralph Johnson, a professor at the University of Illinois, whose students shared his interests and took it even further.

But the first scientific exploration of the principles which support refactoring, was done by one of his students. William Opdyke [Opdyke, Thesis] examined the theories of the preservation of the semantics by applying refactorings. He also provided a first list of refactorings, which is referred to until now. It was his intention to show the possibility of creating a tool that can implement chosen refactorings by itself.

His ideas and the refactorings he prove to be semantics-preserving were taken by John Brant, Don Roberts and Ralph Johnson further on the way [TAPOS] to finally create the Refactoring Browser [Refactoring Browser] which is up to date the most significant tool for (Smalltalk) developers who want to refactor their code.

2.1.1 Catalogs of Refactorings

Besides the tool idea another successful approach was developed to help developers with the application of refactoring in the least difficult way.

The development of a catalog of refactorings, which is to be found in his [Fowler] book, and also online, was the intention of Martin Fowler. Such a catalog does not only lists the currently used and well tried refactorings by name but also introduces a standard notation.

This standard notation makes it easier for anyone to catch the main idea of the refactoring, follow the example supplied and the use the step-by-step instructions to apply the refactoring without introducing bugs.

Unfortunately the idea of a standard notation was not that successful when it was applied to design patterns. Anyone who thought he could contribute to the area of design patterns, thought also of a better notation for writing them down as the original notation by the Gang of Four [Gang of Four] didn't seem to be sufficient.

The notation for refactorings consists of the following elements:

a name for building a vocabulary much like the pattern one

a short summary of when the refactoring is needed and what it does, including a description of the problem, a reflection what one does and mostly a sketch either in code (before vs. after) or as an UML diagram

the motivation describes why the refactoring should be applied and when not

the mechanics are a step-by-step description of how to carry out the refactoring, short notes which can be easily referred to

the example should show a very simple application of the refactoring

With the possibilities of the Internet such a catalog can be easily made available, maintained and extended by anyone who thinks he can contribute to the collection of refactorings, which is also encouraged by MARTIN FOWLER.

MARTIN FOWLER [Fowler, p.107] is aware of the fact that his catalog is by now no complete collection of sensible refactorings:

> As you use the refactorings bear in mind that they are a starting point. You will doubtless find gaps in them. I'm publishing them now because although they are not perfect, I do believe they are useful. I believe they will give you a starting point that will improve your ability to refactor efficiently. That is what they do for me.

2.2 Tools Needed

2.2.1 Thinking About Tools

Applying refactorings such as *Extract Method* requires additional thoughts, but which tend to resemble each other when repeating the refactoring. Doing all the refactorings by hand, can get very boring. The solution for this problem is to create a tool that helps implementing most of the refactorings. This is done by checking the constraints implied by the refactorings, analyzing the syntax and semantics (e.g. for transforming local variables or conditionals) and presenting the results in a convenient way. Then the user is able to choose whether to use the refactoring or not and implement it with click of the mouse. Then the tool does all the modifications necessary. Afterwards the unit tests are run automatically and if the results are correct the user can proceed to the next refactoring.

It would be great if such a tool could also provide the ability to autonomously decide which refactorings would structure the code in a better way and implement them itself. But as stated by MARTIN FOWLER and KENT BECK [Fowler, p.75], it is a very intuitive process to decide if refactoring is necessary and which refactorings could contribute best to increase the quality of the code.

> One thing we won't try to do here is give you precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed

> human intuition. What we will do is give you indications that there is trouble that can be solved by refactoring. You will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines.

Therefore a tool like the one described above is only a helper in applying refactorings but a very powerful one. It reduces the costs of doing refactorings in scope of time, cost and effort to enable the programmer to refactor efficiently without much thinking about productivity losses. It also contributes to instantiate refactoring as an integrated practice within the implementation.

2.2.2 History of Tool Development

The begin of the research done for developing such a tool was in 1992, when William Opdyke presented his doctoral thesis "Refactoring Object-Oriented Frameworks" where he contemplated the theoretical fundamentals needed to create an automatic tool which is able to implement refactorings in a safe manner.

Some time after that John Brant and Don Roberts created the "Refactoring Browser" for the Smalltalk language that supports auto-magically refactoring the parts of the code which are in need of it.

By now several tools for other programming languages as Java or C++ are available such as JRefactory (Java), IntelliJ Renamer (Java), Xref-Speller Plug-in for Emacs (Java, C++, C) and Guru (Self). Unfortunately none of them is as complete as the Refactoring Browser for Smalltalk.

2.3 A Short Look on Some Tools Available

2.3.1 Refactoring Browser (Smalltalk)

The Smalltalk community was the first one which used refactorings consistently during the development process and many of the contributors like Ralph Johnson, Kent Beck, Don Roberts, Martin Fowler et.al. originated from there, the development of the Refactoring Browser was only a logical step within the development. Fortunately the Smalltalk programming environment provides ideal conditions for implementing such an tool, as is later cited in this section.

Because of my shortcomings - I do neither have access nor knowledge of the Smalltalk programming language - I am not able to provide firsthand information about the tool but rather cite the authors [TAPOS] first as well as an enthusiastic user [Wiki, RefactoringBrowser-Kent Beck]:

Since refactoring occurs at all levels within the software development life cycle, the ability to perform refactorings automatically is crucial to software evolution.

The Smalltalk Refactoring Browser is a tool that carries out many refactorings automatically, and provides an environment for improving the structure of Smalltalk programs. It makes refactoring safe and simple, and so reduces the cost of making reusable software.

The goal of our research is to move refactoring into the mainstream of program development. The only way that this can occur is to present refactorings to developers in such a way that they cannot help but use them. To do this, the refactoring tool must fit the way that they work. This goal imposes the following design criteria on the Refactoring Browser:

- integrated into the standard development tools e.g. the browser in Smalltalk

- must be fast. Smalltalk programmers are used to being able to immediately see the results of a change. ... Therefore, refactorings that take a long time to perform an analysis will not be used.

- avoid totally automatic reorganization. e.g. doesn't generate names (which would have no meaning in the user domain) but prompt for them

Another of the key design criteria is that we assume that there is an intelligence directing the refactoring process. ... It has been our experience that systems that perform automatic code reorganization are of little utility since they typically are algorithmic and have no understanding of the problem domain.

...

Our approach is to provide a tool that will search for places in your program where code is duplicated, or unused, and point them out. This allows a human to make the final call whether or not the code should be consolidated and to provide a semantically meaningful name.

...

There are some features of the Smalltalk language and the development environment Visual Works that allow an easier application of refactorings:

- ... reflective facilities, that is, the ability for the language to examine and modify its own structures. ... Refactoring can be performed in the absence of reflective facilities, but then requires a separate, metalanguage that can be used to manipulate the original program. Having a single language simplifies the process.

- ... access to an easily manipulable representation of the program (e.g. the parse tree), which is provided by the VisualWorks environment; ... the transformations that refactorings require are much easier to implement as parse tree to parse tree transformations rather than string to string transformations. Other dialects of Smalltalk do not provide this level of access to the system, which is one of the major reasons the Refactoring Browser has not been ported to other environments.

- ... To ensure that the behavior of a program does not change, every refactoring has associated with it a set of preconditions that must be met in order to apply the transformation. ...

  The static properties of the program must be analyzed to determine if these preconditions are satisfied before performing a refactoring. ... Since we reused many of the existing static checks present in the compilation framework, the refactoring framework is much simpler than it would be if we had implemented all of these checks ourselves.

- ... The Refactoring Browser uses method wrappers to collect runtime information. These wrappers are activated when the wrapped method is called and when it returns. ... As the program runs, the wrapper detects sites that call the original method. Whenever a call to the old method is detected, the method wrapper suspends execution of the program, goes up the call stack to the sender and changes the source code to refer to the new, renamed method. Therefore, as the program is exercised, it converges towards a correctly refactored program. ...

  The major drawback to this type of refactoring is that the refactoring is only as good as your test suite. If there are pieces of code that are not executed, they will never be analyzed, and the refactoring will not be completed for that particular section of code.

...

It is intended by the authors of the tool to extend it to enable users to do their refactorings even quicker and without reluctance:

- the next step on the scale of refactorings are bigger ones which are composed of the well known basic refactorings which were first introduced by WILLAM OPDYKE [Opdyke, Thesis], the application of composite refactorings has the advantage that after performing one refactoring certain postconditions become true. These postconditions can be used without expensive analysis to satisfy preconditions of later refactorings. Therefore, refactorings are sometimes easier to perform in a composition than they are independently. Developing a system that incorporates these batch refactorings is currently being researched.

- Every creator of an API or open library has to deal with problems originating from the evolution of his product, which does also affect the interfaces to the outside world. The authors of the Refactoring Browser want to enable their users to have a partially automatic refactoring of older software to enable it to use the updated interface of the concerned API.

Refactoring is a common operation in the software life cycle and the Refactoring Browser provides automatic support for many of the common transformations that come up in Smalltalk development. The Refactoring Browser is a practical tool in that it can perform correct refactorings on nearly all Smalltalk programs. In fact, we regularly use the Refactoring Browser on itself. That is, we use the tool to refactor its own source code. Additionally, the Refactoring Browser has been used to help develop a wide range of frameworks from the HotDraw framework for graphical editors, to financial models being developed by Caterpillar, to prototypes and models for a major telecommunications company.

Now KENT BECK shares [Wiki, RefactoringBrowser-Kent Beck] his experiences with the Refactoring Browser:

> This is absolutely the greatest piece of programming software to come out since the original Smalltalk browser. It completely changes the way you think about programming. All those niggling little "well, I should change this name but..." thoughts go away, because you just change the name because there is always a single menu item to just change the name.....
>
> The best thing about Refactory is how safe it is. As long as you don't manually edit the source code, you are nearly guaranteed (modulo things like choosing the wrong class for a "move to component") that you won't change the semantics of the program. The more I use it, the more aggressive I am slamming logic around until it makes sense.
>
> Changing names is the least of its tricks. Some others are:
>
> - *Extract Method* – make a sub-method out of the selected text. If there is already an equivalent method, optionally invoke that instead.
> - *Inline Method* – put the invoked code in place of the invocation. This even works for methods in other classes.
> - *Move To Component* – move the code for a method to another class and invoke it

Some other cool tricks:

- *Add Parameter* – add a parameter to every implementor of a message, and to every invocation of the message (with a default value)
- *Remove Parameter* – if no implementor of the message uses the parameter, remove it from the methods and the invocations
- Cross referencing from inside the source code – select any program element in the text and you get a choice of several specialized browsers– senders/implementors of a message, readers/writers of a variable
- *Rename* – you can rename classes, variables (all types), and messages
- *Abstract/Concrete Instance Variables* – make all references to an instance variable go through a message, or make all references direct

With unlimited undo, you can bravely try experiments that might not pan out.

2.3.2 JRefactory (Java)

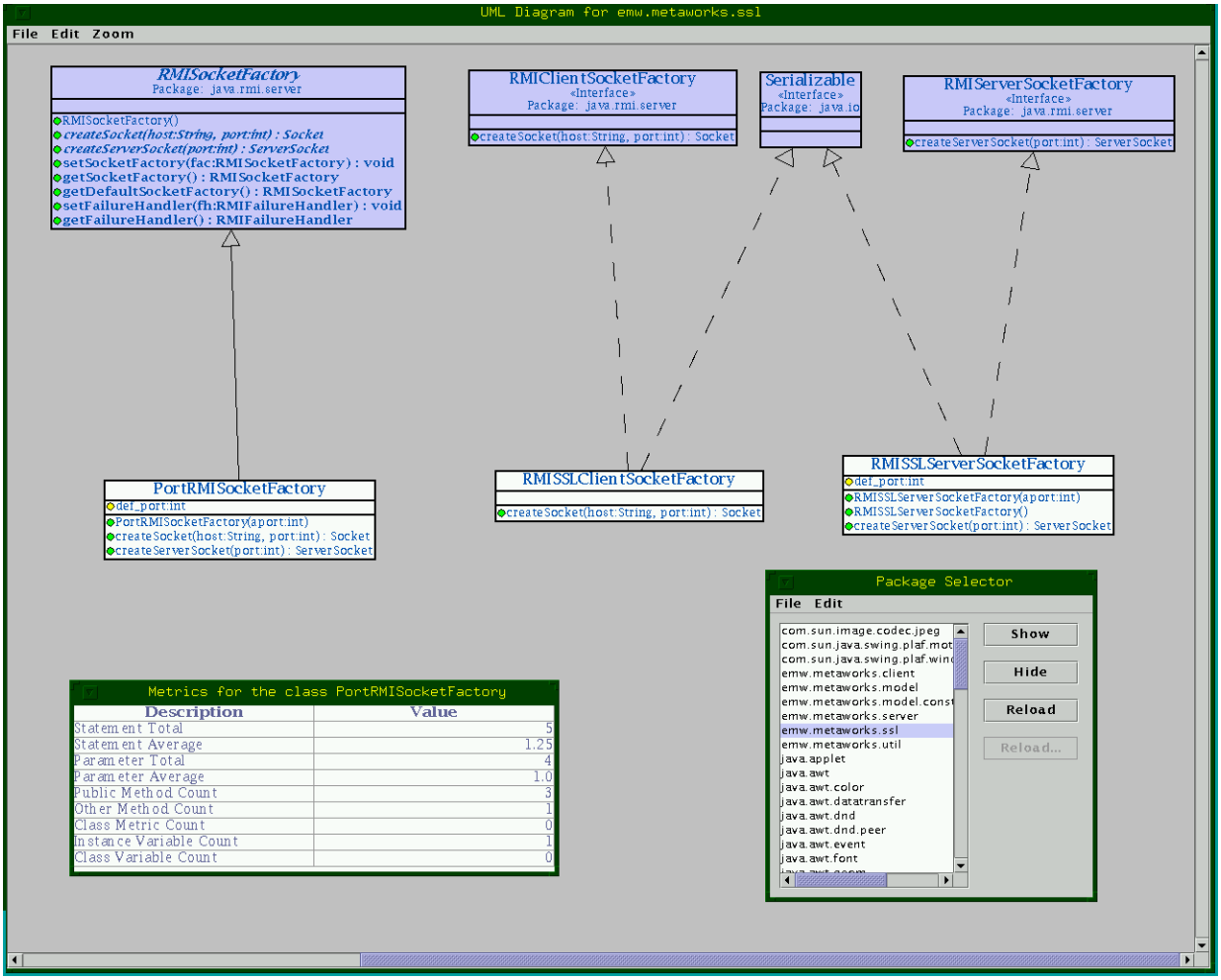

Figure 2.1: JRefactory UML Interface, including Metrics and Package Selector

JRefactory [JRefactory] is a tool provided under the GPL written in Java by Chris Seguin to allow easy application of refactorings by implementing a user interface that is

based on UML diagrams as visualization of the concerned Java classes. It can cooperate with the JBuilder [JBuilder IDE] and Elixir [Elixir IDE] IDE's but also be used in a command line variant. When using with an IDE the tool supports switching to the appropriate code line for a method or field of a class.

These diagrams are compiled out of the Java source files. The incorporation of the supplied source files of the Java Development Kit [JDK] is also required (it takes a long time). The user interface allows zooming (in certain steps), moving classes and changing association lines.

Refactorings are applied by selecting the class(es) to be modified and selecting the appropriate refactoring in a pop-up menu.

The program supports the following refactorings:

- *Repackage or move class*
- *Rename class*
- *Add an abstract parent class*
- *Adds a child class*
- *Removes a class*
- *Push up field*
- *Push down field*
- *Push up method*
- *Push up abstract method*
- *Push down method*
- *Move method*

A feature that is more interesting than really helpful are the metrics supplied. It is possible to show the metrics for a package or a class, including absolute and average numbers of classes, methods, statements each averaged for the higher structural units (e.g. average statements per class, or per method). The metric information highlights problem classes and methods with with either to much methods or to much statements per method. See code smell LongMethod (see 1.6, p.20). Unfotunately the averaged numbers are shown with up to 10 decimal places which does more hindering than helping. I think it would be helpful if the numbers that exceed certain ranges (low and high) were highlighted.

When testing the program the limited possibilities of interfering with the user interface showed up to restrict its usefulness. As the classes of a whole package are shown in one UML class diagram, it easily gets confusing because there are too many classes shown at one time. The fixed step zooming possibilities are of no help either. The class names can only be read at the highest magnification and have to be guessed at the other steps. No class can be hidden to allow focusing on problem areas and packages with many classes slow down the user interface very much.

The classes are arranged in a very inconvenient way - they are just lined up horizontally. It would be very useful if the tool would incorporate the functionality of any UML

diagram editor that is required for useful operation on the diagram (e.g. continuous zooming, automatic ordering, possibility to hide classes/methods per diagram or per class or template, etc.) or even better if if was incorporated in a UML tool itself.

It also produced several exceptions partially originating from the JDK-classes. The refactorings which were applied to the test package worked quite well. The newly implemented *Move Method* refactoring was everywhere disabled, therefore it could not be tested. The *Renaming* worked only with classes. When refactorings were applied to the test package they didn't show up on the diagrams, not even after reloading them. Only after exiting the tool and restarting it, the applied refactorings got visible.

As the tool does not operate on source code level, only general restructuring refactorings can be used. Therefore it would be necessary to use another tool to do the source code level refactorings as *Extract Method* or *Decompose Conditional*.

2.3.3 IntelliJ Renamer (Java)

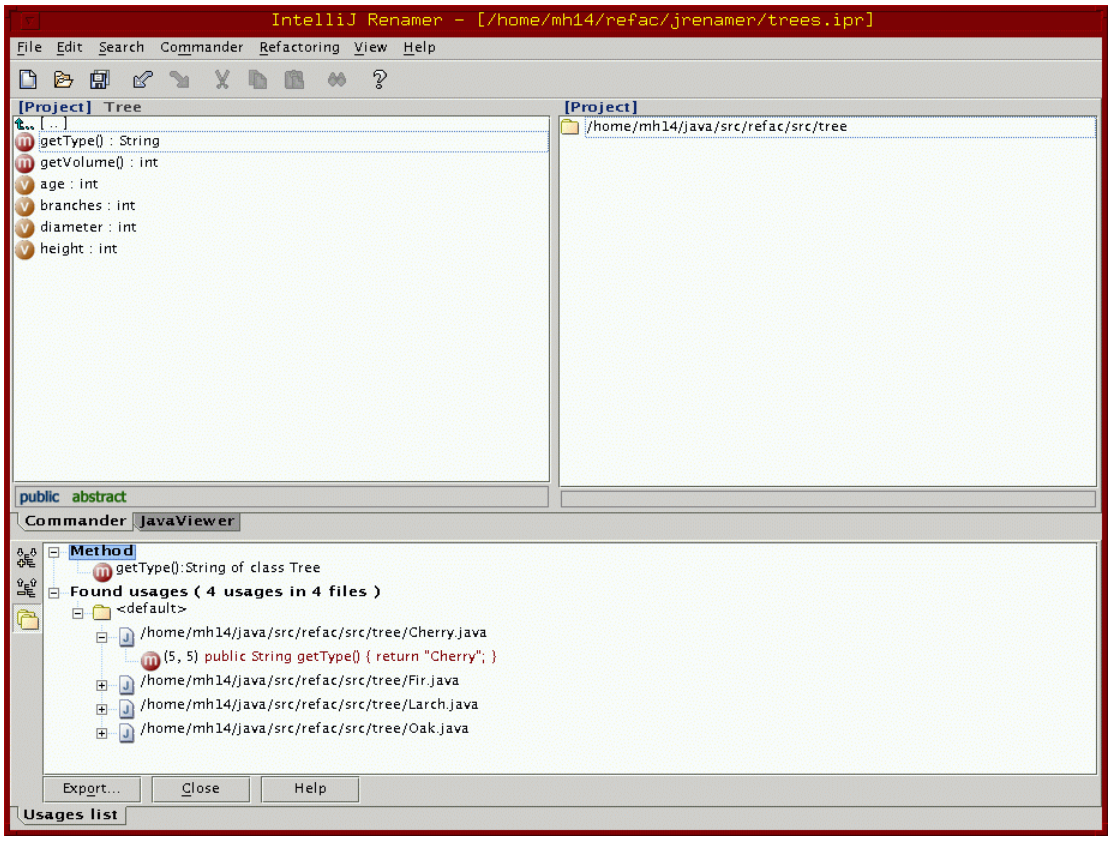

Figure 2.2: IntelliJ Renamer, Search Results for a Method Search

The IntelliJ Renamer [Renamer] is a commercial product which intends to support developers who have to do a lot of renaming, e.g. in order to make their code more communicating. It was created by IntelliJ Software which is located in Prague.

The tool focuses on finding uses and dependencies of packages, classes, methods and variables and offers the possibility of renaming. The general approach is to list the found occurrences and references of the symbol to be renamed in a comprehensive list. There each occurrence can be excluded from the renaming and the list may be exported to a text file. After checking the list the renaming is invoked for all occurrences on this list.

It allows to *move packages and classes* to other packages as well but no moving of variables or methods.

Another useful functionality covers the updating of software systems to changed API's named migration. The migration feature is used by specifying number of renamings of classes and packages which is processed at once for a whole project and can be repeated. As an example the migration map for the changes of the swing package structure (i.e. com.sun.java.swing.* to javax.swing.*) is supplied.

When using on the test classes the occurrences of the searched classes, variables and methods were correctly found. The renaming of classes worked as well including the changes to import and extend statements. But when moving classes to another package, the package statement was included at the end of the file which does not comply to the java specification and is a serious bug. The import statements for the moved class are generated in the right way.

Moving features are only available to classes and packages. The number of classes to be moved to a new packages is restricted to one at one time.

The Renamer offers unlimited undo. It worked in most cases but sometimes created an exception which put the system in a inconsistent state. After undoing a class move the code views did not reflect the change but had to be closed and reopened to do so.

Generally, it may be stated that the tool supports renaming in a very useful way (as its name already suggests) but has limited use elsewhere. Finding occurrences and references of symbols is the main focus of the tool, which is performed with a lot of options with high speed and correctly.

It shall also support a synchronization feature that allows it to react on changes to the source files done by external editors. I was not able to test this functionality because my trial license ended after a fortnight. Therefore I quote the web site [Renamer]:

> Renamer can be used along with your favorite IDE or editor, because it automatically synchronizes with all external source code modifications you make. It also automatically saves all changes when you switch to other applications.

2.3.4 Xref-Speller for Emacs (Java,C++,C)

The xref program was developed by MARIAN VITTEK [Xref-Speller] to support extending source browsing and symbol completion functionality under a variety of development environments. Later refactoring functionality was added. It seemed to be no problem at all because the source of the project to be refactored is already available as a parse tree to the xref program. And similar to the RefactoringBrower [TAPOS] it is much easier to implement refactorings to a parse tree than to pure source code.

Actually the Xref-Speller (see 2.3, p.32)is just a front end for (X)Emacs and Kawa for the tool xref which runs as a separated process and communicates with the front end.

By now the Java and C language are supported by xref. The C part of the program was tested with the linux kernel sources (about 1.5 million lines of code). It includes a full featured preprocessor needed for parsing the preprocessor instructions of C-programs. The creation of the parse tree for the Java language is not done by parsing the source files but rather by decompiling the byte code of the class-files generated by the java compiler, as there the type information (e.g. symbol types, function signatures etc.) needed is already verified and non-ambiguously stored by the compiler.

The xref tools supports the notion of projects and allows the user to include/exclude specific directories to use. The needed files are parsed by an update function into a cross-references file which can be updated either by reexamining all files or by updating only

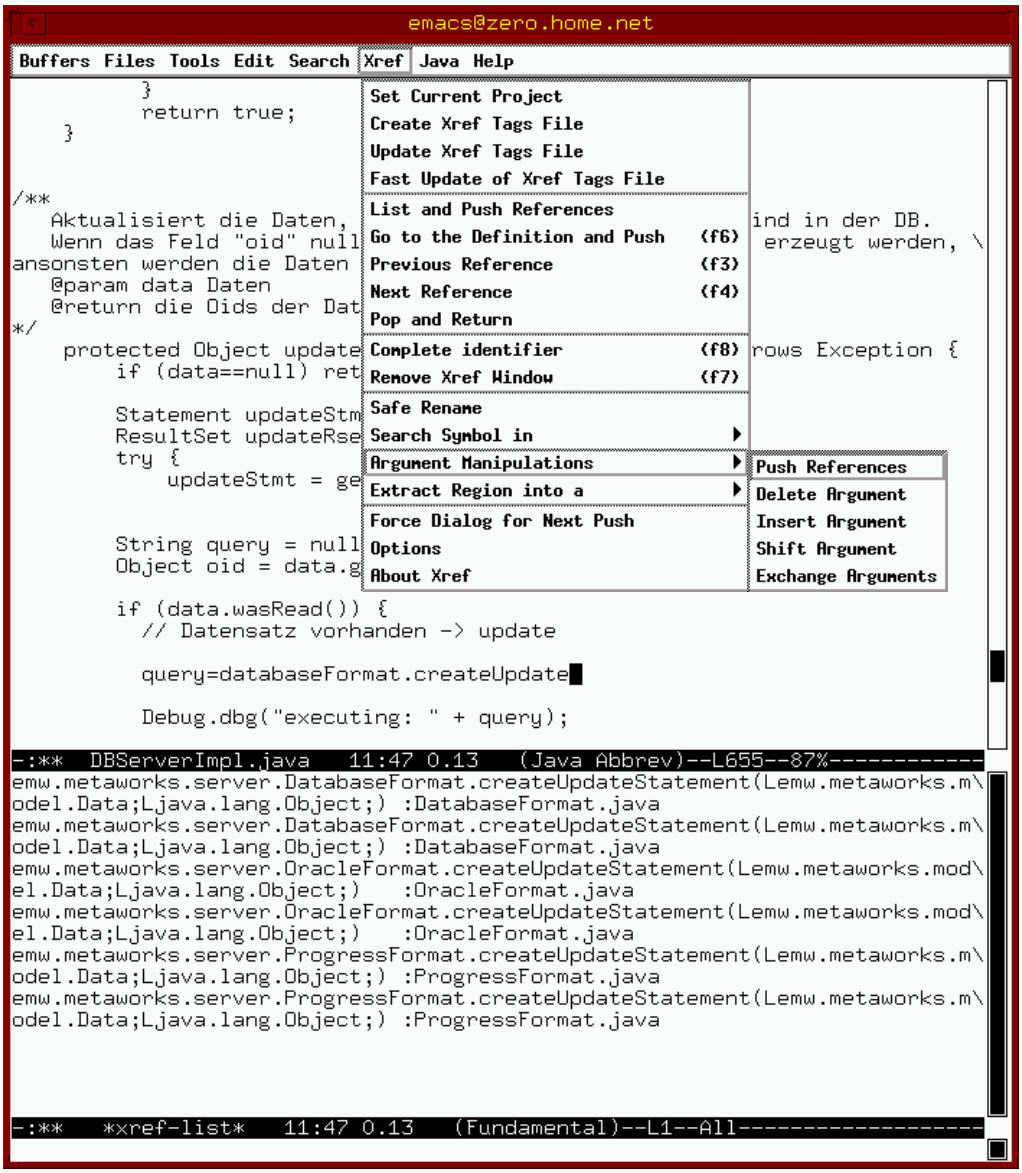

Figure 2.3: The Xref-Speller Interface within Emacs

modified files. This cross-references file is used for all operations as it includes a parse tree of the projects classes.

The refactoring currently implemented in xref are the extraction of marked code into a new method, inclusively creating a call to the new method at its old place. This functionality can be used to do the refactorings *Extract Method, Consolidate Conditional Expression, Decompose Conditional.*

Another refactoring supporting functionality is the modification of function parameters to do *Add Parameter, Remove Parameter, Parameterize Method.* Unfortunately the implementation of Insert Parameter lacks the ability of the Refactoring Browser to supply a default value and also does not care about the use of parameters within the method body which were deleted from the parameter list.

The renaming refactorings can be achieved by using the renaming possibilities offered by xref. As the renaming occurs at the parse tree level, it is not subject to unintendedly renaming the wrong occurrences (depending on type, method signature).

Xref is also able to generate html files for all the source code of a project, including references between all the generated files at each occurrence of variable, class or method names to the correct source of the declaration. This is very helpful when parsing through the static inheritance and association structure of the project.

In general it has to be said that the Xref-Speller is a very useful tool for users of (X)Emacs/Kawa, which allows to reduce the tedious work connected to the application of the refactorings mentioned. Besides the parsing and completion functionality at parse tree level outperforms the ctags functions available within Emacs in scope of correctness.

There are also a lot of possibilities how to extend the xref program which would make it even easier to browse the static inheritance and association structure of a project and allow the user to apply more refactorings without much manual intervention.

Chapter 3

Refactoring an Example

3.1 Refactoring Examples in General

Refactoring a piece of code is a very extensive process which contains many small steps with a lot of testing in between. Therefore it is very complicated to show an example refactoring within the limitations of a written work (e.g. this paper).

The most reasonable approach would be:

1. listing the existing code,
2. pointing at the places where the code smells claim refactorings to be applied,
3. explaining the choice, which refactoring should be used,
4. looking at possible problems occurring by refactoring,
5. writing down unit tests (if they don't exist yet), for checking the current functionality,
6. listing the unit tests and mentioning possible places where they could break,
7. applying the refactorings step by step,
8. thereby listing the code fragments which have changed (highlighting small changes and describing big ones),
9. running tests in between the steps, when they break the refactoring has to be undone and redone in smaller steps,
10. commenting on that,
11. after the refactoring was completed, the modification should be summarized to which new methods and classes were created, which (re)moved, etc,
12. the complete new code should be listed as well, highlighting the changes in the whole picture

It is understandably that such an extensive visualization would require too many pages, even for applying just one simple refactoring. As Martin Fowler [Fowler, p.1] puts it:

> I tried and even a slightly complicated example runs to more than a hundred pages. ...

Therefore I think the best way of refactoring an example would be do speak generally about the reasons why its necessary to refactor especially this code, what refactorings were applied, which effects did they have and what was the result of refactoring the code.

The problems arising from refactoring should not be spared and a very small refactoring should be shown in full length (although I fear that it will easily outsize its importance). The general restructuring that will be the result of the refactoring session will be shown in UML diagrams rather than in code, as this is more comprehensible.

But now let's go on.

3.2 Reasons For Choosing This Example

The example code is taken from a framework I developed when working for a customer who was in need of a B2B bourse for dairy production commodities (e.g. milk, cream etc.). It was designed upfront with a tree tier architecture consisting of clients connected to an application server which communicated with a database server working with the database.

As development goes, the initial prototype was taken along the development process to evolve into the final system. Quite early in development it showed up that the functionality could be easily divided into a general part representing a framework for database front-ends and a customer-specific part, that covers the requirements introduced by the customer.

The framework derived from this development is also used for another project - an ordering system for semi-manufactured products in the seals producing industry.

Like many other system, which evolved from the basic requirements made, the system was adapted to new requirements by just implementing the necessary code, without making design considerations. It did not depart from its original design, but thats the problem at all. The new requirements (e.g. using different databases, changing parts of describing metadata at runtime, implementing multiple language support etc.) were not taken as a reason to restructure the design and adapt it to the new demands, but rather added for their own sake.

Refactoring the whole framework would be a quite extensive task as there are no unit tests covering its functionality. As it is an distributed architecture, implementing them would require either a running, stable context or complete stubs to integrate the tested components (see [Nygard] for testing of distributed systems).

I chose the database server class as an example for this paper, because it was only generally designed without paying attention to delegation and different database systems. It is just a monolithic class doing to much itself, with long methods looking more like their procedural counterparts.

As this class was not written by myself, refactoring it should also improve my knowledge of this class, its functionality and intention. I don't know how this class would have looked like if I had written it myself but I have to admit that other classes of the framework are also in urgent need of refactoring.

3.3 Anticipated Problems

Creating a stub for this class which represents a database is just to much work to be sensible, therefore I use the available database for testing the class while refactoring it.

As this class has published its interface to be usable with the Java Remote Method Invocation facility, it is quite bound to that interface. But as it is only used within the framework, refactoring it is possible by modifying all references to the interface as well.

3.4 Doing the Refactoring

3.4.1 Testing

Because not a single test exists for the class, they have to be written from scratch. I incrementally write the tests as I move through the class. Each method I intend to refactor has to get its tests running first. Only after that I start refactoring the method.

Going that way, as [Fowler] recommends it as well, spares many hours of bug searching and debugging. And because with each run of the tests all hitherto refactored methods are tested as well, a bug introduced anywhere would immediately show up. Because of the distinct, small testing methods and appropriate reporting, it is very easy to spot the bugs, that could have only been introduced in the last step done.

I don't know and I really don't want to know how many hours I would have spend bug searching and debugging without the tests. Writing those tests even told me a lot about the code I was actually going to test. How it worked, which dependencies are hidden in its depths, which assertions must be kept and so on.

One problem that shows up while testing is the time it takes to connect to the remote database and to execute the statements there. A local database would have helped a lot but this is not possible. The time the tests take to run increases from 3 test in 4 seconds to 11 tests in 32 seconds. When my Internet connection is perturbed, it takes even longer (more that 200 seconds) to run the tests.

3.4.2 Log-file and Caring about Initialization

While refactoring the class I write a log file (see A.1, p.47) depicting my activities, the refactorings I apply, the tests I write and run and my thoughts about this work in progress. Together with the resources (starting file, various states in between and resulting files) supplied at [Refactoring, Hunger] the refactoring can be reconstructed. In the following I will excerpt the parts of the log which contain the most interesting information.

I start with getting accustomed with the testing framework JUnit [Gamma, JUnit] which is very easy to use. But first I have to solve two problems there. The first is the problem of killing the JVM when doing a hard `System.exit()` within the database server when encountering a error condition (e.g. no connection to the database). Doing so the testing framework is also killed because it runs in the containment of the JVM as well. So I just comment out the `System.exit(0)` code and throw Exceptions instead.

The second problem is the obstruction of the output of the testing framework by the tons of debugging output the database server produces. Therefore I have to extend the testing framework to have it writing its output to a file which can be separately but continously displayed.

After doing this I start writing the first test which deals with getting instances of the database server. As there are two methods instantiating the server either as stand-alone RMI-bound version or as an version integrated in the current JVM, I have to generalize the initialization. The testing framework can only use the integrated instance method as I don't want to blow the testing up using RMI connections.

As the testing framework is based on providing a context for each test method that is initialized just before calling the test method by `setUp()` and cleared after running the test method by `tearDown()`, I have to enable the database server to shutdown without exiting the virtual machine.

The first tests I write, test this behavior and after they run correctly I can provide the context the other test methods need without having to worry about getting and loosing instance of the database server (see A.1, p.47).

A test method for this looks like the following:

```
public void testdoBindUnbind() {
        dbserver=null;
        try{
            dbserver=new DBServerImpl();
            dbserver.unbind();

            assert("unbound ", !dbserver.isBound());
            assert("bind successful", dbserver.bindServer());
            assert("bound ", dbserver.isBound());

            try {
            Object o=Naming.lookup(dbserver.getBindName());
            assert("correctly bound (DBServer)",o instanceof DBServer);
            assert("correctly bound (Maintenance) ",o instanceof Maintenance);
            assert("ping time <100 ",
               System.currentTimeMillis()-((Maintenance)o).ping()<100);
            } catch(Exception e) {
                fail("Exception was thrown "+e);
            }
        } catch(Exception e) {
            fail("Exception was thrown "+e);
        }
        finally {
          dbserver.finalize();
          assert(" unbound ", !dbserver.isBound());
        }
}
```

Afterwards I first extract a method *ExtractMethod* for getting a Property as an `Integer` value, and then I move it *MoveMethod* where it belonges - to the class `emw.metaworks.util.RuntimeProperties`. The new call replaces all duplicate code that mimicked the same functionality. (see A.1, p.47)

3.4.3 Idea of DatabaseFormat

When I look first at the class it looks very bad - few long methods, lots of code that was commented out, lots of duplicate code and so on.

To get a better overview over the code that really mattered, I remove almost all commented out code (!409 lines) to an extra file called `DBServerImpl.java.removed`. That reduces the file size to 1000 lines (with empty lines) (see A.1, p.48)).

Then I ponder the general structure of the class and which design will be fitting to refactor into its direction:

thinking as the different databases differ in their syntax, I think about creating a `DatabaseFormat` class, which gets the responsibility of formating the raw query/update data to correct SQL sentences

thinking because the syntax order differs as well not only the symbols, the formation of a template method with appropriate calls to the DatabaseFormat is not possible

thinking the functionality of the class shall cover: SQL terminal symbol names, formation of setences of the structured query language and additional formating of String and Date objects

thinking either the complete functionality of querying the `Kriterium` or `Data` objects to form SQL sentences must be moved to the `DatabaseFormat` or its methods are only called with the extracted data

(see A.1, p.48)

With those thoughts in mind I start refactoring the basic methods of the database server, which are all about creating SQL statements of their parameters, executing them in the database and converting the results to sensible return values.

3.4.4 More Testing

For testing the methods I need a test table which can easily be created, filled, emptied and destroyed with each (test-) context update. I use the worn out table of an employee with some personal data and his manager (see A.1, p.49).

I start writing tests for creating and dropping the table, and afterwards for the creation of the necessary metadata (`DataDescriptor`), `Kriterium`, and `Data` objects, which are needed to test the methods of the database server to fill, query and delete data in the database.

After those tests, which don't directly test methods of the database server but are rather helper methods to create necessary test data, run finem, they are converted to their real purpose - being helper methods. This is accomplished by *extracting the code* that focuses on a single task (e.g. creating the table). But all the assertions which must be fulfilled when executing the functionality are kept alive in the helper methods (see A.1, p.49).

Those helper methods put the data they create in global variables and returne it , so that they can easily be used to compose real test methods.

```
public void testInsertQueryAndDelete() {
    createMetadata(); // helper
    createData(); // helper
    createKriterium(); // helper

    try {
        createTable(); // helper
        insertData(); // helper
        queryData(kriterium); // helper
        assertNotNull(query_data); // real test
        assert("1 row", query_data.length==1);
        assert("insert vs. result",insert_data[0].equalsData(query_data[0]));
    }
    finally{
        dropTable(); // helper
    }
}
```

Meanwhile I think of the following:

thinking writing tests consumes a lot of time but one learns much about the own code (in particular its shortcomings), the testing framework and in this case the database

3.4.5 Refactoring Outside The Database Server

I also refactor some other classes of the framework a little bit as it seems sensible to do so. The `Data` class gets methods to set and get attribute values not only by id but also by name, which is by now accomplished by calling a method on `DataDescriptor` (see A.1, p.49).

```
Data data=new Data("employee");
data.setAttribute(data.getDescriptor().getAttributeId("name"),"Michael");
```

```
is now done by:

data.setAttribute("name","Michael");
```

The `Kriterium` class is also equipped with a new method that allows it to set an expression in one step rather that in two.

```
Kriterium krit=new Kriterium("employee");
// which is possible due to the previous step
krit.setAttribute("name","Michael");
krit.setOperatorId("name",Kriterium.LIKE);

is now done by:

krit.setComparision("name",Kriterium.LIKE,"Michael");
```

This not only increases the readability of the code but also reduces the amount of not duplicate but similar code.

3.4.6 First Methods Extracted To DatabaseFormat

The tests soon point me to an bug that has been hiding in the database server for a long time. It does not correctly format boolean values for interpretation by the progress database. The bug shows up when I run the test method for inserting a value into the employee table. Removing it would have been a matter of seconds if I only have the documentation for progress at hand. So it takes me some minutes of guessing and trying.

As one colleague of mine has already started (but unfortunately only started) to refactor the database server class, the `query()` method is accompanied by the extracted methods`createFrom()`, `createJoin()`, `createWhere()` and `createQuery()`. Those methods provide a good starting point for refactoring. Like every time I start writing and debugging the test code and afterwards I look at the methods for possible refactorings.

I already thought about creating a distinct class *Extract Class* for doing all database related formating. This class is created now as `emw.metaworks.DatabaseFormat`. At first all calls to this class are done with static class methods, as this spares me pondering instantiation issues. (see A.1, p.50)

One thing I do everywhere is replacing `String` concatenations with `StringBuffer` operations as this spares lots of `String` creations. One other thing is *extracting methods* for concatenating strings to `StringBuffers` with correct processing of `null` values and necessary delimeters. Those methods - `andExpressions()` and `concatenateExpressions()` - are *moved* to `DatabaseFormat`.

A basic functionality that does belong to `DatabaseFormat` is formatting objects for use with the database. The functionality fulfilled by the `ObjectToOracle` class and some hacks are transformed to a `formatObject()` method now belonging to DatabaseFormat (*Extract Method*, *Move Method*). After all of the former code is transformed to use the new method, the resulting methods already look much cleaner to me (see A.1, p.50) because this eliminates a lot of duplicate code.

```
Object attributValue = data.getAttribute(j);
String valuesString=ObjectToOracle.getOra7String(attributValue);
if (valuesString.startsWith("to_date"))
      valuesString="date('"+df2.format((Date)attributValue)+"')";
if (attributValue!=null && dd.getConstraint(j).getType()==Types.BIT)
      valuesString=(((Boolean)attributValue).booleanValue())?"yes":"no";

is now written as:

valuesString=DatabaseFormat.formatObject(data.getAttribute(j));
```

```
 with:
class DatabaseFormat {
    public static String formatObject(Object obj) {
        if (obj == null) return "NULL";

        if (obj instanceof Boolean)
            return (((Boolean)obj).booleanValue())?"yes":"no";
        if (obj instanceof java.util.Date || obj instanceof java.sql.Date ) {
            return "date('"+dateFormat.format(obj)+"')";
        }

        if (obj instanceof java.lang.String) {
            return "'"+ obj + "'";
        }

        return obj.toString();
    }
}
```

3.4.7 Switching off a Switch (Statement)

Another very heavy piece of code is a `switch` statement in `createWhere()` that is 60 lines long and duplicates code in every `case` statement. This one is *extracted* (see A.1, p.50) into an own method named `getExpression()` which in turn *delegates* its work to `formatObject()` and some other helper methods. An excerpt of the relating code is appended (see A.1, p.56).

After refactoring those methods, they are *moved* to `DatabaseFormat` as they only represent formatting issues and nothing related to the interaction with the database.

The `query()` method then uses all resulting strings of its extracted methods to create the complete query and execute it on the database (see A.1, p.51).

One heavy benefit that has already gained me a revenue is eliminating duplicate code. When I find errors that reside in already refactored code, I only have to correct them in one place and not in eight or ten as before.

3.4.8 A Bright Moment

At this point I am doing testing and refactoring and have quite a lot of good results and fun as well (If everything runs OK and things get better quite easily it should be fun to continue). But now I come across a problem that should have required more thinking if done in a conventional way. Fortunately I don't do it the conventional way. To catch the thoughts that possess my mind at that moment I think its best to quote from my own log (see A.1, p.52).

refactoring I am even more surprised, when pondering about problems arising from the temporary local variable (boolean array) `isPartOfQuery` for the elements of the `Kriterium` object, I realized that applying the *Replace Temp with Query* refactoring could not only be helpful but also allows to restructure the `createSelect()` method as this is the method producing the boolean array.

refactoring So I extract the code needed for `isPartOfQuery()` from `createSelect()`. Then I have a very strange moment when things begin to fall in place and the code really communicates to me that this is the right thing to do. Suddenly I am able to cut the `createSelect()` method to its basic task, creating a part of a SQL select expression for the fields of the `Kriterium` object that have to be retrieved from the database. The work of determining which are those fields, doesn't have to do anything with this task but is mingled before in `createSelect()`.

refactoring In the end I have two methods which are both well structured, small and which concentrate on their basic tasks.

thinking I think that is the real gain from refactoring. On the way to restructure your code you apply a refactoring and this starts a chain reaction which results in a - not previously anticipated but convincingly superb - code that does still all the things it should but is much more focused and clear. Not mentioning the removal of lots of duplicate code by working through it. I don't think one would come to this conclusion that easily when trying to imagine it with an upfront design.

testing and all test still run fine ;)

thinking I must say when experiencing this moment of clarity, I have to agree with MARTIN FOWLER who states that refactoring is a lot of fun. In my humble opinion refactoring has even more gains than programming from scratch . You not only take a bad thing and turn it into a high quality piece of code without much thinking but you can also learn much more about the system in general and in detail, about programming style and design and about the inhoerent structure which is needed to solve a problem, but which is usually disguised by many lines of bad code.

The code mentioned above can also be found in the appendix (see A.2, p.57).

3.4.9 Reaching The Limits

Unfortunately I have not enough time (refactoring this class took me about 25 hours) to refactor everything I want. When working on the `query()` method, I realize that a change in the metadata structure of the framework, would have allowed me to split the remaining `query()` method into even smaller pieces. As I later determine this change would have had a impact on the updating method as well. But because this refactoring will have quite dramatic consequences for the whole framework I only note the fact and continue. I think refactoring is also about when to stop doing it (see A.1, p.52).

The methods that follow are quite easily to refactor as I have already laid the foundation for it. I *extract* the code that creates the SQL statements used to interact with the database to own methods which are *moved* to `DatabaseFormat` after clearing them up. Of course I continue writing tests and running them until they succeeded.

When I ponder about refactoring the `update()` method I find a failure in the design of the approach of distinguishing between update and insert operations on the database. By now only data that has been read from the database is used to update, newly created data is just inserted. But rethinking the approach does not show up an easy solution for this problem. Either newly created data can be used to update existing data or read data could have been removed in the meantime requiring an insert operation to recreate it. Therefore the most convenient method to detect the correct mode of operations would be to check the database before deciding on the operation. But this will require far to much time and effort to be useful. ...

After finishing the `update()` method and doing some clean-up at the end of the class I am happy to see that the refactored class looks much better than its predecessor.

3.4.10 Final Moves

The final refactorings I make, are to change `DatabaseFormat` methods from static ones to normal instance methods. By adding a static getInstance() that is able to return single instances of different subclasses of DatabaseFormat I create an access point to the class. Two of those subclasses are created. One for the `Progress DBMS` the other for `Oracle` that provide the different syntaxes needed for producing correct SQL statements for those databases. They just overwrite those methods of `DatabaseFormat` that have to produce different results (see A.3, p.59).

The database server now uses a parameter supplied in the configuration file to determine which database it is connected to and gets the according instance of `DatabaseFormat` by using the `getInstance()` method.

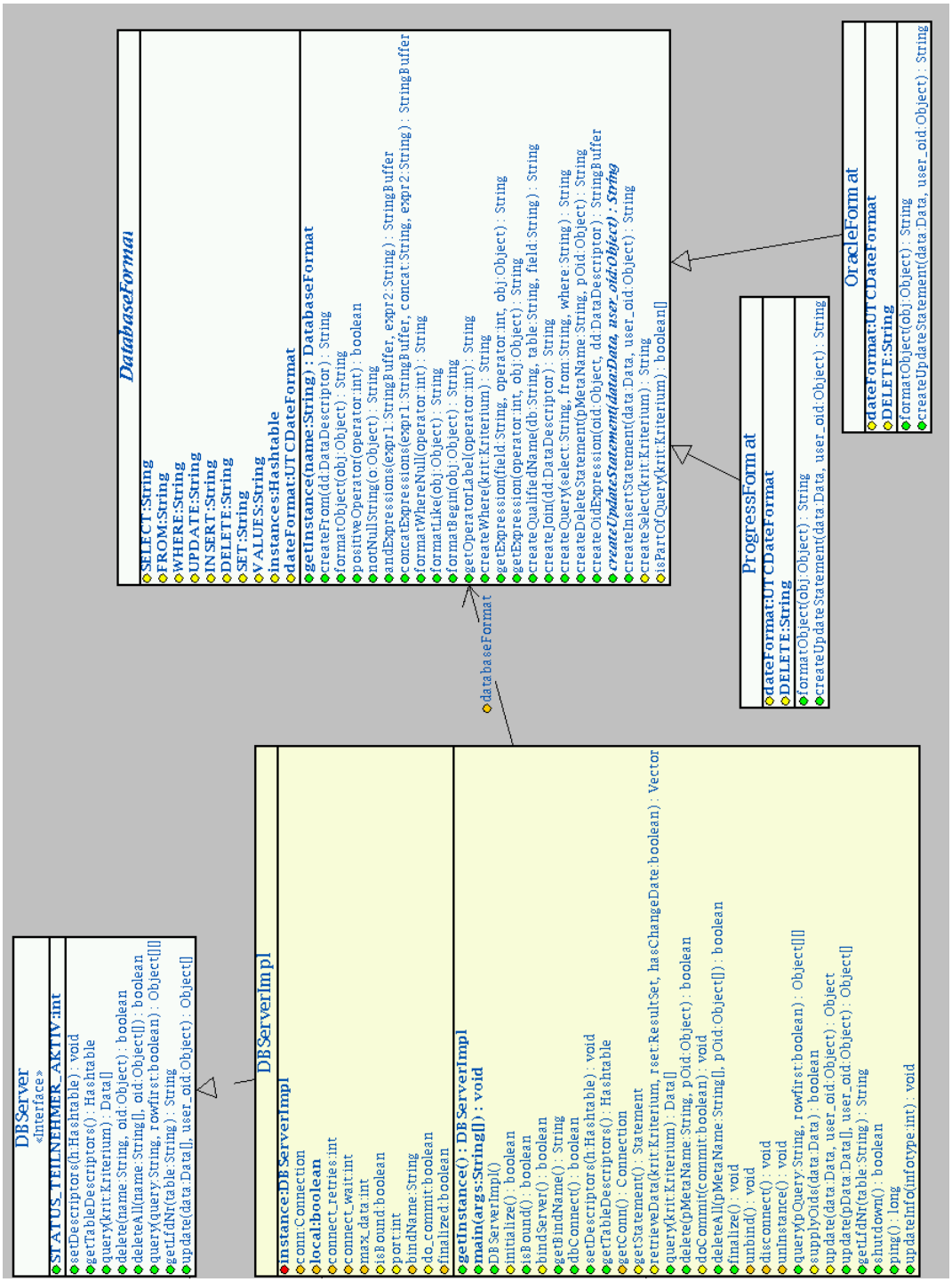

Figure 3.1: UML-Diagram of Resulting Design

The resulting structure is shown at the UML diagram of the concerned classes (see 3.1, p.42), which is created using JRefactory (see 2.3.2, p.28) on the package and by hiding the unwanted classes under the margin.

3.5 What were the Benefits and Disadvantages of Refactoring the Class

3.5.1 Benefits

I must admit that refactoring the database server class highly increased the quality of the code. It not only makes the code more readable but also allowes it to communicate its purpose and intention without the use of commenting. That was achieved by choosing explaining names for methods and variables and by restructuring the class from containing large methods which did everything to small specialized methods which call functionality they don't provide themselves. This can be seen at the following example which presents the final version of the `update()` method. (I don't dare to show the first version I only want to mention that it covered 230 lines.)

```
protected Object update(Data data,Object user_oid) throws Exception {
    if (data==null) return null;
    Statement updateStmt=null;
    ResultSet updateRset = null;
    try {
      updateStmt = getStatement();
      String query = null;
      Object oid = data.getOids();

    if (data.wasRead()) {
      query=databaseFormat.createUpdateStatement(data,user_oid);
      updateStmt.executeUpdate(query);
    }else {

        if (!supplyOids(data)) return null;
            oid=data.getOids();
            query=databaseFormat.createInsertStatement(data, user_oid);
            int row = updateStmt.executeUpdate(query);
            if (row == 0) return null;
    }
    return oid;
    }
  finally {
    updateStmt.close();
  }
}
```

Other benefits I experience when refactoring the class are a better understanding of its intention and functionality and of the database behind it. I learn a great deal about unit testing. Writing tests can save very much time when bugs are introduced during the coding. I also experience the feeling how reassuring it is when running the tests again and again without having them reporting errors.

Another very time saving result of refactoring the mess that the class presented to me at the first time, is the ability to correct errors - when found - in seconds rather than in minutes or hours. The elimination of the duplicate code that was heavily present in the class before limits the necessary changes to a single place.

As I now know and have understood the refactorings listed in the book of MARTIN FOWLER [Fowler], it is never complicated to find a refactoring that allows me to restructure the present code.

When thinking about the refactorings and using them, they just became tools as other techniques of software engineering like design patterns. Therefore I think although the reading of the catalog of refactorings in the book was not always thrilling it sharpens one's senses for when to apply which refactoring.

The main result of refactoring the example apart from my personal experiences and the time issues is the convincing design that appears incrementally while working through the class. Instead of having one bulky class that grabs all activity for itself, it now delegates all formatting issues to another class and some other functionality to the classes that are responsible. The database server class now only deals with calling the formatting methods, doing the database queries and extracting and returning the results. This design is very robust but also open to changes as the functionality is separated in small pieces that interact with each other.

3.5.2 Disadvantages

One burden that refactoring always bears is the testing effort that is the base for doing a refactoring. When code has to be refactored for which no tests exist, it take a lot of time to create all tests that are necessary because a running test is the prerequisite for refactoring a method. If the tests for all methods of a class to be refactored are written at once, the expenditure will be to big. Therefore I take the approach of only writing tests for the method I am going to refactor and the methods it calls (if necessary).

So I get an iterative cycle of writing tests and making them run and only after that I start refactoring. This also provides me with more satisfaction as changing the focus of my work helps keeping an open eye for potential failures.

Another disadvantage of refactoring is that the whole process is quite alluring. You have to force yourself to stop refactoring as it's really fun and you see your code getting grown up. But I think most projects limitations naturally put an end on your refactoring activities.

Unfortunately I was not able to do refactoring with a peer. I can imagine that it helps and hinders as well. It is absolutely helpful if you are stuck in a problem or if you have lost the right direction. And as I experienced sometimes, refactoring over some hours is also exhausting. A peer can take the keyboard away from you and continue refactoring while you are able to let your thoughts fly around a bit.

As I refactored a class that was implemented in the Java language, I only have 3 tools available to support me with refactoring. Those tools support only a very limited number of refactorings in different qualities. The different tools don't integrate that easily as well. Xref for example does support my own editor named emacs and allows me to rename, extract methods and modify parameter lists. JRefactory only supports its own interface which is not that easy to handle but allows the creation and removal of classes, moving classes and methods (at least theoretical) even in the inheritance hierarchy, and renaming as well. The IntellJ Renamer on the other hand has its own interface but supports only moving classes and packages and renaming all symbols. Therefore using all these refactorings would have required me to run the tools (who have different but long startup times), according to the refactoring I want to apply.

Fortunately I only have to deal with one class without tampering its public interface. Therefore all changes affected only that single class. So I was faster using the capabilities of my editor to implement the refactorings. I had a problem running the Xref-Speller that was my own fault, otherwise I would have used it because it easily integrates in my development environment.

Chapter 4

Summary and a Look Ahead

4.1 Summary

After dealing with refactoring for the time of writing this paper, I was convinced and convinced myself that refactoring shall be seen as a technique which every programmer can and shall count on. Not knowing of refactoring and not applying it to the code one has to work with, leads to a considerable loss of quality.

That the design of software decays over time is a sad but fixed fact that can't be easily ignored. Ultimately the one who must maintain or extend the software system has to deal with the bad design that evolved over the time. He will have problems understanding the code, extending or debugging it when he is not supported by the code itself i.e. his antecedent programming colleague.

Because doing refactoring is not that hard, it should be done every time it is necessary. Although the considerations about the time lost when refactoring may be pressing, the benefits gained from code and design of higher quality more than compensate for the effort.

Fortunately refactoring doesn't depend on a certain development process. It can be used constantly during coding (or named otherwise in the other phases, e.g. redesign) without interrupting the process. If it is steadily used, it even doesn't add much effort to the coding because the tests required already exist. Moving some code around is not a very hard work at all. Only documenting the changes done may require more effort.

Although some development processes provide a better environment for refactoring as they are also highly concentrated on developing high quality software in small steps ([UML+Patterns, Larman], [Beck,Xp]), refactoring shouldn't be ruled out in other processes.

As I already stated, refactoring should be an often used tool in the toolbox of every programmer much like design- or coding patterns. But not only the one who applies refactoring to the code should know of the benefits it has. Every other person involved in the project (e.g. manager, customer, consultant) should be aware that refactoring is an easy way to provide code of high quality and has to be done without questioning.

In general it can be said that the positive effects of refactoring heavily outweighs the disadvantages which are reduced when refactoring is steadily used and the base for implementing refactorings already exists. When the notion of saving time by refactoring first and extending the functionality afterwards is incorporated in the thinking of the people the benefits will be always notable.

4.2 Future Development

The future development should be directed to the reduction of the effort for implementing refactorings further. This can be done by tools which have the Smalltalk Refactory Browser as an example. The half-automatic application of refactorings based on the parse tree of the code to be refactored with restrictions and semantic checking rules is the step that has to be taken for other programming languages than smalltalk as well. I think for everyone who does not extensively busy himself with refactoring, the exhausting task of applying the individual refactorings can be a reason

not to do it. That reason will only fade away, when tools exist that make life easier by automating much of the boring and error prone handwork.

Unfortunately by now none of the tools examined in this paper can catch up to the example given by the Smalltalk Refactory Browser.

The next step of development is seen with cautious eye by many developers. Although it seems possible to apply refactorings automatically if the rules and restrictions when and how to implement the refactorings are specified carefully, the result of such work may be unsatisfying. The decision when and where to refactor often does not only depend on the structure of the existing code but has also to do with the limitations and the demands of the project and the intentions of the programmer who is able to refactor in the direction of a certain goal. No tool can (at least in the next years) replace the creative thinking and experience of a programmer who has worked with good and bad code. He knows that an investment into keeping the quality of the produced software high does not only satisfy the customer but supports everyone who has to deal with this software in the future. Perhaps this will be even himself. ...

Appendix A

Refactoring Example

A.1 Refactoring Log

Legend:

testing Unit Test

coding Source Changes to make the tests run, remove bugs etc.

refactoring Refactorings

thinking saved states and thoughts

The log file written wile refactoring emw.metaworks.DBServerImpl:

testing Unit Test testSetup for instantiating DBServerImpl

coding using finalize instead of shutdown as this does not exit the JVM via System.exit(0)

testing testGetInstance for getting instance of DBServerImpl

coding finalize also has to unbind server from RMI-Registry, as multiple instances without quitting the virtual machine, occupying the same socket port

refactoring renamed bindString to bindName and making it a global variable

refactoring made port a global variable

coding bindString and port are used in finalize() to unbind the server

refactoring splitting method finalize() into two methods - disconnect() and unbind()

coding as DBServerImpl is designed for singleton use, a method to unInstance it is added to the finalize method

coding for preventing finalize() to be called twice (a second time by the garbage collector) a finalized state variable was introduced,

coding unbinding only if not called from a local VM, i.e. if it was not bound to a separate address

refactoring accessor for bindName was added

coding finalize() is extended to act differently on RMI-bound and local servers

coding due to the impossibility of removing an installed registry, the exception the createRegistry() call creates is silently ignored

coding commenting out System.exit(0) as this causes the test environment to exit as well

coding new refactoring is to extract the methods needed for setting up the rmi binding from startServer() it is called bindServer()

refactoring as:

```
try {
  port=new Integer(RuntimeProperties.u().getProperty("DB_PORT")).intValue();

} catch(Exception e) {}
```

is duplicated very often throughout the class, it is extracted to an own method, getPropertyInt

refactoring to catch the exception an default value must be supplied

testing Unit Test testGetPropertyInt is written and runs from the first run

coding bindServer() returns now a boolean value instead of using System.exit(0) when failures occur, this boolean value is stored in the variable isBound

thinking state 2 saved

coding remove a lot of commented out and testXX() code (409 lines, of 1400) to a separate file DBServerImpl.java.removed, to make the class more comprehensible

testing rerun tests (problems with reloading actual class files in the Swing Version, switching to text version, test results should also be written to a file to not interfere with outputs from tested class)

thinking state 3 saved

refactoring replace other uses for getPropertyInt in constructor, moved try clause to not longer watch these lines

refactoring added failure warning for getPropertyInt

testing tests run, the testSetup and testGetInstance are commented out because they consume to much time (connecting to db)

refactoring moved all initialization (e.g. reading properties file and setting variables according to properties) to method initialize(), to collect these things at one place

testing test getInstance broke, as bindName is now created regardless of, bound or not bound, changing test to check to isBound flag

testing test commented in again ;)

refactoring as start Server does only contains delegation it is inlined into the constructor

refactoring looking at dbConnect reveals that the JDBC driver is looked up each time a connect try is done, move it out of the loop

thinking considering dbConnect to return a Connection object instead of boolean result, but delayed that decision

refactoring a test is added to check RMI stuff, binding and unbinding the server, checking if the correct object was bound and performing a method call on it

refactoring moved getPropertyInt directly to RuntimeProperities

testing tests run correctly after adaption to moved reference

refactoring setDescriptors() and getDescriptors don't require testing methods as they only delegate their work

thinking as the different databases differ in their syntax, I think about creating a DatabaseFormat class, which gets the responsibility of formating the raw query/update data to correct SQL sentences

thinking because the syntax order differs as well not only the symbols, the formation of a template method with appropriate calls to the DatabaseFormat is not possible

thinking the class should cover, SQL terminal symbol names, formation of sentences of the structured query language and additional formating of String and Date objects

thinking either the complete functionality of querying the Kriterium or Data objects to form SQL sentences is moved to the DatabaseFormat or its Methods are only called with the extracted data

testing test for DBServerImpl.createFrom is created

testing an test table within the database is needed, the following format is used

```
Table employee:

id number(10) // primary key
name varchar(20)
birthday date
salary number(7,2)
age number(3)
skilled boolean
manager number(10) // foreign key to employee
```

testing getConn() is implemented for getting a valid connection to the database used by the TestCase; has to be removed later

testing a testCreateTable is created to create and drop a table

thinking writing tests consumes a lot of time but one learns much about the own code (in particular its shortcomings), the testing framework and in this case the database

testing testCreateTable succeeds

testing a test for creating a DataDescriptor for the testTable

testing a test for creating a Data object

coding added convenient methods for setting and querying data attributes with names rather than only with id's

testing test for creating a Kriterium object

coding added a method for setting the operator and the comparison value in Kriterium together

coding added methods for setting and getting operators by name (Kriterium) and for modification check also by name (Data)

testing test runs

refactoring using ExtractMethod on the testCreateTable method to split the creation and removal of the table into two separate entities

testing a test for inserting, querying and remove a row into the table is created

testing as creating a connection to the database within setUp and closing it within tearDown is too costly (regarding the time consumed), a method getStatement is added that provides a valid statement for the database

coding found an error in DBServerImpl regarding the storing of boolean values in the database, corrected, tests run

testing creating a test queryData();

thinking inserting and deleting don't require the database name, otherwise querying does, this produced an error when running the test

coding added database name e-db to createDescriptor

testing all tests are now composed of small creation methods

thinking there are problems creating a database-qualified table within the restrictions of the database, but a non qualified table produces errors when querying

coding correction of the error when querying

coding as assertEquals does not cover the comparison of two Data objects, a method named equals-Data is added to the Data class

thinking creating a Data with java.lang.Float is converted to java.lang.Double when stored in the Database even with type float

thinking type FLOAT in Progress is mapped to Double in java

thinking a NULL value written in the database is returned as an 0 value when using a integer, this problem seems quite serious as all values seem to be converted to basic values of the Object Types,

coding problem solved using ResultSet.wasNull() in the DBServerImpl query() method to test for null values after retrieving the objects

thinking perhaps this must be moved to the DatabaseFormat class

testing wrote test method for createFrom()

testing test failed, because a space was missing, corrected the expectation an rerun successfully

testing added second metadata (foreign key reference employee.manager -¿ employee.id) for more testing of createFrom

testing after some error searching the metadata creation and tests run

refactoring removed some duplicate Code from createFrom, and replaced String with StringBuffer

refactoring an emw.metaworks.server.DatabaseFormat class is created

refactoring createFrom is moved there

testing testCreateFrom is adapted to the new source of the method, testing is successful

testing testCreateWhere missed its first run by some spaces and missing parentheses

testing added second Kriterium to be tested with createWhere

testing found bug representing boolean values in the database query, but as this one is deeply buried in the code of the database servers createWhere method, it seems as if refactoring has to go first

refactoring as DBServerImpl relies on ObjectToOracle for formatting Object values, the functionality that resides there has to be moved to the DatabaseFormat class, the references in the DB-ServerImpl has to be modified as well

testing test still run after moving ObjectToOracle.getOra7String() to DatabaseFormat;

coding modified the moved method to correctly respond to boolean values

refactoring replaced now incorrect name getOra7String() to formatObject()

refactoring I would like to overload formatObject for formating the different types but the method is called only with Object objects and not with type specific object

coding by the way removed a hack associated with wrong formatting of date objects regarding to progress database, as well as the hard wired formatting of boolean values

testing test still run fine

refactoring extracted huge switch statement into new method getExpression()

refactoring extracted methods for formating NULL values, like and begin expressions to DatabaseFormat

coding introduced method getOperatorLabel in DatabaseFormat

coding removed 43 lines of code from the switch statement and replaced it with just one line of code:

```
default:
   return DatabaseFormat.getOperatorLabel(operator) +
          DatabaseFormat.formatObject(obj);
```

testing after clearing some spaces and cases (upcase in this case) the tests run fine

thinking saved state 5 of DBServerImpl.java

thinking next method to be tampered with is DBServerImpl.createJoin()

testing testCreated for createJoin

thinking as the test encounters problems with not set databases (e.g. null.employee.manager = null.manager.id) i write a method encapsulating the creation of such structures which produces correct results

coding first some cosmetics, ordering code and using Stringbuffer instead of String

testing the tests showed up an error I produced when I introduced StringBuffer which did not show up at testCreateJoin() but rather at testInsertQueryAndDelete()

testing tests run fine now

refactoring as createJoin has useless parameters they should be removed createJoin(int attributeCount, DataDescriptor dd, Vector fromTableVector) to createJoin(DataDescriptor dd)

testing after adapting to the new parameter list the tests run correctly

thinking the next method to be looked at createQuery is small enough and well structured, it doesn't have to be refactored, but rather moved to DatabaseFormat

refactoring as the query() method that uses all the previously refactored methods is next, I think it is a good time to move the appropriate methods to DatabaseFormat (createWhere(), getExpression(), createQualifiedField(), createJoin(), createQuery()

testing after moving the methods and adapting the references all test still run fine and even faster as no longer an instance of DBServerImpl is required to perform the tests which does automatically connect to the database

thinking DBServerImpl shrank by another 100 lines of code

refactoring the next method to be refactored is DBServerImpl.query() which is a quite heavy one and which yearns for refactoring

refactoring extracted all code regarding the creation of the select statement from the query() method into the createSelect() method

thinking saved state 6

refactoring refactored the createSelect() method

testing when refactoring I introduced the error of using the String "table" instead of the variable table, the test failed, and because the method was already refactored I only had to change the code in one place

testing added some more test data and found another bug regarding the not equals operator in Progress, corrected it as it only appears once now in DatabaseFormat.getOperatorLabel, now all tests run again

testing added more test data for querying referenced tables by using the "Manager" metadata

refactoring move the test of an available database statement to an extracted method named getStatement()

thinking the notion of the external definition if a table has a change date field, should be internalized into the appropriate metadata structure, but not now :)

refactoring as the code structure (object!=null) ? object.toString():""; appears much to often, it is replaced with calls to the method notNullString(object)

refactoring it is moved together with andExpressions() to DatabaseFormat as it is a kind of formatting and its more often used there, both methods are introduced at all necessary places

testing tests are still ok (I'm astonished ;)

refactoring I am even more surprised, when pondering about problems arising from the temporary local variable (boolean array) isPartOfQuery for the elements of the Kriterium object, I realized that applying the *Replace Temp with Query (120)* refactoring could not only be helpful but also allowed it to restructure the createSelect method as it is the method producing the boolean array.

refactoring So I extracted the code needed for isPartOfQuery() from createSelect() !! code and had a very strange moment when things began to fall in place and the code really communicated me that this was the right thing to do. Suddenly I was able to cut the createSelect() method to its basic task, creating a select part for the fields of the Kriterium object that have to be retrieved from the database. The work of determining which are those fields, doesn't have to do anything with this task but was mingled before in createSelect().

refactoring In the end I had two methods which were well structured, small and which concentrated on their basic task.

thinking I think that is the real gain from refactoring. On the way to restructure your code you apply a refactoring and that starts a chain reaction which results in a not previously anticipated but convincingly superb code, that does still all the things it should but is much more focused and clear. Not mentioning the removal of tons of duplicate code by working through it. I don't think one would come to this conclusion that easily when trying to imagine it as an upfront design.

testing and all test still run fine ;)

thinking I must say when experiencing this moment of clarity, I have to agree with MARTIN FOWLER who states that refactoring is a lot of fun. In my humble opinion refactoring has even more gains that programming from scratch, because not only you take a bad thing and turn it into a high quality piece of code without much thinking about but also you can learn much more about the system in general and in detail, about programming style and design and about the inhoerent structure which is needed to solve a problem, but which is ususally disguised by many lines of bad code.

thinking saved state 7

refactoring extracted the retrieval of the Data objects to an own method retrieveData()

testing tests run without problems

refactoring although I'd like to refactor the query() method further, I stop here because further refactoring would have to introduce change date aware metadate and to deal with multiple return types for each part of the select statement which is created separately. As I can't predict if those parts are concatenated everytime in the same way without changes, I also don't add a method for creating the whole select statement string at once by calling the creation methods itself.

thinking now I'm done with the first half of the DBServerImpl class which lost almost one third of its code.

coding changed createQualifiedField to createQualifiedName covering now all possible combination of valid parameters to create a qualified reference symbol.

testing test confirmed

refactoring refactored delete by using Extract Method to move a part of the code into createDeleteStatement wich can be transferred to DatabaseFormat, and some of it into doCommit() which deals with committing the changes made be delete() and deleteAll()

testing I'd like to add a test for tables with multiple oid's, but this would require to get the Data class to make the oid values contained available using one method

coding changed the Data class

testing modified the tests to work with two oids and it worked

refactoring used MoveMethod to move createDeleteStatement to DatabaseFormat

testing tests failed due to a wrong String constant in DatabaseFormat

testing tests run

refactoring deleteAll skipped as it didn't require refactoring

thinking fast forward to line 597 of 975, as the methods betwenn were already refactored, saved state 8

thinking next method to be refactored is a query method that executes a SQL string directly and returns the results as Object[][]

testing first writing test

testing the old problem of java.util.Date stored and java.sql.Date retrieved showed up again, now I set the Date values as instances of java.sql.Date but this problem is in need of further evalutation

refactoring refactoring query() by introducing getStatement()

refactoring replaced variable col_count by column_count

refactoring changed Vector of Vector to Vector of Object[] as this lowers the conversion efforts

testing tests run fine

thinking thought about using ExtractMethod to extract the functionality of converting a Vector of Arrays to an own method but as this is not that much code and as it is not duplicated by now I leave it in place

refactoring removed empty method initTableDescriptor() as it is used nowhere

thinking next method is updateData which is also quite large and the last method of such importance and size, its now approximately 210 line long :(

testing a test for the inserting functionality of update() ran all the time by now and it is expanded now to run a second time to update the values

thinking the date object seems to have a failure in design, updating data values occurs only on Data values that were read from the database, not for newly created ones

thinking it would be better to distinguish the methods of updating and inserting or doing a previous test if the row of data already exists within the database (which costs a lot of time btw)

testing actually I'm marking the data values to be updated (instead of being inserted as coming from the database), and I have to supply a nonexisting (i.e. null) Date value to its constructor, meaning there is no last_change field in the database

testing modified in that way that the test does first insertion of the data values and second updating with changed values, which are subsequently read for verification from the database

testing test runs now

refactoring added method getExpression(field,operator,object) to DatabaseFormat as this is used everywhere instead of getExpression(operator, object)

refactoring introduced getStatement() into update()

refactoring moved the code within the outer for loop of update into an extracted method of update(Data,user_oid)

refactoring extracted a method createUpdateStatement() from the single update() method

testing tests run

refactoring replaced weird String adding constructions with calls to concatExpressions(getExpression())

testing tests threw an exception as I missed the SET statement within the query, corrected

testing tests run

refactoring extracted method createOidExpression() from createUpdateStatement() which resembles the code used in delete()

testing tests run

thinking btw. multiple oids work fine now as well

refactoring MoveMethod for createUpdateStatement and createOidExpression to DatabaseFormat

testing tests run fine ;)

refactoring replaced duplicate code in DatabaseFormat.delete() with createOidExpression()

testing tests ok

refactoring Extracted Method createInsertStatement() from the second part of update()

testing tests ok

refactoring Extracted Method supplyOid() for inserting a new, valid oid if it is not exitent. As this method, i.e. the way new oids are created, is project specific, it should be overwritten as needed.

testing tests run

coding introduced the concatExpressions() instead of String concatenation in createInsertStatement()

coding added method getOidArray to the class Data as this covers most of the ugly workarounds which were introduced by Data.getOids()

refactoring Moved Method createInsertStatement() to DatabaseFormat to sit around with its siblings

testing tests still run

refactoring commented out method DBServerImpl.getLfdNr() as this is specific to a project (much like supplyOids), which should be overwritten by an subclass of DBServerImpl. Therefore it should be marked abstract.

thinking *I am done with the DBServerImpl class !!*

thinking the class shrank from 1235 loc (DBServerImpl.java.0) to 648 loc, with 278 loc moved to the class DatabaseFormat, and produced 220 lines of this report ;)

thinking ideas for refactoring report, unfortunately they appeared to late to me to be used in a sensible way: timestamps, perhaps durations of the steps, number and duration of the tests (I must acknowledge, that I've written not enough tests but rather modified the testfile sometimes for another type of test (e.g. 2 oid's instead of one). My 10 tests run now in less thant 36 seconds which is quite long :(but due to the database connection inevitable.

refactoring TODO: as the extracted methods evolved during the refactoring and were not available from the beginning the class has to be reexamined which pieces of code would benefit from the application of the extracted methods (e.g. to remove code duplication)

coding added DatabaseFormat.createQualifiedName() and DatabaseFormat.concatExpressions() to DBServerImpl.createSelect()

refactoring change static calls in DatabaseFormat into public method class to a singleton pattern instance of this class, which should be overwritten for the database specific formatting (e.g. ProgressFormat and OracleFormat)

coding added getInstance() to DatabaseFormat which returns a single instance

coding enhanced getInstance to work with class names to return also those classes which are descendants of DatabaseFormat, internally the successfully created instances are stored within a hashtable

testing added tests for getInstance() which run fine

refactoring DBServerImpl should have an config file parameter that denotes the database, an could be used to get the appropriate instance (much like getDriver())

coding added config file parameter named DBMS_NAME

coding added variable databaseFormat which substitutes static call to DatabaseFormat, and which is set by using DatabaseFormat.getInstance(DBMS_NAME)

testing tests still run perfectly

thinking as the current DatabaseFormat covers the syntax and specialities of the Progress DBMS, its functionality should be partially moved to a subclass named ProgressFormat

refactoring Using the Extract Subclass refactoring to create the appropriate subclass moving the necessary parts of the methods to the subclass

refactoring started with formatObject() which does special formating for the Boolean and Date values in ProgressFormat

testing tests run

refactoring next in row is createUpdateStatement() whose results' syntax differs to much in Progress and Oracle

refactoring the constant DELETE is also moved to ProgressFormat, because it covers a different syntax as well

testing all tests stills run

thinking I think all of the methods moved to ProgressFormat should cover the differences to DatabaseFormat

thinking now another subclass named OracleFormat can easily be created, (it is instantiated automatically by calling DatabaseFormat.getInstance("Oracle")), the same methods have to be moved there but containing other code

testing the OracleFormat should be tested as well but unfortunately I don't have access to an Oracle System right now, a new test class must be created as well, as the assertions in the current test class clash with the syntax produced by OracleFormat, ideally this would be accomplished by subclassing DBServerImplTest to DBServerImplTestOracle

thinking DBServerImpl should be overwritten on a per project basis, it has to be reexamined which parts are still project and (hopefully none) database dependent

A.2 Refactoring Switch Statement

```
createWhere() {
...
// within an iteration over all attributes of the Kriterium object
  switch(krit.getOperatorId(i)){
  case Kriterium.NONE:
    break;
  case Kriterium.LIKE:
    where =  (where.equals("")?(" (" + spalteName):(where + " and (" + spalteName));
    if (obj==null)
    where = where + " IS " + objOra7String+ ") ";
    else
    where = where + " like '\%" +
        objOra7String.substring(1, objOra7String.length()-1) + "\%')";
    break;

// this is repeated 8 times to form the complete switch statement
...
now the code looks like:

createWhere() {
...
// within an iteration over all attributes of the Kriterium object

        operator=krit.getOperatorId(i);
        if (operator == Kriterium.NONE) continue;

        spalteName = createQualifiedName(dd.getDB(i,false),
                     dd.getDBTable(i, false), dd.getAttributeName(i));

        obj = krit.getAttribute(i);
        where=andExpressions(where,
                             getExpression(spalteName,operator, obj));
...
with located in DatabaseFormat:

    public String getExpression(String field, int operator, Object obj) {
        return field + " " + getExpression(operator,obj);
    }
    public String getExpression(int operator, Object obj) {
        if (obj==null)
            return formatWhereNull(operator);

        switch(operator){
        case Kriterium.LIKE:
          return formatLike(obj);
        case Kriterium.BEGIN:
          return formatBegin(obj);
        default:
          return getOperatorLabel(operator) + " "+
                 formatObject(obj);
        }
    }
...
```

A.3 Replace Parameter with Query

```
  public synchronized Data[] query(Kriterium krit) throws RemoteException{
    boolean[] isPartOfQuery=new boolean[krit.getAttributesCount()];
...
      String select=createSelect(krit, isPartOfQuery);
...
}
// the following method was extracted from query()
    protected String createSelect(Kriterium krit, boolean[] isPartOfQuery) {
      String select = "";
      Debug.dbg("DBServerImpl.query("+krit.getName()+")");
      DataDescriptor dd=krit.getDescriptor();
      Vector selectVector=new Vector();
      Vector fromTableVector=new Vector();
      Constraint c=null;

      for(int i=0;i<dd.getTables().length;i++)
        fromTableVector.addElement(dd.getTables()[i]);

      for (int i= 0; i < krit.getAttributesCount(); i++) {
        isPartOfQuery[i]=false;
        c=dd.getConstraint(i);
        if  (c.getTable() == null) continue;

        if (c.getType() == DataDescriptor.REF || c.getType() == DataDescriptor.BACKREF) {
          if(dd.getDBTable(i, true) !=null && dd.getDBTable(i, false) !=null ){
            // nur Tabellen nehmen, deren Spalten selektiert werden
            if(fromTableVector.indexOf(dd.getDBTable(i, true)) != -1 &&
               fromTableVector.indexOf(dd.getDBTable(i, false)) != -1){

              selectVector.addElement(dd.getDBTable(i, false) + "." +
                                      dd.getAttributeName(i));
              select+=((select.length()>0)?", ":" ")+dd.getDBTable(i, false) +
                      "." + dd.getAttributeName(i);
              isPartOfQuery[i]=true;
            }
          }

        } else {
          selectVector.addElement(dd.getDBTable(i, false) + "." + dd.getAttributeName(i));
          select+=((select.length()>0)?", ":" ")+dd.getDBTable(i, false) +
                  "." + dd.getAttributeName(i);
          isPartOfQuery[i]=true;
        }
      }
      return select;
    }

// this heavyweight was refactored by using only ReplaceParameterWithMethod(292)
// this is a lightweight that was later moved to DatabaseFormat which
// deals only with the names of fields

    protected String createSelect(Kriterium krit) {
      boolean[] isPartOfQuery=isPartOfQuery(krit);
      StringBuffer select = null;
```

```
      Debug.dbg("DBServerImpl.query("+krit.getName()+")");
      DataDescriptor dd=krit.getDescriptor();
      String qualifiedName;

      for (int i= 0; i < krit.getAttributesCount(); i++) {
          if (isPartOfQuery[i]) {
              qualifiedName=
                  databaseFormat.createQualifiedName(
                     null,dd.getDBTable(i, false),  dd.getAttributeName(i));
            select=databaseFormat.concatExpressions(select,",",qualifiedName);
          }
      }
      return databaseFormat.notNullString(select);
    }

// this is also a lightweight that was later moved to DatabaseFormat
// it doesn't care about creating a select string but rather marks the fields
// necessary for the select

    protected boolean[] isPartOfQuery(Kriterium krit) {
      boolean[] isPartOfQuery=new boolean[krit.getAttributesCount()];
      DataDescriptor dd=krit.getDescriptor();
      Vector fromTableVector=new Vector();
      String table, reftable;
      Constraint c=null;

      for(int i=0;i<dd.getTables().length;i++)
        fromTableVector.addElement(dd.getTables()[i]);

      for (int i= 0; i < krit.getAttributesCount(); i++) {
        isPartOfQuery[i]=false;
        c=dd.getConstraint(i);
        if  (c.getTable() == null) continue;

        table=dd.getDBTable(i, false);
        reftable=dd.getDBTable(i, true);

        if (c.getType() == DataDescriptor.REF
            || c.getType() == DataDescriptor.BACKREF) {

            // nur korrekte Referenzen nutzen
            if(reftable !=null && table !=null ){

            // nur Tabellen nehmen, deren Spalten selektiert werden
            if(fromTableVector.indexOf(reftable) != -1 &&
               fromTableVector.indexOf(table) != -1){

              isPartOfQuery[i]=true;
            }
          }

        } else isPartOfQuery[i]=true;
      }
      return isPartOfQuery;
    }
```

A.4 The Two Subclasses of DatabaseFormat

```
// PROGRESS
package emw.metaworks.server;

import emw.metaworks.model.*;
import emw.metaworks.util.*;
import java.util.Date;
import java.util.Hashtable;

public class ProgressFormat extends DatabaseFormat {

  protected static UTCDateFormat dateFormat = new UTCDateFormat("MM/dd/yyyy");

  protected static String DELETE = "delete from";

    public String formatObject(Object obj) {
        if (obj instanceof Boolean) return (((Boolean)obj).booleanValue())?"yes":"no";
        if (obj instanceof java.util.Date || obj instanceof java.sql.Date ) {
            return "date('"+dateFormat.format(obj)+"')";
        }

        return super.formatObject(obj);
    }

    public  String createUpdateStatement(Data data, Object user_oid) {
        DataDescriptor dd = data.getDescriptor();
        String mainTable = dd.getDBTable(dd.getOidId(), false);
        String qualifiedName=createQualifiedName(dd.getDB(),mainTable,null);


        StringBuffer set=null, where = null;

          for (int j = 0 ; j < data.getAttributesCount(); j++){

            String attributTableName = dd.getDBTable(j, false);

            if (!mainTable.equals(attributTableName) || !data.wasChanged(j)){
              continue;
            }

            set=concatExpressions(set,",",
                  getExpression(dd.getAttributeName(j),
                                Kriterium.EQ,
                                data.getAttribute(j)));
          }

          where = createOidExpression(data.getOids(), dd);

          if (data.getLastModify()!=null) {

            set=concatExpressions(set,",",
                  getExpression("geaendert_von",
                                Kriterium.EQ,
                                user_oid));
```

```
            set=concatExpressions(set,",",
                  getExpression("geaendert_datum",
                                Kriterium.EQ,
                                new java.sql.Date(System.currentTimeMillis())));

            where=andExpressions(where,
                  getExpression("geaendert_datum",
                                Kriterium.LE,
                                new java.sql.Date(data.getLastModify().getTime())));
          }

          return  UPDATE + " " + qualifiedName      + " " +
                  SET    + " " + notNullString(set) + " " +
                  WHERE  + " " + notNullString(where);

    }

}

// ORACLE

package emw.metaworks.server;

import emw.metaworks.model.*;
import emw.metaworks.util.*;
import java.util.Date;
import java.util.Hashtable;

public class OracleFormat extends DatabaseFormat {

    protected static UTCDateFormat dateFormat = new UTCDateFormat("yyyy-MM-dd HH:mm:ss");

  protected static String DELETE = "delete";

    public String formatObject(Object obj) {
        if (obj instanceof java.util.Date) {
            return "to_date('" + dateFormat.format(obj) +
                           "','YYYY-MM-DD HH24:MI:SS')";
        }
        if (obj instanceof Boolean) return (((Boolean)obj).booleanValue())?"1":"0";
        return super.formatObject(obj);
    }

    public  String createUpdateStatement(Data data, Object user_oid) {
        DataDescriptor dd = data.getDescriptor();
        String mainTable = dd.getDBTable(dd.getOidId(), false);
        String qualifiedName=createQualifiedName(dd.getDB(),mainTable,null);


        StringBuffer set=null, where = null, values=null;

          for (int j = 0 ; j < data.getAttributesCount(); j++){

            String attributTableName = dd.getDBTable(j, false);
```

```
            if (!mainTable.equals(attributTableName) || !data.wasChanged(j)){
              continue;
            }

            set=concatExpressions(set,",", dd.getAttributeName(j));

            values=concatExpressions(values,",", formatObject(data.getAttribute(j)));
          }

          where = createOidExpression(data.getOids(), dd);

          if (data.getLastModify()!=null) {

            set=concatExpressions(set,","geaendert_von");
            values=concatExpressions(values,",",formatObject(user_oid));

            set=concatExpressions(set,","geaendert_datum");
            values=concatExpressions(values,",",formatObject(
                new java.sql.Date(System.currentTimeMillis())));


            where=andExpressions(where,
                  getExpression("geaendert_datum",
                                Kriterium.LE,
                                new java.sql.Date(data.getLastModify().getTime())));
          }

          return  UPDATE + " " + qualifiedName      + " " +
                  SET    + " (" + notNullString(set) + ") = " +
                  " ( select "  + notNullString(values) + " from dual ) " +
                  WHERE   + " " + notNullString(where);

      }

}
```

A.5 Comparison of the Code before and after the refactoring

To visualize the result of the refactoring of the database server class, I listed the code of the single class before the refactoring and the code of the 4 resulting classes after the refactoring.

Besides reducing the general number of lines, the resulting code looks much lighter as there are no more large blocks of code clumped together in a single method but rather small methods which either call other methods to delegate the work to do or containing only a few lines of computing code.

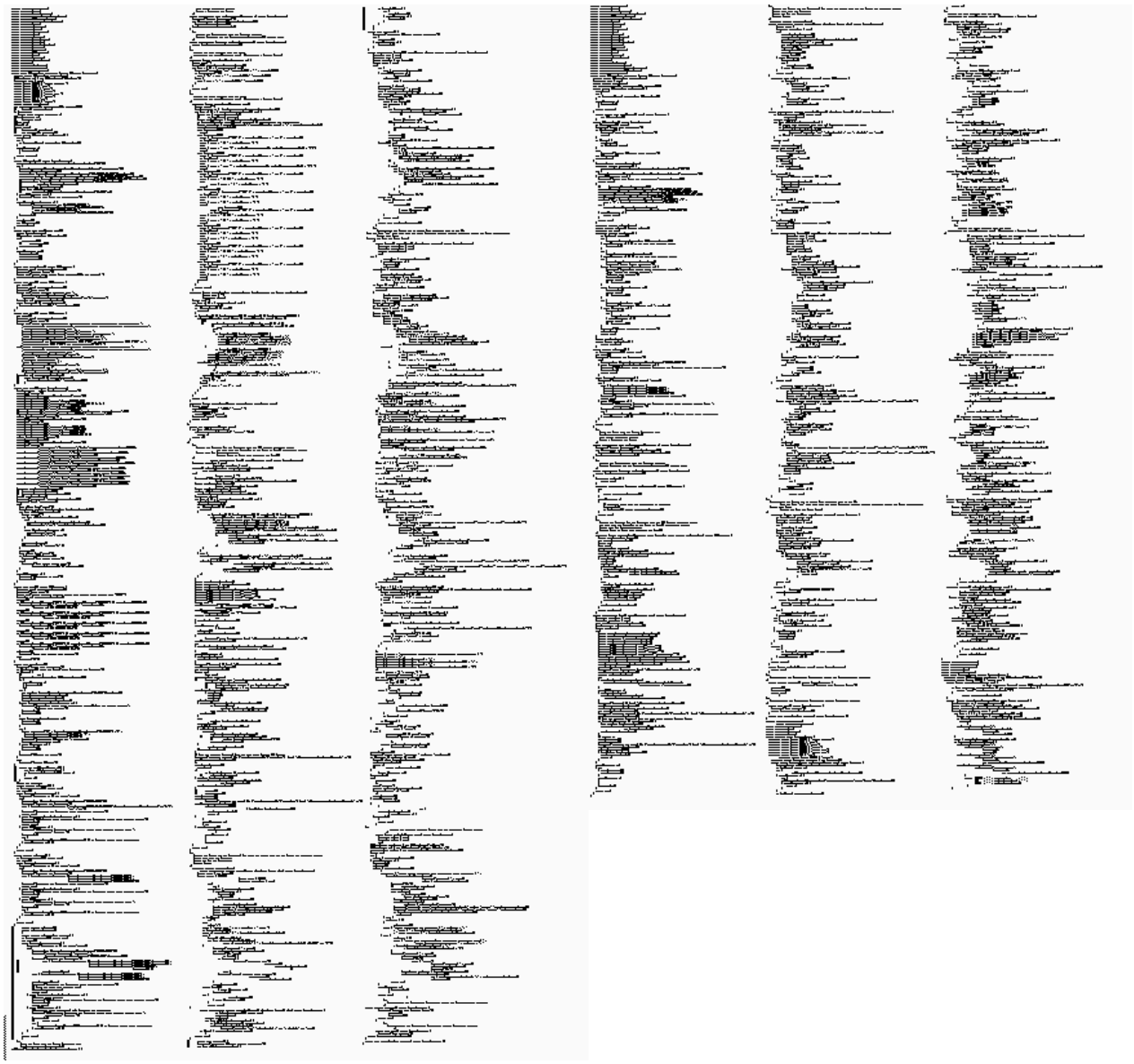

Figure A.1: Comparison of the Code before and after the refactoring

Bibliography

[Fowler] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Co., Inc, Reading, MA, 3rd printing Nov 1999.

The first comprehensive book about refactoring, most of this work depends on it. See also the online resources at http://www.refactoring.com, where also an extended online catalog of the refactorings mentioned in the book is found.

[Beck,Xp] Beck, Kent. *eXtreme Programming eXplained: Embrace Change.* Addison-Wesley Co., Inc, Reading, MA, 2nd printing Nov 1999.

Kent Beck shows convincingly how simple programming principle combined make up a great programming process.

[Gang of Four] Gamma, Erich., Helm, R., Johnson, Ralph, Vlissides John. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Co., Inc, Reading, MA, 1999.

This is the book where four people gave the expert knowledge possessed by every programmer a shape - Patterns. It was the first step in a very exciting development.

[UML+Patterns, Larman] Larman, Craig *Applying UML and Patterns*

This book concentrates on the introduction of an iterative, incremental development process which is based on UML techniques and notations for documenting its results and on Patterns for formalizing the steps from analysis to design.

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

The first decent-length writing on refactoring [Fowler]. Get it at: ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z.

[Gamma, JUnit] Gamma, Erich Beck, Kent. *JUnit Open Source Testing Framework.*

See http://members.pingnet.ch/gamma/junit.htm for an easy introduction to unit testing by the authors of the testing framework JUnit.

[Refactoring Browser] Brant, John Roberts, Don. *Refactoring Browser Tool.*

Unfortunately to this time only available for the Smalltalk Programming language at http://chip.cs.uiuc.edu/users/brant/Refactory/.

[Wake, JUnit] Wake, William. *A XP session.*

On http://users.vnet.net/wwake/xp/xp0001/index.shtml William Wake walks through a sample session of using unit tests to test an implement an simple User Interface.

[Wiki] *The Wiki Pages about Refactoring*

See http://c2.com/ppr/wiki/WikiPagesAboutRefactoring, for lots of material in form of jointly edited web pages to which many experts of the area contributed.

[TAPOS] Don Roberts, John Brant, and Ralph Johnson. *The Theory and Practice of Object Systems: Supporting Software Evolution with Automated Refactorings: A Refactoring Tool for Smalltalk* University of Illinois at Urbana-Champaign, Department of Computer Science, 1997.

http://st-www.cs.uiuc.edu/ droberts/tapos/TAPOS.htm

[JRefactory] *JRefactory* by Chris Seguin.

A tool licensed under the GPL which allows applying refactorings using a generated UML-diagram interface. Available at: http://users.snip.net/ aseguin/chrisdown.html.

[JBuilder IDE] Borlands Java Development Environment *JBuilder*

Look at http://www.borland.com.

[Elixir IDE] The Java IDE *Elixir*

A very powerful environment written in Java for developing Java applications. It includes features like version control, template driven class generation, auto expansion, incremental obfuscation etc. Available at: http://www.elixirtech.com/ElixirIDE/

[Renamer] The *Renamer* tool from IntellJ Software.

The Renamer can be downloaded from http://www.intellij.com. It is mainly useful for finding occurrences of and references to symbols within Java source files and for performing operations such as renaming on them.

[Xref-Speller] The *Xref-Speller* a Plug-In for Emacs by Marian Vittek.

The Xref-Speller is actually a front end for the xref program, which allows it to perform operations on the symbols of Java programs with the use of an extracted parse tree. A trial version can be found at: http://www.xref-tech.com.

[JDK] The *Java Development Kit* (actually 1.3) by Sun Microsystems.

Much information about the JDK as well as current releases are available at Sun's web site http://www.javasoft.com.

[Nygard] Michael T. Nygard Tracie Karsjens, *Test infect your Enterprise JavaBeans* JavaWorld, May 2000.

The article features a very convincing introduction to unit testing and covers the extension of the JUnit [Gamma, JUnit] for testing Java Enterprise Beans in the application server environment. Can be found at: http://www.javaworld.com/javaworld/jw-05-2000/jw-0526-testinfect.html.

[Refactoring, Hunger] Michael Hunger, Thesis:*Refactoring - Benefits and Disadvantages of an Amazing Technique*, Dresden, Oct 2000

All documents related to this paper can be found at http://emw.inf.tu-dresden.de/ mh14/refactoring.