Roots of Refactoring

Jan Philipps and Bernhard Rumpe

Institut für Informatik,
Technische Universität München,
D-80290 München

www.in.tum.de/~{philipps,rumpe}

September 18, 2001

Abstract

Refactoring is a new name for a transformational approach to iterative software development. Originally focused on class diagrams, it is now commonly associated with object-oriented programming languages like Java. In this article, we trace some of the conceptual roots and the ideas behind refactoring, and sketch its relation to other techniques, such as behavioral and structural refinement or compiler optimization. Based on these observations, we firmly believe that improved and adapted refactoring techniques will belong to the methodical tool set of tomorrow's software engineers.

1 Introduction

Rarely is something invented in a "big bang". Ideas evolve over time, are influenced by a number of groups and individuals, are applied to different domains, are integrated to and re-emerge from a variety of existing techniques. Finally the time may come to pin down an idea into an abstract form and to give it an appropriate name.

Computer science is no different. In particular in its discipline of software engineering, reinvention is common, as this discipline deals with immaterial artifacts rather than with a given world of observable phenomena.

Refactoring is such a concept. It was made prominent basically by Martin Fowler [13], based on the programming language Java [16]. In this article, we first look at the basic principles of refactoring (Section 2) and give an outline of related techniques that demonstrate that these basic principles are present within other techniques as well (Section 3). We then take a —somewhat subjective— look at a few of these techniques in greater detail (Sections 4 to 6). In Section 7 we point out a few questions that need to be addressed for refactoring techniques and small-cycle iterative development techniques to become indispensable in modern software engineering.

2 Refactoring

The concept of refactoring (and also the word "refactoring" itself) was coined already several years ago (see e.g. [22]), but its breakthrough came with the integration of refactoring into the software

[PR01] J. Philipps, B. Rumpe.
Roots of Refactoring.
In: Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15, 2001.
Kenneth Baclavski, Haim Kilov (eds.).
Northeastern University, 2001.
www.se-rwth.de/publications

development process *Extreme Programming* [4]. In fact, Fowler himself contributes much of the ideas of refactoring to Ward Cunningham and Kent Beck. In [13, pp. 53f], he defines *refactoring* as follows:

Refactoring (noun) A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing the observable behavior of the software.

Refactor (verb) To restructure software by applying a series of refactorings without changing the observable behavior of the software.

Fowler informally applies refactoring techniques to the programming language Java and explains the structural changes through exemplifying class diagrams. He presents 72 refactorings in his book, among them "extract class", "move field/method", "introduce explaining variable", "replace delegation with inheritance", or "substitute algorithm".

All refactorings are presented in the same systematic format: The name of the refactoring, a short summary, a motivation, the mechanics and examples. The two most important sections are motivation and mechanics. The motivation includes a problem description that allows a programmer to match his problem to the refactoring and to understand whether the refactoring will solve this problem. The mechanics section lists a series of concise steps to be applied when carrying out the refactoring. These steps a presented in a constructive manner, such that they can immediately be applied.

Refactoring in the sense of Fowler [13] can be characterized by the following statements:

1. Refactoring deals with internal *structure of software*. Thus, the techniques of refactoring are applied to programming artifacts such as source code.

2. Refactoring explicitly *preserves the observable behavior*. This demonstrates that although refactoring primarily deals with structure, it cannot disregard behavior.

3. Refactoring aims at improving a given situation according to a given informally expressed goal; examples for such goals are reduction of development costs, or improvement of readability, maintainability, speed of execution, or memory demands.

4. Refactoring steps are rather small and can systematically be combined to more powerful *sequences* allowing to build sophisticated tactics to achieve ambitious goals.

5. Refactoring is a constructive, rule based technique that starts with a given situation, a goal and a series of constructive steps, such as "move a to b, then rename c, then check d" to achieve that goal.

6. Refactoring is applied by software engineers. Refactoring techniques are designed to be applied manually. However, there are attempts to implement tool assistence, such as the refactoring browser [6].

7. The correctness of an application of a refactoring rule is in the responsibility of the developer. In the XP process application of refactoring rules is assisted by tests to increase confidence in the correctness of the rule application. However, there is (currently) no proof system that allows to formally prove correctness – neither automatic nor interactively.

As mentioned in this list, refactoring as presented in [13] only deals with behavior preserving transformations. Progress in the evolution of the system design is defined only informally: The new

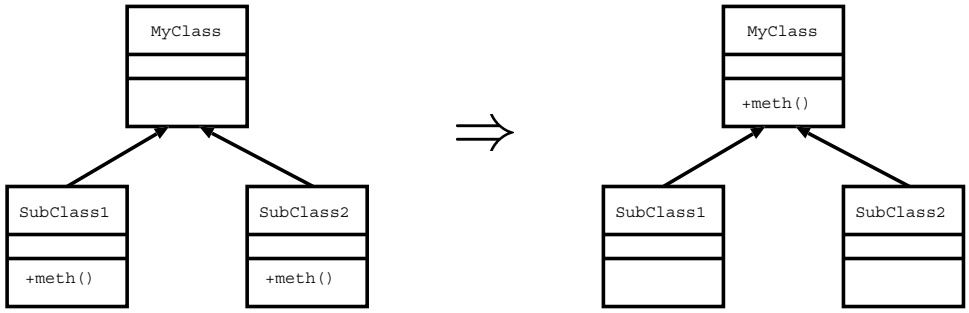


Figure 1: Refactoring Example ("Pull Up Method")

system should be easier to understand, to maintain and to extend. The guiding principles behind this are based on well-known, but informal and sometimes conflicting programming heuristics.

Not only the applied measure in refactoring is informal, however. The correctness of the transformation steps in the sense that they preserve system behavior is not formally justified. In fact, necessary context conditions are all too often left implicit. For example, Figure 1 shows the well-known refactoring step that lifts common methods to a superclass ("Pull Up Method", [13, p. 322]) has the obvious side condition that the methods `SubClass1.meth()` and `SubClass2.meth()` are behaviorally equivalent. This is a rather strong requirement, as verification of object-oriented programs is highly nontrivial. A sufficient, but very strong criterion is obviously syntactical equality which at least can help to remove the redundancy that results from cut-and-paste of source code.

In the context of Extreme Programming, this lack of formality is somewhat alleviated by a rigorous testing discipline. In a pragmatic sense, two methods can be regarded as equivalent, if they pass the same (rather complete) set of test cases.

Not all refactorings need to be justified by formal reasoning: For example, if just `SubClass1.meth()` is moved to the superclass while `SubClass2.meth()` remains in its subclass, the proof obligation can be omitted, because overriding preserves the previous behavior.

3 Transformational Approaches

In the last section, we noted that refactoring steps improve the system structure under a given metric, while the observational behavior of the system remains unchanged. Obviously, this is only true for appropriate and usually rather coarse notions of observable behavior. If execution time is considered part of the observable behavior of a system —as is the case for real-time process control systems—, a refactoring step that extends execution time can be regarded as a critical change in behavior.

Since there is some variability in the choice of behavior notion and goal metric, it seems reasonable to relax the requirement of observational equivalence somewhat, and to allow or even enforce certain "improvements" in behavior as well as optimization of non-functional goals. This generalization puts refactoring close to the well-known concept of *refinement*, as pioneered by Dijkstra [12], Wirth [34] and Bauer [2].

Although many approaches use the concept of behavior preserving or refining transformations, the first approaches to explicitly make use of behavioral equivalence and refinement were algebraic

specification techniques.

For example, OBJ [14] employs *hidden sorts* that allowed to explicitly distinguish between internal and externally visible behavior. In [15], Goguen describes this approach and its implications to the preservation of externally visible behavior from a current perspective. Most important, this and other approaches (e.g., Spectrum [9]) show,that it is possible to explicitly define "externally visible behavior" and base rigorous proof techniques on this definition. The standard refactoring approach, however, only uses an informal and implicit notion of behavior defined through its tests. In the sense of Goguen these tests are "experiments" on the system, which are possible since the probed functionality is externally visible. In practice, however, the tests defined for a system are usually based on different visibility assumptions. Method tests are more fine-grained than unit tests and can therefore see more details. This becomes apparent, when a (local) method is refactored and its method tests are not appropriate anymore. External tests instead still apply.

Let us now give a few examples for refinement techniques from different computer science areas. We do not attempt to give a complete overview, but mainly try to show variants of applications of this principle, where refactoring is only one of the most recent and prominent.

Behavioral refinement of state machines as shown in [28, 31] and also as an example in [29] has a large variety of variants. State machines (in various incarnations such as Statecharts [18], I/O Automata [21], ω-automata, Mealy and Moore machines and many others) describe component or system behavior rather than system structure. Manipulation of statemachines therefore directly affects behavior, and the preservation of behavioral equivalence normally would be too restrictive. Instead, the idea is to add details to derive concrete implementations from abstractly specified behavior.

In Section 6, we will give a short overview over such state machine transformations.

Refinement of dataflow architectures as discussed in [24, 25] describes a number of transformational rules on a software architectural language that is suited for distributed systems with asynchronous communication. Some of the transformations just improve the structure of a system, while others also affect the behavior of the system at its interfaces. A clearly defined notion of *observable behavior* allows that approach to precisely define what preservation and refinement of behavior means.

Section 5 discusses this approach in more detail, and gives an example of such a set of transformations.

Refactoring of Class Diagrams by William Opdyke [22] showes how to migrate functionality and attributes between classes as well as how to merge and split classes on a class diagram notation (essentially a subset of today's UML [17]). The goal is to improve the design of the system for further maintenance and extension.

Refinement Calculus [1] is a framework for the stepwise derivation of imperative programs from a specification, based on early work of Dijkstra [12] and Wirth [34]. As a verification methodology, refinement calculus is quite successful; as a software development methodology, it has its weak points, as pointed out by Michael Jackson [19]: "You must already have solved the problem before the solution process is begun".

Computer-Aided Intuition-Guided Programming (CIP) was a larger project led by F. L. Bauer at the Technische Universität München, one of the organizers of the famous conference on Software

Engineering 1968 in Garmisch, near Munich.
The project developed a wide-range language [3] that included several sublanguages for specification, functional, applicative, imperative and assembler programming. Its main purpose was to allow interactive transformation of an abstract specification into an efficient and executable program. There are steps involved that strongly remind to refactoring, but they do have a very precise and formal underpinning.

In Section 4, we will show a a number of CIP-transformations on program structure.

Common to all these approaches is that —like refactoring— they are conscious design decisions. While in principle refactorings can be automated, for instance in program transformation systems, such automatization has so far failed to enter mainstream programming practice.

In compiler design, however, automatic refactorings are ubiquitous: Already early FORTRAN compilers offered optimizations based on rewritings of the program, and elaborate optimization phases are state-of-the-art in compiler technology for imperative programming languages. They preserve the functional behavior, while improving execution time and memory usage. Some optimizations try to reduce branching or to optimize register use; others apply on the source code level: Algebraic transformations based on solid mathematical semantics of a programming language allow to transform expressions such as $a + b - a$ to b, $a * 0$ to 0 etc. Transformation steps are at the core of modern compilers for functional programming languages [20]. Tail-recursion elimination, where recursion can be translated into iteration, is a common optimization of functional programming language compilers; it is also one of the transformations known from the CIP project (see [2]).

Transformations that preserve behavior or correctness are by no means an original invention of computer science. Arguably the most ambitious "refactoring" so far undertaken is the unified presentation of mathematics of the group of mathematicians known as Nicolas Bourbaki [11].

Mathematicians use refactoring-like techniques also on a smaller scale: Given a proof for a theorem, it is always worth searching for a more beautiful or shorter proof. The application of mathematical calculi can also be regarded as refactoring. Solving an equation in order to find the value of a variable is done by stepwise transformations that preserve the value of the variable until finally the equation has the form $a = \ldots$ explicitly showing the value.

4 The Munich Project CIP

We now briefly sketch the Computer-Aided Intuition-Guided Programming project (CIP) and relate its ideas from over 20 years ago to modern concepts and languages. Some of the project's main results are published in [3] and are strongly connected to [2].

The central theme of the CIP project was to develop programs by a series of small, well understood transformations. Beginning with an abstract specification written in an algebraic style, the transformation steps lead to an executable and efficient program. A precise underlying semantics ensures the correctness of the applied steps.

The CIP project was based on its own language. The applicative and imperative parts where inspired by languages from the Algol family. Object-orientation was just about to emerge at that time and therefore was not directly incorporated —it only showed up in form of "devices" that couple the data and module concept. In particular, a module concept (influenced by Parnas, [23]) and strong concepts for the definition of abstract datatypes are present.

Beyond the programming and specification language, the project used a rule-based language to describe transformations. A transformation is specified as an *abstract schema*, using so called *schema variables* that replace parts of the target language. Here is an example for a simple rule to eliminate a conditional statement:

$$\frac{\text{if } E \text{ then } A \text{ else } B}{A} \;\Big[\!\Big[\; \text{Boolean expression } E \text{ is a tautology}$$

This rule uses scheme variables E, A and B to identify parts of the expression to be transformed. The side condition of the rule states that the transformation is only valid if the Boolean expression $\mathbb{E}$ is equivalent to "true". This is a typical rule that is close in spirit to the refactoring rules of [13]. In a similar vein, CIP provides rules for algebraic optimization, control structure manipulation, folding and unfolding of functions, and for the change of data structures.

$$\frac{E \;+\; E}{2{*}E} \;\Big[\!\Big[\; \begin{array}{l}\text{Expression } E \text{ is side-effect}\\ \text{free and deterministic}\end{array} \qquad \frac{V := E;\; S}{S;\; V := E} \;\Big[\!\Big[\; \begin{array}{l}\text{Variable } V \text{ unused in statement}\\ S\text{; expression } E \text{ is side-effect free}\end{array}$$

(a) (b)

Figure 2: CIP Transformation Rules

Algebraic optimization mainly occurs with expressions, or folds and unfolds parts of an expression. An example for a simple algebraic optimization is shown in Figure 2(a). With this rule, `3+3` can be replaced by `2*3`. However, it can not be applied to `(i++)+(i++)`, as this expression is not free of side effects. Neither can it be applied to `Math.random()+Math.random()`, as this expression is not deterministic. This example also shows that many of these rules apply in both directions.

Figure 2(b) shows an example of control structure manipulation. It deals with the reordering of program statements. There are many more sophisticated rules, in particular rules to treat branches, loops, or rules that fold statement sequences into procedures; these rules also deal with result assignment, side effects, and other peculiarities.

The rules shown so far deal with structural or algebraic manipulations that preserve observable behavior. CIP also provides rules for refining behavior. Such a transformation adds details to an abstract specification of a program, e.g. describing not only the desired outcome, but also how to calculate the result. This however is only possible if the artifact to be manipulated is abstract in the sense that it allows several different behaviors and implementations. For this purpose CIP uses an abstract specification sublanguage that allows declarative formulation of program properties. Interestingly, this sublanguage has some conceptual similarities to OCL [33].

An example of the use of behavior refinement is a specification that describes that each object in a given set shall receive a method call. Since sets usually are unordered while the effect of the specified operation may depend on this order, there is some natural underspecification. We can apply CIP transformations to replace such a set by an appropriate implementation (e.g. `SortedSet` in Java) that allows us to determine the order. Therefore we transform the previously underspecified definition into an executable program.

When looking at the characteristics of CIP transformations, we find —apart from the different programming language— some differences to the refactoring approach:

- Rules are used both for refactoring and for refinement.
- Rules are precisely specified and have explicit context conditions that fit to the underlying semantics.
- CIP rules are used not only for improving existing code, but for deriving new code from abstract specifications.

This makes an important difference in methodical use, as CIP transformations are designed to constructively assist iterative step-by-step development. To further improve assistance of iterative development, the small transformational steps can be combined into powerful *tactics*. Such a tactics can be used for algebraic optimizations, repetition of a series of steps, or can even be understood as the explicit manifestation of a design pattern in procedural form. In this respect CIP has a more fine grained iterative development process than even XP.

5 System Structure Refactorings

Transformational approaches are not limited to programming languages. In this section, we demonstrate distributed systems can be refactored at the architectural level of distributed systems.

We model the distributed system as a network of component that communicate asynchronously over buffered unidirectional channels. The message exchange between components of a distributed systems is represented by message streams, finite or infinite sequence of data. Each message stream represents the communication history over a channel between two components.

The behavior of a system component is then modeled by a relation between its input and output communication histories. We impose a number of restrictions that ensure that the component behavior is causally correct, i.e., the component output may not depend on future component input (see [10] for details).

Based on streams and I/O history relations, a precise notion of an architecture for distributed message-passing systems can be defined. A system consists of input and output channels, a set of components and a connection structure that satisfies the following restrictions: Components have no common output channels, each component input is either a system input or a component output (possibly of the same component), and each system output emerges as an output of one of its components (Figure 3). Under these assumptions, the behavior of a system is precisely defined by the intersection of the component I/O relations; hiding of internal channels is accomplished by existential quantification. A system where all internal channels are hidden can be itself regarded as a component. Therefore, systems can be composed hierarchically.

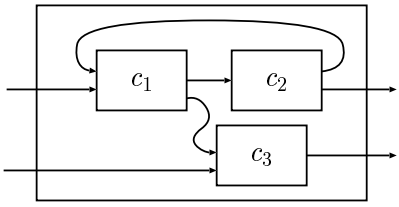


Figure 3: Message passing system

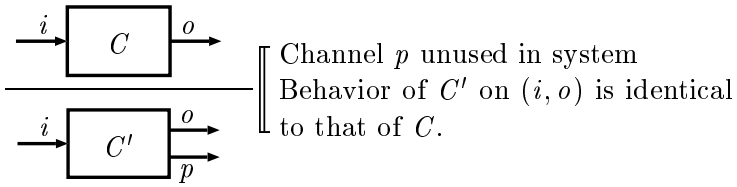
Since component and system behavior are described as relations of input and output histories, there is a natural refinement concept based on the behavior subset relation. Elaborate refinement rules for behavioral and interface refinement can be defined [10].

These refinement rules are based on an abstract syntax of component specifications and composition operators. For practical application in an incremental development process, refactoring rules that are based directly on the graphical representation of a system architecture are more useful. In [24, 25], we introduced such a set of rules, that allows a system designer to

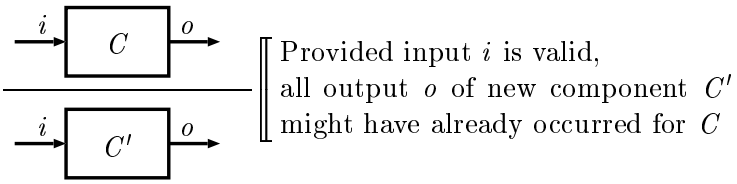
- introduce and remove system components,
- introduce and remove component input channels,
- introduce and remove component output channels,
- refine component behavior, possibly under consideration of an invariance predicate that characterizes the behavior of the other system components, and to
- replace a component by a subsystem and vice versa.

Each of these rules preserve the restrictions on the system architecture mentioned above. Some of the preconditions of these rules are syntactical ("the output channels of a component to be removed may not be used by other components"), some of them refer to system behavior invariants.

To show the flavor of the rules, the following rule is used to introduce a new component output channel p for a component C of a distributed system:



Note that the behavior on the new output channel p can be arbitrary; this introduces underspecification which can later be removed by behavior refinement, for instance using this rule:



Here, a system invariant is used as a predicate over streams that characterizes the valid input histories of the system. [25] contains a more formal presentation of this rule and a justification of its correctness.

Figure 4 shows the transformation of a simple data collection system. The component PRE gathers data, preprocesses it, and sends it to a remote database RDB. To reduce the required transmission bandwidth, in a new version of the system only the difference of the current data to the previous data shall be transmitted. The six structure diagrams show the necessary transformation steps: Introduction of encoding and decoding components (ENC and DEC), connection of the new component to the existing system, elimination of the previous connection between PRE and RDB, and the folding of PRE and ENC as well as DEC and RDB to new components.

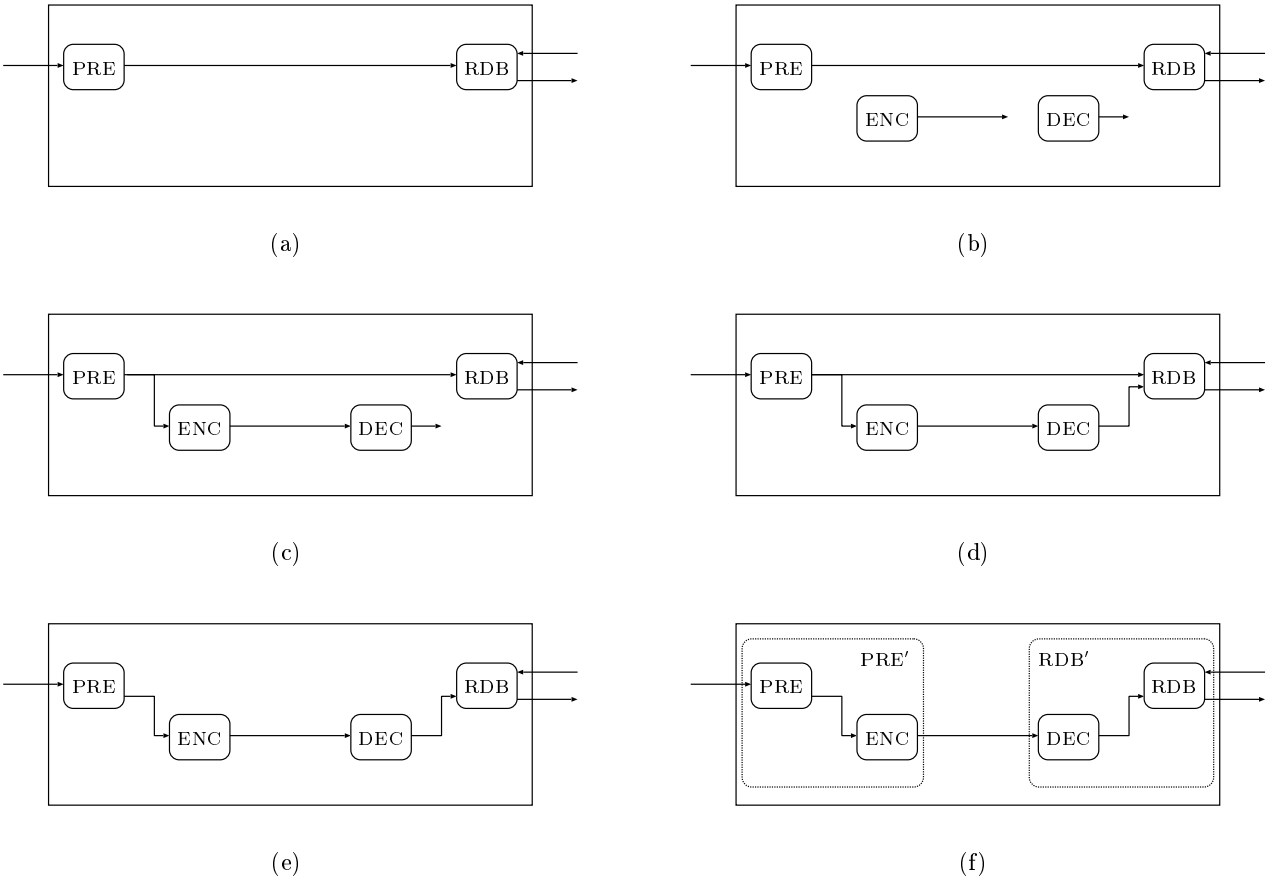


Figure 4: System Structure Transformation

Most of these steps are purely syntactical; only the step from Figure 4(d) to Fig. 4(e) requires some formal reasoning to show that the encoded data is essentially the same as the unencoded data (see [25] for a formal proof, it is based on the behavior refinement rule presented above).

The example also shows how to assemble the transformation rules to more complex domain specific transformation pattern. There are only very weak assumptions about the behavior of PRE and RDB, which means that this example can be used without further proof obligation for similar situations.

The architectural transformation rules are similar to the CIP transformation rules in that they encompass not only refactoring but also refinement; they are also based on a precise semantics so that the rules can be give explicit and precise context conditions.

Architecture refinement is by no means limited to software systems, but may be applied to business models or organizational models as well; [30] shows an example for the transformation of business processes.

6 State Machine Refactorings

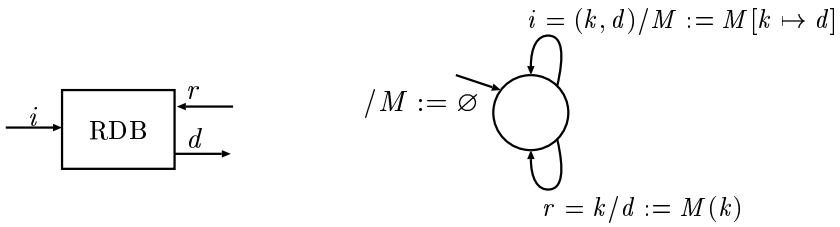
During the last years, it has become common practice to describe the behavior of system components by state machines, which are described graphically by state transition diagrams [8, 28] or, extended with hierarchy and parallelism, by Statecharts [18, 17].

For components in distributed systems, state machines can be given a formal semantics based on streams that is compatible with the architecture model of the previous section [28, 8].

This semantics can be used to give a set of refactoring and refinement rules that, among some other transformations, allows us

- to add new states and to remove unreachable states,
- to add new input messages to a component,
- to remove transitions, provided there are alternative transitions for the same input,
- to add transitions, provided there are no existing transitions for the same input.

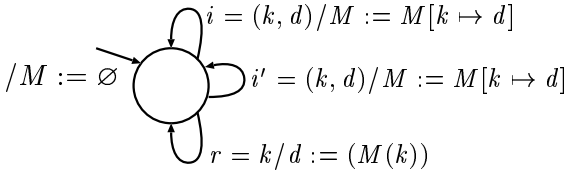
As an example, below is the component interface and a —very much simplified— state transition diagram for the remote data base of the example in Figure 4:



Here, M is a assumed to be the data base; it is initially empty. New data (k, d) arrives as pairs of data and key on the input channel i and is stored in M by updating M. Queries arriving on the request channel r are answered by looking up the corresponding data and forwarding it over o.

The state machine refinement rules can be used to capture the step from Figure 4(c) to Figure 4(d) in the example, where a new input channel is added to the data base:

- The state machine's input signature is extended from $\{\ i, r\ \}$ to $\{\ i, r, i'\ \}$ for the new input channel which carries the data from the decoder. The justification of this step is that while the behavior of the machine for input on i' is undefined, its behavior for input on r and i is identical to the original behavior.
- In the next step, the behavior for input on i' is defined. For the simple example here, input on i' is treated exactly as input on i: The decoding of the data occurs already in the separate component DEC.



Note that these two transformation steps are likely to be used together in practice: While the first step introduces internal underspecification in the system (because the behavior for input on he new channel is undefined), the second step immediately restricts or even removes it again, thus making the new channel "useful".

The step from Figure 4(d) to Figure 4(e) removes the old input channel i of the data base. While this step should seemingly be symmetric, it is much more ambitious. In general, input channels cannot be removed without deeply changing component behavior. In this case, however, we can show as a system invariant that the same data arrives on channels i and i' (for instance, using the verification techniques for I/O history specifications of [7]); removing the transition that reads from i therefore leaves the data base behavior unchanged, and it is safe to remove the input channel i altogether.

For a more formal discussion of the state machine transformation rules, see [28]; a similar rule has been developed for a Statecharts dialect [31].

7 Conclusion

In the previous sections, we have presented various approaches of system transformation that are quite similar to refactoring techniques in the sense of Fowler. Common to these approaches is that the artifact or pieces of the artifact that is being developed are changed in small and systematic steps, where each step improves the system according to a —not necessarily formalized— metric, such as maintainability, reduction of underspecification, speed, memory consumption or even simply esthetics.

Refactoring steps can also be based on mathematical models. Each refactoring step can then be given precise context conditions that have to be justified in order for the step to be applicable. These mathematical approaches generalize to *refinement* transformations, where behavior descriptions can be specialized during the development process. Of course refinement is not limited to programming languages or the pipe and filter architectures used in this paper, but they can be applied to a variety of styles, such as interpreters, communicating systems or event based systems [32].

The use of a series of small and systematic transformations during the software development does not fit into the waterfall or the spiral models of software engineering [5]. Instead it corresponds to a short-cycled, iterative method. It is therefore no coincidence that refactoring became prominent together with Extreme Programming [4], where development cycles are kept as short as possible.

The experience with transformation rules for structure and state transition diagrams lead us to believe that the core concept of Extreme Programming —systematic and small steps to improve the final result for certain goals— can be applied for a large number of modeling techniques beyond mere programming languages.

And indeed it should: While refactoring and Extreme Programming somewhat helped break the stasis of the established large-scale development processes, the scalability of XP is rather doubtful. An adaption of the XP and refactoring principles to high-level modeling techniques, coupled with code generation from models can extend the reach of small-cycle incremental development approaches.

Unfortunately, for the currently emerging standard UML [17] there is still a deficit of accepted transformation techniques; perhaps first the modeling and refactoring power of UML tools has to be improved. With emerging improved tool assistance and better understanding of refactoring techniques of various modeling and programming languages (not only UML), development cycles will become even shorter and systems redesign more flexible.

Another deficit of the UML is the inadequate mathematical foundation; there is no commonly accepted formalization that could be used to establish the notion of behavioral equivalence that is at

the core of refactoring. It might therefore seem worthwhile to more closely follow XP and to start with an indirect equivalence notion based on test case specifications; current work on model-based test sequence generation techniques (e.g. [26, 27]) could offer some machine assistance in this respect.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer, 1998.

[2] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer, 1982.

[3] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] B.W. Boehm. A spiral model of software development and enhancement. *Software Engineering Notes*, 11(4), 1994.

[6] J. Brant and D. Roberts. Refactoring browser tool. `http://st-www.cs.uiuc.edu/~brant`, 2001.

[7] M. Breitling and J. Philipps. Step by step to histories. In *AMAST 2000, LNCS 1816*, 2000.

[8] M. Broy. The specification of system components by state transition diagrams. Technical Report TUM-I9729, Institut für Informatik, TU München, 1997.

[9] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technical Report TUM-I9312, Technische Universität München, 1993.

[10] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer, 2001.

[11] J. Dieudonne. *Mathematics - The Music of Reason*. Springer, 1992.

[12] E.W. Dijkstra. Notes on structured programming. In C.A.R. Hoare O. Dahl, E.W. Dijkstra, editor, *Structured Programming*. Academic Press, 1971.

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

[14] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, pages 52–66, 1985.

[15] J. Goguen. Hidden algebra for software engineering. In *Conference on Discrete Mathematics and Theoretical Computer Science*, volume 21 of *Australian Computer Science Communications*, pages 35–59. Springer, 1999.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[17] UML Group. Unified Modeling Language. Version 1.4, Object Management Group (OMG), www.omg.org, 2001.

[18] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[19] M. Jackson. *Software Requirements and Specifications - a lexicon of practice, principles and prejudices*. Addison Wesley, 1995.

[20] S.L. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[21] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2):219–246, 1989.

[22] W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[23] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[24] J. Philipps and B. Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, *ICFEM'97.* IEEE CS Press, 1997.

[25] J. Philipps and B. Rumpe. Refinement of pipe and filter architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.

[26] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, 2001.

[27] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, 2001.

[28] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* PhD thesis, Technische Universität München, 1996.

[29] B. Rumpe. A Note on Semantics (with an Emphasis on UML). In B. Rumpe H. Kilov, editor, *Second ECOOP Workshop on Precise Behavioral Semantics*, I9813. Technische Universität München, June 1998.

[30] B. Rumpe and V. Thurner. Refining Business Processes. In Ian Simmonds H. Kilov, B. Rumpe, editor, *Seventh OOPSLA Workshop on Precise Behavioral Semantics*, I9820. Technische Universität München, June 1998.

[31] P. Scholz. *Design of Reactive Systems and their Distributed Implementation with Statecharts.* PhD thesis, Technische Universität München, 1998.

[32] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[33] J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison Wesley, Reading, Mass., 1998.

[34] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.