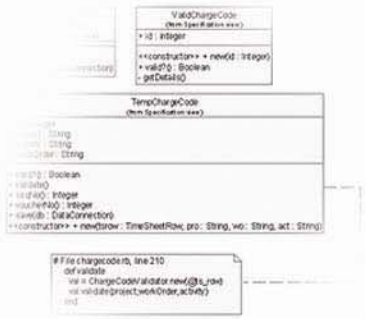
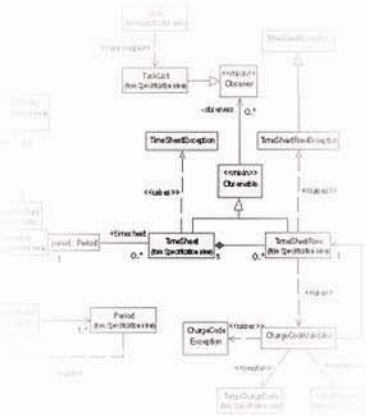
UNIVERSITY OF OSLO
Department of Informatics

RAPID PROTOTYPING USING UML AND DYNAMICALLY TYPED LANGUAGES

Erik Inge Marcussen





Cand. Scient. Thesis

October 2002

PROTOTYPING USING DYNAMICALLY TYPED LANGUAGES AND UML

By

Erik Inge Marcussen

A thesis submitted in partial fulfilment of the requirements
for the degree of

Cand. Scient.

University of Oslo

October 2002

_____

_____

_____

Date _____

UNIVERSITY OF OSLO

ABSTRACT

PROTOTYPING USING DYNAMICALLY TYPED LANGUAGES AND UML

By Erik Inge Marcussen

This thesis explores the idea of combining languages for specification and experimentation in mixed approaches of software development. Extending on the idea that a mixed approach of analytical and experimental behaviour countermeasures the weaknesses of the sole use of one approach, the thesis concentrates on how to combine two languages typically used for each approach. A method for combining two such languages, Ruby and the Unified Modelling Language, is developed. This method is then applied in a real prototyping and design effort and evaluated.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to thank Birger Møller Pedersen for the professional mentoring given throughout the development of the thesis. As a contributor to the OMG UML revision taskforce, he has given me invaluable insight into UML both in its current form and how it is developing into version 2.0. Especially I want to thank him for the motivation I needed to continue when the original research project was put to an end, and for helping to form a new objective and problem for the thesis.

At Unit 4 Agresso I would like to thank Michel van de Veen and Jens Glattetre for introducing me to Ruby and the original idea of combining it with UML in a code generation effort. Further I must thank Tom Gooch and Ole Strandengen for allowing me to perform my studies at Agresso. Thank you also to everyone that has contributed to the experiment, giving me insights to the problem domain. To Kristoffer Berg, thanks for the patience you have shown and the flexible work hours throughout the project.

To Erik Syversen, Afsheen Ali and Tonje Myrvang Viken, thank you for offering your time reviewing the thesis in the hectic final stages.

I also would like to thank my family, who has supported and encouraged me. Finally thank you to my Trine for showing love, patience and for motivating me throughout the last two years.

1 INTRODUCTION

Designing good computer based systems is both challenging and difficult. The software developers are expected to deliver the right system at the right time, and at the right cost. They also need to appreciate and understand the work practices and qualities of the user organization. They have to be up to date on and find the best uses of the latest technology, which changes with ever increasing speed. Contracts, requirements, ideas and preferences change as they perform their work. At the same time the complexity of the problems being solved by computers increases as computer technology is applied in more and more areas. The time when computers were primarily used to automate known manual routines is over. Information Systems, workflow management, E-Commerce, ERP, groupware and Computer Supported Cooperative Work are examples going beyond the early uses of computers.

The problems and challenges of developing software can be categorized as elements of complexity and uncertainty [Mathiassen90]. Methods, approaches and tools to deal with such problems have been researched since the industry first emerged. Since the problems are interrelated and vary throughout the development process [Mathiassen92], mixed approaches, accounting for both complexity and uncertainty are recommended [Mathiassen95].

These approaches have different tools to support them. Object orientation as an approach to handle complexity was introduced in the 1960s with the Simula language. However it was not until the end of the 1980s that development methods for system analysis, design and implementation based on object-orientation was adopted by the industry. The technology has proven to be able to handle the complexity of the real world and helps developers abstract and structure concepts into specifications and implementations. Both modelling languages and programming languages have evolved as a result of this that has become industry standards for specification and programming notation. Related to the problem of understanding and formalizing the reality is to know if your abstractions and understanding about the problem is the right one. An approach to handle such uncertainty is to learn by doing. Often a prototype of the system is developed to envision the system and check if the specification is a feasible solution. Also here object-oriented languages can be used.

Originally I was participating in a research project on code generation from object-oriented models. The modelling language was the Unified Modelling Language (UML), an industry standard for object-oriented modelling. The problem was that without support for actions, 100% code generation from UML models is impossible. The idea was to use translational code generation, meaning that the UML object models would be translated into an Abstract Specification Language (ASL), where actions would be specified before generating to the target programming language. The ASL was based on the object oriented scripting language Ruby. A mapping from UML to the ASL had to be developed so round-trip engineering between the two could happen. In an envisioned tool it should be possible to switch between UML and the ASL so that a change in one language was reflected in the other. The ASL should also be executable, making it possible to demonstrate the feasibility of the design before generating code. Unfortunately the research was ended due to a strategic decision.

But it raised a question: Would it be fruitful to mix these two languages from another motivation? Since mixed approaches of specification and prototyping is recommended, would combining two

languages typically used for each approach make it easier to test a design and understand the design of a prototype?

1.1 Objective

The idea is, that if it is possible to translate between a specification language and a prototyping language, it will be possible to test if the design does what it is intended to, and it would be easier to conceive the design of a prototype.

The objective of the thesis is to explore this idea. Languages for specification and prototyping have been developed for different purposes. Languages for analysis specification are more or less formal and designed to express details at different levels of abstraction. The prototyping languages mainly facilitate rapid development, because prototype development must be cost effective and require quick feedback. This naturally introduces differences in the languages themselves and in the way they are used. The main question of this thesis is formulated like this:

- To what extent is it possible to combine languages for specification and prototyping? And what are the implications of doing so?

The two languages selected here, the Unified Modelling Language (UML) and Ruby have a common characteristic since they are both object-oriented. Because of this they share many language constructs, like the notion of class, object and inheritance. Still there are profound differences, and they come from two very different communities, not to mention paradigms within software development. For instance the UML is statically typed and Ruby dynamically typed. Another example is that using Ruby actions can be specified in detail and UML has limited support for action specifications. Which problems rise from such differences?

The second part of the question is how such a combination will influence the process. Will the mere use of the two different languages force a mixed approach? What will inspire translations between the two languages, and which situations demand the use of one particular language?

Three tasks will be central in answering the main question: 1) Work out a method of translation between the languages, 2) test this translation to see how it works in practice and 3) draw experiences from this experiment. The approach and method used is discussed further in the following section.

1.2 Approach

The first task is to study literature to understand and build the motivation for the idea. The motivation for mixing approaches must be studied in relation to mixing languages for these approaches. It is important to clarify if someone has done something similar before. For instance we can find examples of processes that use the languages chosen, and see if they have been tried mixed.

Since no mapping between UML and Ruby exists, or others have not realized this combination, the first important task is to develop this mapping. A preliminary task here was to identify a subset of the UML that has the most in common with Ruby. This was first and foremost to limit the scope of the thesis to an appropriate level, but also because the UML contains much that is outside the capabilities of any programming language. The subset was chosen to be UML class diagrams (see also 2.4.2 that discusses this further).

By developing the mapping itself, important differences and tensions between the languages will be discovered. The translation model will have to treat every element of each language and make a decision if one element in one language can be represented in the other. If not, can the other language be extended to support the semantics? The experiences from developing the mapping will be valuable in the analysis phase.

When the mapping is realized, we have to test it in practice. A real life example of developing a prototype using the two languages and the translation is a good way to do this. By involving users and other developers in this process, valuable feedback can be gathered in addition to my own experiences.

The experiences from the literature study, developing the translation and combining the two languages in an experiment will hopefully be sufficient to say something about whether the idea is feasible, and answer the question put forth in the previous section.

1.3 Method

Research is to apply a set of techniques and tools to gain insight and knowledge about a problem. Mcgrath formulates it in this way [Mcgrath95 p 152]:

"Doing research" simply means the systematic use of some set of theoretical and empirical tools to try to increase our understanding of some set of phenomena or events.

The method of a research effort is the set of techniques used. I will in short describe the characteristics and weaknesses of the methods used in this thesis. The quality of the result is judged based on the methods applied as the choice of methods greatly influences how the result should be interpreted.

Research methods are classified as quantitative or qualitative [Easterby-Smith91]. In turn they are also seen in relation to the research paradigms or epistemologies of positivism and post-positivism. Positivism sees the world as observable and describable, knowledge about it can be deduced from observation and measurement. Post-positivism argues that the world cannot be objectively observed; there is a reality independent of our reasoning that science can study (critical realism). Positivistic research often applies quantitative methods. Typical for these methods are focusing on objective measurement and then deducing knowledge from these measurements. For example by performing surveys. Qualitative methods have the intention of describing, decoding and give meaning to a phenomenon, thus often used in phenomenological research. A typical example of a qualitative method is interviews. By doing method triangulation, combining two or more qualitative and quantitative methods, we achieve a better result as the weaknesses of one method are tested by strengths of the other.

The approach described in the previous section is qualitative. When a new idea or hypothesis is to be formed and tested the method of grounded theory can be used. Grounded theory is a qualitative approach developed by Glaser and Strauss in the 1960s. The goal is to develop theory about phenomena of interest, this theory has to be grounded or rooted in observation. This is an iterative process starting with generative questions, which are narrowed down to core concepts and finally one through data gathering. By developing the translation model, testing it in practice and noting the experiences throughout the process, ideas will form, be tested and rejected until the result is satisfying or the core concept has to be rejected. Consequently following the practices of grounded theory. Literature study, field experiments and unstructured interviewing are methods used in such a process. The work in this thesis can be seen as a first iteration of a grounded theory process,

where the goal is to arrive at a core concept, verify the feasibility of the concept and make a decision to motivate further work, or to reject it.

The problem with the chosen methods is that they are all qualitative. Optimally triangulation should be used to verify findings using another quantitative method. But such triangulation is often made difficult because of shortage of resources. A way to quantitatively say something about the problem of the thesis would be to perform a study where two groups of people develop a prototype and a design using a mixed approach. One group uses the translation model, UML and Ruby. The other group develops a prototype and a design without mixing the languages in the process. Using metrics on the result, one could say something about how many transitions was made from prototyping to specifying, the quality of the result and the speed of which the design was produced. Such an experiment requires resources in the form of people experienced with mixed approaches, Ruby and UML, and a well-formed translation already developed and explained to the participants. Such resources were unavailable. If the idea of the thesis seems to be feasible, performing such an experiment could be a next step.

1.4 Layout of thesis

The objective of the thesis and the chosen methods are explained in the previous sections. Chapter 2 study approaches in systems development in general, in particular the theory behind mixed approaches. Examples of two processes that use the languages are presented. It elaborates on specification and prototyping languages and discusses the main question of the thesis further. The model of translation between the two languages is presented in chapter 3. The chapter treats UML class diagram and Ruby notation element by element and will also help readers unfamiliar with Ruby syntax. Chapter 4 describes the case study. In chapter 5 the translation model and experiences from the case study are analysed. Chapter 6 and 7 contain my conclusions and suggestions for further work.

2 APPROACHES FOR SYSTEMS DEVELOPMENT

Theoretical background

Compared to other engineering disciplines, software engineering is still a relatively young science. The term "software engineering" was first introduced in the late 1960s at a conference held to find solutions to what was referred to as the "software crisis". Over the last 30 years of development methods of software specification, design and implementation has evolved that leverages our understanding of the activities involved in software development. New notations and tools reduce the effort required to produce complex computer systems. However, many software projects are still delivered over-due and cost more than first estimated, and software that does not meet the customers' needs is produced. This chapter is a literature study and the theoretical background for the thesis, exploring the concepts of complexity and uncertainty in software development, approaches to deal with them and the notion of mixed approaches with emphasis on the spiral model. It will answer why mixed approaches are regarded as a good theory and give examples of two such approaches. Concluding the chapter (2.5), the problem of my thesis is further elaborated, decomposed, and seen in light of the theory explored before, in a discussion about mixed tools for mixed approaches.

2.1 Complexity and uncertainty

The problems facing a software developer or software development team can be classified as elements of complexity and uncertainty [Mathiassen90]. The level of complexity is influenced by the amount of relevant information available for making design decisions. On the other hand the degree of uncertainty represents the availability and reliability of the information that is relevant for the same purpose [Mathiassen95]. The developers are continuously faced with elements of both throughout the process. Further, the degree of complexity and uncertainty in a specific project are not stable factors, but increases and decreases during development [Mathiassen92].

2.1.1 Dealing with complexity

Complexity increases with project size, number of users and stakeholders, the amount of information and the complexity of the information itself. Human beings have a limited ability to handle large amounts of information. When faced with large amounts of information they tend to do abstractions, problem decomposition and information shielding to cope with the situation. Mathiassen and Stage state that when developers use such problem solving techniques they behave in an analytic way [Mathiassen92] [Mathiassen95]. This process of problem solving has been incorporated into many approaches to better deal with increased complexity.

Abstraction is the process of gaining clarity through selection and structuring of relevant information in the problem domain. In [Sommerville95 (p100)] we find a definition of the term abstraction: "An *abstraction* deliberately simplifies and picks out the most salient characteristics". Problem decomposition means splitting a problem into smaller pieces that are more manageable. Information shielding is to organize the information about a problem in different levels of detail. The results of this behaviour are specifications that form the basis for user communication and division of labour between developers. These specifications are expressed in models made in more or less formal languages. Using such techniques is often referred to as specifying, because of the intense use of specifications. This term is also used throughout the discussion.

Stepwise refinement is a kind of "divide and conquer" approach using these techniques, adopted early by system developers. The idea is to do a series of decompositions so that each composition yields a description of the system that is more detailed than the previous version. This is an iterative process that stops once the developers feel they have reduced complexity to an acceptable level. Then the individual decompositions are put together to form the complete specification of the system. When the system has been developed it is checked to see whether it meets the specification. The stage wise model was introduced as early as 1956 and suggested that software should be developed in successive stages. The individual stages include requirements analysis, design, implementation of the design, testing and deployment and maintenance. Such an approach emphasizes planning before acting.

The waterfall model, introduced in the early 70s, is a refinement of the stage wise model. It has two major enhancements over the stage wise approach. First it recognizes the need of feedback loops between the stages of development. Guidelines exist for confining the feedback loops to successive stages to minimize the expense of going back many stages. Errors done in the earlier stages are more expensive to correct. This because it implies going back through all the stages, for which the error has remained uncovered, before it can be corrected. Secondly the waterfall model also has an initial incorporation of prototyping in the life cycle, via a "build it twice" step running in parallel with the earliest stages such as requirements analysis and design. Waterfall-based approaches are called life cycle plans because it includes all stages of a software development process from requirements analysis until operation and maintenance. The waterfall model has become the basis for most software acquisition standards in government and industry [Boehm88]. An extension to the model, incremental development [Mills80] has become a standard for developing large systems. Other extensions, like accommodation of evolutionary changes, formal specification, verification and risk analysis, have been introduced to cope with some of its initial difficulties.

The waterfall model, even with all the extensions and revisions done over the years, suffer from fundamental problems. In Boehm's article about the spiral model of development, he argues that the main problem with the waterfall model is that it is document driven. Fully elaborated requirements analysis and design documents must be completed before going on to the later stages. He claims that for some kinds of software, like compilers or operating systems, this is the best way to proceed[1]. But for many other classes of software, especially bespoke interactive end-user applications, this document-driven approach has led development projects into writing large quantities of unusable code because of inadequate task description on the part of the users, poorly understood user interfaces and decision support functions. The users seem to be involved only in the earliest phases (requirements analysis and design). The consequences and expenses of going back in the waterfall model, make intuitive rethinking of the requirements and redesign of the software very unattractive when the development have reached the later stages. The reality is that requirements are vaguely and ambiguously defined, the requirements change during the development of the system and details left out in the earlier phases turn out to be of major importance. This has lead to many project failures and systems being rejected by the user organization. Many writers have criticized the formal approach put forth by the waterfall model, and argue that software cannot be developed purely in a rational and analytical way.

[1] Erik S Raymond would probably disagree with this point, in his article "The Cathedral and the Bazaar" [Raymond99] he looks at the success of Linux, and how the open source community has in a very evolutionary way developed an operating system with high focus on security.

2.1.2 Dealing with uncertainty

Many of the problems with specification centric approaches are due to lack of effort to reduce the elements of uncertainty of the situation. Remember that elements of uncertainty are introduced because of the availability and reliability of the information as basis for making design decisions. Factors influencing this degree of uncertainty are many. There may be lack of structure that characterizes the users' work. People tend to act according to the situation at hand and not to plans [Suchman87]. This means that workarounds and ad hoc solutions to problems are part of the everyday work. Such processes are not easily formalized. The users may also have less understanding about their own work than they think or have difficulties communicating their work practices to the system analysts. They don't reflect thoroughly about what their work really consist of or where it fits in with the overall work process of the organization. The users may also be very unsure of what they really want: "I don't know what I want, but I'll know when I see it". In larger software projects that take long time to develop, the users' requirements also change during development; the initial requirements are not longer valid when the system is finished. Another factor of uncertainty is the level of experience and training the system developers have. They may be unfamiliar with both the user organization's work practices, and with the technology that is to be used in the development of the system. The effects the system will have on the user organization are also uncertain.

All these situations are best met with experimentation. Mathiassen and Stage say that developers behave in an experimental mode of operation when trying to meet elements of uncertainty. The main technique for experimentation is prototyping. Prototyping is an approach based on an evolutionary view of software development and incorporates the following features [Budde91]:

- Operative versions of one or more parts of the system are produced at an early stage. These are evaluated to learn about the different aspect of the future system or potential solutions.

- Relevant problems are clarified by experimentation.

- Prototypes provide a common basis for discussion between developers, users and other stakeholders.

Budde's classification of prototypes is one of type, goal and what is being prototyped. The following is a short description of this classification:

First looking at different kinds of prototypes: The first is called *prototype proper*. This kind is developed in parallel to the information system model and generally used to clarify a problem or meet uncertainty about the requirements. Another kind is the *breadboard*, which is derived from the specification of the system. It has a main focus on technical issues of one particular solution and often built to learn from. A third kind of prototype is a *pilot system*. This is a prototype that is deployed in the application area, i.e. the user organization. Here there is no clear distinction between the prototype and the application itself. After having reached a certain degree of sophistication, the prototype evolves into a production system through incremental cycles. The users have as much responsibility for specifying the software development objectives as the software developers themselves, and the system increments are geared towards user priorities.

There are different approaches to the use of prototypes, depending on the kind of uncertainty that is to be met. Based on this, Budde referencing Floyd divides the goals of prototyping into three kinds:

1. *Exploratory prototyping*. This is when the developers try to meet uncertainty about the problem at hand and what product the users want. The initial ideas one has about the situation are developed into several prototypes. This way, a number of design options can be examined so that ideas are not prematurely restricted to one specific approach. Budde says that exploratory prototyping is of particular importance in projects where the developers and users belong to different organizations. Exploratory prototyping mainly uses several different prototypes of type prototype proper.

2. *Experimental prototyping*. The main goal of experimental prototyping is to clarify the technical implications of one particular solution before investing resources to develop a complete system using it. The essential aspect is communication between the developers and the users about technical problems and questions related to what is possible. It inspires the developers and users to learn from each other. Breadboards are the preferred kind of prototype here.

3. *Evolutionary prototyping*. Here prototyping is used as a continuous process for adapting an application system into a rapidly changing organization, changes that cannot be specified in advance. One tries to aim for short development cycles and eliminate the differences between the prototype and the application system using a pilot system. This kind of prototyping is the kind enforcing the ideas of evolutionary system development the most. The system developers become technical consultants working close with the users and their organization to improve the application system.

The final classification is that of *vertical* and *horizontal* prototyping. We view the software system as being built up by layers. The top layer is the user interface down to the lowest layers being the database or operating system. When doing horizontal prototyping only one or some layers are built. Traditionally the most frequent layer used in horizontal prototyping is the user interface or the human-computer interface. However a horizontal prototype can also be used to experiment with for instance the business logic layer. Vertical prototyping on the other hand, features all layers of specific parts of the computer system. The motivation for doing vertical prototyping is when the system's functionality and implementation are still open, often the case when building pilot systems.

Prototyping seems to cope better with uncertainty than specifications, however there are some problems with prototyping, which should be discussed. Boehm argues that evolutionary development can sometimes be hard to distinguish from the old code and fix model, whose spaghetti code and maintenance problems was the motivation for the waterfall model [Boehm88]. Further prototyping is fast, using very high level languages and program generation as opposed to application development, which takes much more time. This can make the users get false ideas about completion time of the application. A prototyping approach also raises organizational problems because it is hard to create opportunities for the users to be constructively involved. There are conflicts of interest between groups of users with different demands to the next version of the system. Iterations and involvement of the users make managing the development process harder, and raises questions about when the experimentation should end. Using only prototypes there is a risk overlooking other solutions and knowledge about the users' work are put in the background. Repeated extensions to a prototype (pilot system) may result in a weak and unwise structuring of the program [Andersen86]. Agreements, provisions or established approaches are often geared towards the traditional "milestone documents". This makes contracting and software acquisitions difficult since prototyping and evolutionary development may seem "unstructured" and even "anarchistic".

2.2 Mixed approaches

At this point we might say that specification centric approach, with an analytical mode of operation copes best with complexity. Likewise evolutionary or prototyping approaches with an experimental mode of operation are most effective in meeting uncertainty. But what if both uncertainty and complexity is high? Worse, the degree of each also changes throughout the development process. What is the relationship between complexity and uncertainty? To explore this I take a look at what Mathiassen and Stage calls "The Principle of Limited Reduction".

2.2.1 The Principle of Limited Reduction

In their paper, Mathiassen and Stage give a thorough treatment to the relationship between complexity and uncertainty. Looking at software design in light of human problem solving, they find theoretical support for what seems to be the experience of many practitioners: an effective design effort that is based on combinations of experimenting and analysing. The theory behind this statement stems from Simon's notion of bounded rationality. "The capacity of the human mind for formulating and solving complex problems is very small compared to the size of problems whose solution is required for objectively rational behaviour in the real world – or even for a reasonable approximation to such objective rationality" [Simon57]. The first consequence of the principle is that the actor's intended rationality requires a simplified model of the reality to be developed (dealing with complexity). He can then try to behave rationally in respect to that model, but that behaviour is not rational with respect to the real world, since he is only dealing with a simplified model of such. Uncertainty rises about how good the simplified model reflects the real world. The second is that organizations coordinating and dividing labour among actors become necessary and useful instruments for dealing with complex problems [Mathiassen92].

They argue that contradictory to this realization, complexity and uncertainty have traditionally been regarded as independent. The reality seems to be that complexity and uncertainty are intrinsically related, and there is no evidence that we can hope to reduce one of these without affecting the other. The relationship is stated in The Principle of Limited Reduction [Mathiassen92]:

- Relying on an analytical mode of operation to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures.

- Relying on an experimental mode of operation to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

The implications are that software development requires systematic effort to combine analytical and experimental modes of operation, regardless use of specifications or prototypes. It is advised to adopt an experimental attitude to specifications with walk-troughs, reviews and tests. Also one should adopt an analytical attitude to the use of prototypes. This can be done through using prototypes to clarify desirable features and clarify the relevance and adequacy of a specific design. Also one should emphasize what specifically needs to be learned through the evaluation of a prototype. By adopting such ideas into the use of specifications and prototypes Mathiassen and Stage hope to countermeasure the effects of the principle.

The principle also suggests that a mixture of specifications and prototypes or evolutionary ideas is recommended over a pure specification or prototyping approach. Does such an approach combine the strengths of the two approaches? I go on to discuss mixed approaches and the spiral model in the following section.

2.2.2 The spiral model

A mixed approach is the result of realizing that software development faces elements of both complexity and uncertainty as discussed above. As we have seen, the approaches for dealing with each through specifications and experimentation have their weaknesses. Boehm, Gray and Seewaldt conducted an experiment to reveal the weaknesses of each in 1984 called the UCLA experiments [Boehm84]. They found that specifying scored high on robustness and functionality and lower on ease of use and ease of learning. On the other hand, an evolutionary approach scored higher on ease of use and ease of learning than the specifying groups, but lower on robustness and functionality. Mathiassen and Stage later conducted a similar experiment where they used a mixed approach based on the spiral model [Mathiassen95]. The hypothesis was that such an approach would combine the strengths and make up for the weaknesses of the two approaches. I will discuss what they found, but first I will take a look at the spiral model they used as a framework.

The spiral model is a framework for combining whatever approach is suitable in a given situation. It evolved as a result of the various refinements done to the waterfall model in large government software projects. The approach is risk driven, meaning that the evaluation of risks at several stages of development is the main driver of the model. The name of the model comes from the concept of development in cycles, each beginning with an evaluation of the risks apparent for the next cycle. The number of cycles will vary from project to project.

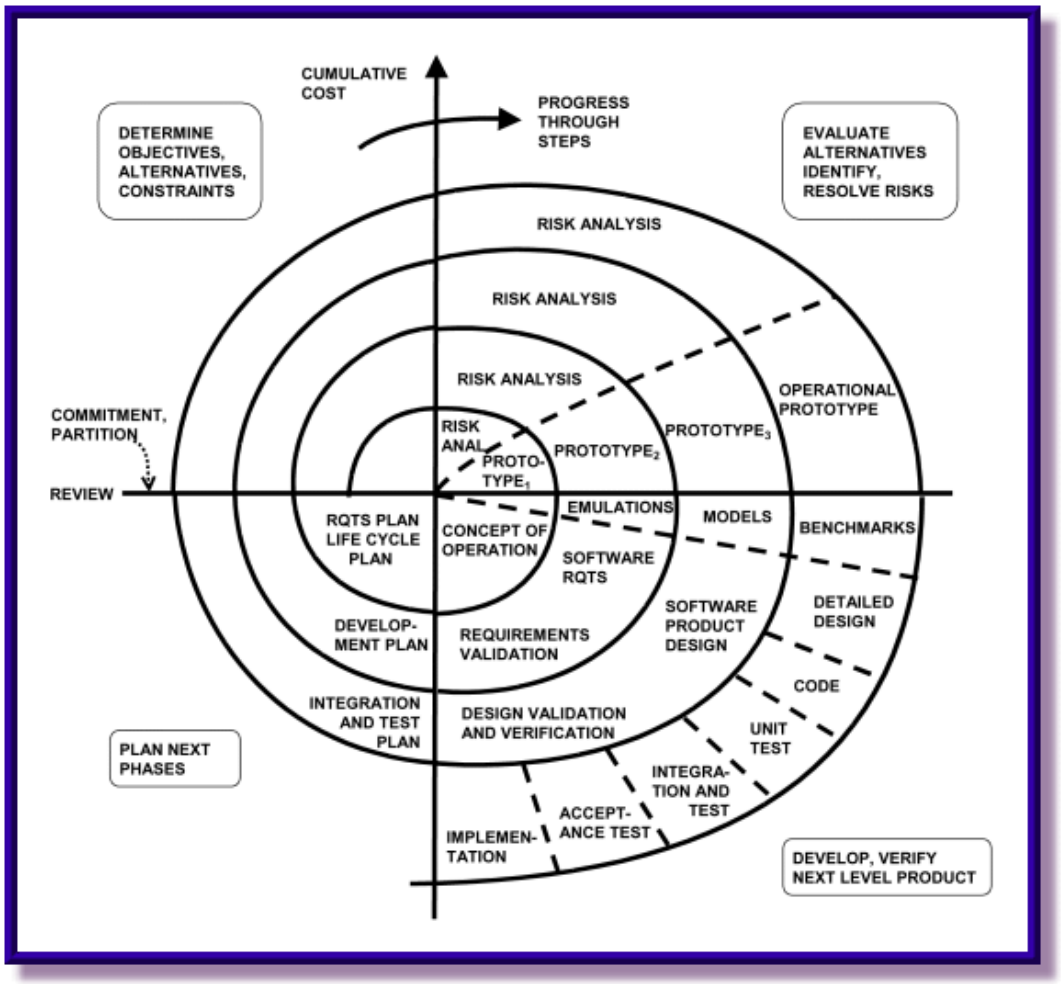


Figure 2.2-1 The Spiral Model

The figure of the spiral model makes it easier to see the concepts, see Figure 2.2-1 [Boehm88].

The stages of one cycle of the spiral always include determination of objectives, alternatives and constraints, evaluation of the alternatives and the risks connected to them, the development and verification of the next level product and a plan for the next phase. The following elaborates a typical cycle of the spiral.

A cycle first begins with identifying the objectives of the part of the product that is the focus of the next cycle. This includes performance, functionality, scalability, maintainability and other relevant issues. Second, the alternative means implementing the product are evaluated. Examples being different designs, re-use of components and purchasing off-shelf products. Third, one identifies the constraints of the different alternatives such as cost and schedule. The next step is to evaluate the different alternatives in terms of risks associated with them. Often this process reveals areas of uncertainty that are sources for project risk. If the risks are great enough the next step will be to find countermeasures to resolve the risks found. This may include prototyping, analytic modelling, benchmarking, user questionnaires or any combination of these or other approaches found suitable. Here the reason for the spiral model being called a mixed approach can be seen. One scenario could be that the user interface risks are greater than the program development risks. Then the next step would be to take on a more evolutionary approach, developing a horizontal prototype of the user interface in close cooperation with the users to find a solution matching their needs. On the other hand if previous prototypes have resolved all the risks of the user interface and the program development risks are dominating, the next step could follow an approach similar to the waterfall model or incremental development of the application. Thus the risk driven approach allows the spiral model to accommodate any appropriate mixture of specification centric, evolutionary or other approach to software development. Particularly risk driven specifications can have different degrees of completeness depending on the risks of performing too much or too little specification.

One of the most important parts of each cycle is that they end with a formal review involving all the stakeholders concerned with the product.

It may be difficult to see when the spiral starts or ends. Boehm says that the spiral starts by a hypothesis that a particular operational mission or set of missions could be improved by a software effort. The spiral itself is a test of this hypothesis. If at any time the hypothesis fails, or the envisioned improvement is installed and having the wanted effects, the spiral ends. Observe that this also includes maintenance of the product, i.e. if one sees need for improvement a new spiral is started.

Regarding uncertainty and complexity the spiral model has the following advantages and features:

- It fosters the development of specifications that are not necessarily uniform, exhaustive or formal, in that they defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until high-risk elements of the design are stabilized. Only complexities that are relevant risks are elaborated.

- It uses prototyping as a risk and uncertainty reduction option at any stage of the development. In the practical example Boehm includes in his article that prototyping and reuse were often used in the process of going from design to code.

- It accommodates reworks and go-backs to earlier stages as more attractive alternatives are identified or as new risk issues need resolution. It also focuses on eliminating errors and unattractive alternatives early.

I would call the spiral model a method generator. With that I mean that each use of the spiral model will differ from other uses, it has to be interpreted and adapted to the specific conditions of a project. But, using the framework it provides, it represents a systematic setting for combining different approaches as suggested by the Principle of Limited Reduction.

The results of the experiments conducted by Mathiassen and Stage were in fact, that using a mixed approach did have advantages over a pure specification or prototyping approach. The mixed approach seemed to have a more even distribution on the four performance criteria of functionality, robustness, ease of use and ease of learning. It also was some evidence that the mixed approach resulted in high score on robustness, avoiding the weakest aspect of prototyping in the UCLA experiments. They also found that the spiral model seems to support early commencement of design and prototype development and leads to a combination of activities using both specifying and prototyping approaches. Comparing with Boehm's experiments, Mathiassen and Stage found that the mixed approach emphasized early design considerations much more than a pure approach of either prototyping or specifying.

The spiral model, although being a good framework for software development regarding both uncertainty and complexity, still suffer from some problems. The students that were involved in the experiments of Mathiassen and Stage concluded that the spiral model is not a simple framework to be followed; rather it is a general framework for understanding and managing software projects, which is open for personal interpretation. Boehm also reflects upon this, saying that the spiral model needs further elaboration of the individual steps. Especially one needs to formulate more detailed definitions of the nature of the spiral model specifications and milestones, the nature of the reviews and techniques for cost estimation, risk assessment and so on according to the situation. There is also need for guidelines and checklists to identify the most likely sources of project risks.

This leads us to another area that is essential when using the spiral model, namely the reliance on risk-assessment expertise. The risk driven approach of the spiral model ensures that high-risk areas are elaborated in great detail at the right time, and leave low-risk elements to be elaborated when they need to. If the developers are inexperienced in risk assessment they may specify the same low-risk elements in detail and overlook or under-specify the elements that really need attention. The effect is an illusion of progress when in reality the project is going in the wrong direction. This means that there is a need for insightful reviewers, experienced in risk assessment, to effectively reveal such situations. Boehm presents the Risk Management Plan that consists of some guidelines for early identification of the main risks of a project [Boehm88]. The idea is that even if a company is not ready to adopt the spiral model as a whole, the Risk Management Plan is an improvement over traditional life cycle plans such as the waterfall model. The spiral model does not overcome the problems with software contracting and acquisitions of the evolutionary approach. Because one does not know how many cycles the spiral model will consist of at project start, the degree of freedom and flexibility needed by the spiral model match poorly to the nature of contracts. The challenge becomes forming contracts that offer enough control of the project when still offering flexibility to adopt the spiral model of development.

2.3 Lightweight and heavyweight processes

In light of the discussions in the previous sections it has been established that the mixed approaches are better to deal with the problems facing a software development project than specification or evolutionary development on their own, and that the spiral model is a good framework to facilitate such a mix. Since the spiral model is not a process or method on its own I will use, as example, two particular approaches both receiving much attention. The reason I have chosen these two particularly is because they use, and are influenced by or influence, the languages used in this thesis. The terms heavyweight and lightweight might both have a negative ring to them, here the terms are merely used as an indication of how much they influence and govern the construction of software and how large software projects they are designed to support.

2.3.1 Rational Unified Process

The Rational Unified Process (RUP) [Booch99] is a life cycle approach, well suited to the UML, which on its own is process-independent. The Rational Software Company that also contributes to the work of defining UML sells the process. It is sometimes called a heavy weight process because it is mainly aimed at large software teams developing complex software solutions, and influences many parts of the organizations' own processes. RUP has the following characteristics [Booch98], and the influence of the spiral model is clear:

- Iterative. The approach is iterative in the sense that increments are used over multiple cycles. The driving force behind the approach is risk evaluation.

- Architecture centric. Focuses on early development and baselining of software architecture.

- Use case driven. The notions of use cases and scenarios are used to align the process flow from requirements capture through testing and as requirements tracing from the final product backwards in the process.

- Object-oriented. The process focuses on object oriented techniques, each model is object oriented.

- Configurable. The process can be tailored to fit various situations, ranging from small software teams to large organizations.

- Risk management and quality control are built into the process.

The process is divided into four phases, each having different weight on 9 different process workflows, as shown in Figure 2.3-1. Each phase is the time span between two milestones, where a set of objectives is met and decisions are made whether to move to the next phase. Within a phase a number of iterations take place, at the end of each an executable project is released. The ideas behind are clearly influenced by the spiral model in Figure 2.2-1.

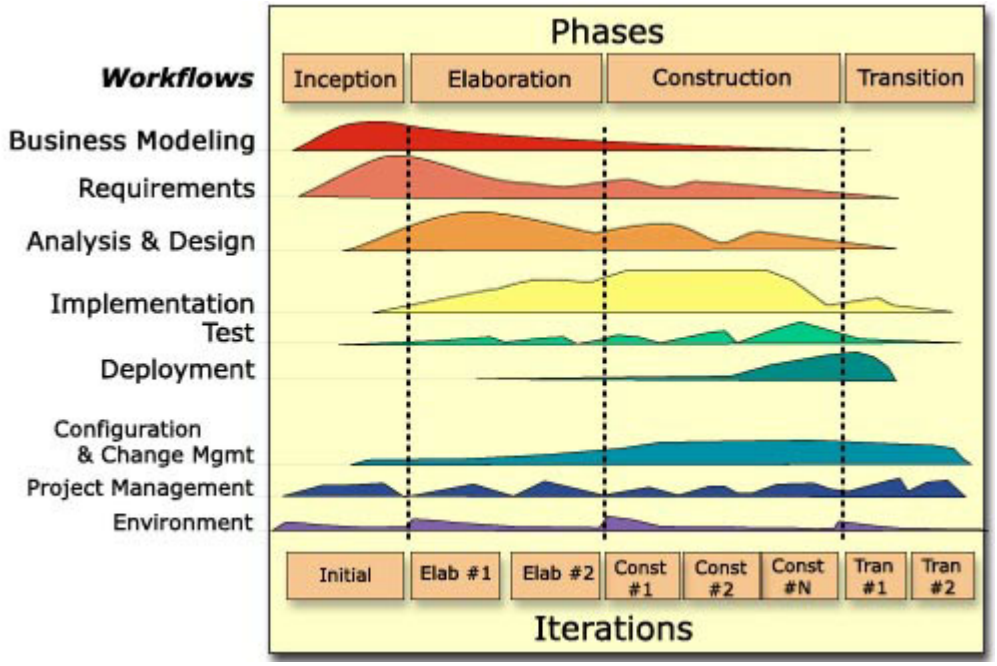


Figure 2.3-1 The Software Development Life Cycle (RUP)

- Inception. The business case is established and delimits the scope project. It is common to develop an executable prototype during this phase.

- Elaboration. The goals of this phase are to analyse the problem domain, establish an architectural foundation, develop the project plan, and eliminate the highest risk elements of the projects. To do this means that the most of the systems' requirements are described. To verify, a system is implemented that exploits the architecture and executes significant use cases.

- Construction. During this phase a complete product is iteratively developed. This implies completing requirements, design the solution and completing implementation and test of the product.

- Transition. The software is deployed in the user community. Issues not accounted for in the previous phases rise, and must be solved. The phase usually begins with the installation of a beta release or pilot system, which is then replaced by the production system.

Going through the four major phases is called a development cycle, and new development cycles will continue to take place until the life of the product stops, hence RUP is a life cycle process.

Note that evolutionary development and prototyping are recommended even in the earliest stages and parallel to analysis and test activities. However, literature about RUP and UML [Booch98][Booch99] and RUP itself fail to give any guidelines on how to apply prototyping in a design effort [Ploesch97]. The reason for this is unclear, it might be that it has been overlooked or not regarded as important, or that there have been attempts to develop such guidelines but it has been too difficult. UML itself is not a language for prototyping but for analysis and specification,

nor does it provide guidelines for how to use the language in its specification. It might be that because RUP emphasizes the use of UML, prototyping guidelines are neglected.

2.3.2 Extreme Programming

During the last years of software development research, a new method called Extreme Programming (XP) has received much attention. The Extreme Programming methodology is the result of the Daimler Chrysler project conducted in 1996 by Kent Beck [Kent99]. Strictly speaking XP is not, nor designed to be, a full life cycle methodology like RUP. Rather it is a collection of disciplined practices that can be formally introduced to a process or used as a supplement to existing processes. As an example of the first see Scrum [Sutherland97], and an example of the last is the RUP plug-in for XP that extends RUP with XP practices. Agile Processes is a term for processes that promote XP practices. XP is designed to be a lightweight process, meaning that it is not a document, or artefact driven approach and it is primarily aimed at small to medium sized projects. The Agile software development manifesto promoting XP and agile processes is a clear indication of the first [AgileManifesto].

Extreme programming practices are a collection of "best practices", many of which has been around for a long time. XP merely provides guidelines to apply them. The main point is the high degree of user collaboration. The stakeholders should be regarded as a co-developer not as a business partner. Communication with the stakeholders is highly valued throughout the whole process, as they provide the requirements through user stories, review prototypes and designs and give valuable input to the developers. Focus on frequent small releases and rapid development is high, maximum time span between two releases is a few months, more often it takes only days or weeks. Robustness is achieved through test-first development and pair programming; -perhaps the most exotic features of XP compared to other methodologies. Test-first development means that unit tests for every piece of software are written before the functionality is implemented. The software must pass every unit test before construction can continue. Pair programming is two programmers working together on one computer, taking turns programming and reviewing the others' code, writing and running unit tests. The process is highly iterative, as it iterates on every small release. The iterations are called "sprints"; each development cycle (requirements, analysis, design, evolution and delivery) is covered by one or a series of sprints. Upon the end of each sprint an executable product is delivered. Naturally prototyping is heavily represented in the process. The influence of the spiral model can also be seen here; each stage in a cycle, a turn in the spiral, and each sprint, a mini spiral within that stage. But the biggest break with other methodologies is that analysis and specification are continuously refined during the lifetime of the product rather than in a separate phase before construction.

There is a misconception that because XP promotes lightweight practices and has a high focus on construction or code production, specification and modelling and documentation is not part of XP [Ambler01a]. Agile Modelling (AM) is a recent addition to the group of agile processes. AM provides guidelines for how to apply modelling techniques to other agile processes like XP or Scrum, and is not a process on its own. The fundamental practices of AM are [Ambler01c]:

- Model with purpose. Do not make models unless you know why.
- Create several small models in parallel, applying the right artefacts for the situation. Do not try to make "all encompassing" models.
- Iterate from one artifact to another to ensure progress.

- Provide code with your models to show that they work.
- Use the simplest possible models, avoid high detail and use the simple tools.
- Use your models to communicate

AM does not use any particular modelling notation or language, rather a range of different diagramming techniques and languages can be applied all depending on the situation [Ambler01b].

2.4 Tools and languages for the different approaches

In the previous sections the characteristics of uncertainty and complexity and their relation in software development have been discussed. Further we have looked at how different approaches have been developed to deal with problems related to these characteristics. The problem of this thesis is how two tools or languages can be combined within a mixed approach. The languages are Ruby, a highly dynamic language supposedly good for rapid development, prototyping and experimentation, and UML, which has become the de facto standard for object oriented modelling. From the previous theory study we see that a mix of approaches is recommended to better deal with the challenges of software development. The question is whether combining two languages traditionally used in different approaches, will be fruitful as well. We have to look briefly at where the two chosen languages come from before elaborating further on the problem.

2.4.1 Object orientation

Because the two languages chosen are both object-oriented, it is interesting to discuss how object orientation (OO) itself deals with complexity and uncertainty. In the two aforementioned approaches (RUP and XP) OO is in many cases a central aspect. First proposed in 1967 with the development of the Simula-67 language, the object-oriented approach has been widely adopted by the industry since the late 1980s.

Object oriented analysis (OOA) and design (OOD) deals with complexity using the discussed techniques of abstraction, problem decomposition and information shielding. It fosters abstraction by the use of objects representing real world entities. A class is a further abstraction over a set of objects identifying common attributes and services. Classes can in turn be organized in inheritance hierarchies, each class higher up in the hierarchy representing a more general abstraction. Problem decomposition is somewhat achieved by putting solutions and functionality in the classes where it belongs (cohesion) separating it from other parts of the solution (decoupling). However problem decomposition is often achieved at higher level through the use of components or packages isolating the set of classes related to the solution of a particular part of the problem. Encapsulation is a feature of OO that is related to information shielding. The details of exactly how a service is carried out are hidden from the clients, who only have to know what services an object offers.

How OO deals with uncertainty is unclear. It might be argued that because an object oriented model is based on entities that the customers know from the problem domain, using such a model to clarify if the domain is correctly understood can be easier. This because the model helped by standards for notation, can be used as a good basis for communication between developers and users. The model express what the developers have understood, the challenge is whether the users understand the model or not.

2.4.2 Specification languages

Tools, languages and notation for producing specifications are many and diverse. What they have in common is that they describe the reality or the envisioned system at different levels of detail, and help the developers to use aforementioned techniques to deal with complexity. The development process in an organization usually has standards for which tools and languages to use. Within OOA and OOD the Unified Modelling Language has become an industry standard.

UML

The UML is a language for visualizing, specifying, constructing and documenting the artifacts of a software intensive system [Booch98]. - A standard language for software blueprints. The current version of UML is 1.4 and the next major revision 2.0 is planned at the end of 2002. It is important to say that UML is only a language offering notation, and just one part of a development process; in itself UML is process independent. However some process authors recommend that it should be used in a process that is use case driven, architecture-centric, iterative and incremental [Booch98]. UP and RUP are examples of such processes.

UML is a unification of previous OOA and OOD notations, and has become a standard modelling notation in the object-oriented design community. The reasons for its widespread use and rapid popularity are [Richter99]:

- *It is based on experience.* UML adopts concepts and ideas of other modelling languages. It also adds concepts and notation missing from those languages.

- *It is an expressive language.* Different views of the system can be described, and UML offers notation for each view. An application can be described at any chosen stage of development, from high-level analysis to low level design. UML also includes well defined extension mechanisms in its metamodel so the notation can be extended.

- *It is a standard adopted by the Object Management Group (OMG), a consortium of companies in the object business.* Therefore many CASE tools support the notation.

To describe the different views of a system, nine different diagram types are used. The views that can be described and the diagrams used are:

- *Functional view*: A functional view describes the functional requirements of a system. *Use case diagrams* are used to express the static functional view. It depicts the system in terms of services offered by the system and their relationship. The static view is extended with *activity diagrams* to specify the dynamic functional view.

- *Static structural view*: This view describes the static structure of the system, using *class and object diagrams*. The class diagram defines structure, meaning what objects and relationships (such as associations and compositions) that may exist in the system at any given time. The object diagram depicts a particular configuration (an example configuration) at a specific time in the execution of the system.

- *Dynamic structural view*: This view describes the behaviour of the system, using interaction diagrams. UML interaction diagrams include *collaboration-* and *sequence diagrams*. They represent the same information, but sequence diagrams include time aspects, and

collaborations focus on the roles in the interaction. They both express the sequences of interactions between objects or components in different situations. The state based behaviour of class instances can be modelled using *state transition diagrams*.

- *Architectural view*: The architectural view depicts the logical and physical structure of the major building blocks of a system. *Component diagrams* show the logical architecture and *deployment diagrams* show the physical hardware architecture. UML interaction diagrams with concurrency can also be introduced at this level to show the process and thread architecture.

UML is a very expressive language that can be used in domains ranging from enterprise information systems to distributed Web-based applications and hard real time embedded systems [Booch98]. UML claims to be a language that is easy to use and understand. Not everyone would agree to this, an argument against this claim is in fact that the language is too expressive.

The choice to use UML to represent the specification language in this thesis, was that it is an industry standard and the most used object oriented modelling language today and the foreseeable future. Looking at the spectre of views where the UML can be applied, it is evident that the UML contains much more than what is useful in this thesis. A subset of the UML that seemed most compatible with Ruby was found to be class diagrams. It is possible that other parts of UML like state machines and sequences also can be mapped to Ruby, but to narrow the scope of the thesis only class diagrams are treated. If we look at other language mappings that have been realized, often with code generation as motivation, we see that they all concentrate on the class diagram. Mappings for state machines have been developed, but state machines are typically used in a limited set of applications and class diagrams more wide spread in use. Sequence diagram extraction and generation exist in some tools, for example Together Control Centre. Still the sequence diagrams have a problem of completeness. In complex programs, such generated diagrams will become very unwieldy and complex because of exceptions and a large number of scenarios. Further code generated from sequence diagrams will only provide call chains in a particular scenario, a small part of the solution when not combined with other actions.

2.4.3 Prototyping languages

For prototyping there exist no standards for languages that can be used, such as UML for object oriented specification. Prototyping languages have one thing in common; they should facilitate rapid construction of the prototype itself, as the cost of constructing a prototype must be minimized and feedback from users should come quickly. Sommerville lists four techniques that facilitate rapid prototype construction [Sommerville95 (p146)]:

- Executable specification languages
- Very high level languages
- Application generators and fourth-generation languages
- Composition of reusable components

An executable specification language is the animation of formal specifications, expressed in formal, mathematical languages like Z [Spivey92]. This is attractive because it combines an unambiguous specification with a prototype, and there is no additional cost in developing the prototype when the

specification is developed. However development is not particularly rapid because formal specification is time consuming, the result is often slow and insufficient, and it only tests functional requirements. An interesting side note here is the work on executable UML (xUML) by OMG consortium member Kennedy Carter and others, having similar goals; precisely define actions and facilitate code generation from UML models.

Very high-level languages (VHLL) facilitate rapid development through powerful data management and much run time support. Examples include Smalltalk and Prolog. It is rare to see these languages used in production systems because of the performance intensive run time support they require. Smalltalk [Goldberg83] is a very powerful prototyping system. It is an object-oriented programming language with a closely integrated development environment. It is extensible through the language itself, so any part can be changed to fit a specific need. Ruby is influenced by ideas of this language.

Fourth generation languages (4GLs) are data processing languages used in the business system domain. Often complete packages for generating applications. An example is Microsoft Access and Excel.

Composition of reusable components has been heavily emphasized in software development recently, not only in prototype production. Production is rapid because many parts of the system are reused rather than designed and implemented. The success of languages such as Smalltalk and LISP is partly because of reusable component libraries as well as their built in language features.

Ruby

Ruby is a fully object-oriented, dynamic scripting language influenced by other languages like LISP, Smalltalk, Perl and CLU. It is getting increasingly popular as a language for system administration, web application development, GUI frameworks, xml-based applications and mathematics among other areas.

Ruby is referred to as a scripting language. Scripting languages is a name for languages that can be used to easily integrate other tool fragments, also called glue. Usually these languages have extremely good text processing capabilities and operation system interfaces. Other languages in this family include Tcl/Tk, Perl and Python, languages that support for rapid application development, which is useful in developing prototypes [Ghezzi98 (p22)]. Scripting languages can sometimes be a negative classification of a language, because people view them as limited to everyday programming tasks and system administration. Because of this many refer to Ruby as a VHLL. Like many other languages within this family, Ruby is interpreted and dynamically typed. There is a general idea that this excels over compiled languages when it comes to prototyping and handling change [Palsberg91].

Like Smalltalk, everything in Ruby is an object. Even the basic types like integer and string. Also the language is extensible, every built in class and module can be redefined or inherited. Highly dynamic and introspective capabilities facilitate metaprogramming, meaning changing the behaviour of the interpreter without changing the interpreter itself. For example methods can be redefined or added to classes and even instances during run-time.

Ruby is open source, distributed under the GNU Public License. This means that its source is available and open for change, and the language is developed like described in [Raymond97].

Ruby is a small language in the sense that it has a small footprint and applications are written in a concise matter, typically comparable to Perl programs in size. On the other hand it is a big language in the sense that it is semantically rich. You can express much in little writing and there are many options for expressing the same thing. Ruby claims to be human-oriented, designed to solve problems in an intuitive way, making it easy to express what you think. It also claims to be easy to read and understand due to simple syntax partly inspired by Eiffel and Ada, and to facilitate rapid development. A part of this thesis is to evaluate these claims, and find out whether Ruby is a good prototyping language. Lately it has become popular in Extreme Programming projects because of the similarities with Smalltalk and good libraries for test-driven development.

The inspiration for using Ruby to represent the prototyping language in this thesis was partly personal fascination for the language and partly because it is dynamically typed, raising interesting questions when it is coupled with a strict statically typed language like UML. For readers not familiar with the language syntax an online version of [Hunt00] is available. Introductions to the language are also available on the Ruby web page [RubyWeb]. In addition, chapter 3 of the thesis will explain the basic elements of the syntax. It has to be mentioned that Ruby could have been replaced by another object-oriented prototyping language like Smalltalk or Python and serve the same purpose in this thesis, Ruby was chosen because of personal experience with this language.

Libraries – Ruby application archive

Libraries and reusable components is a prerequisite for a successful prototyping language. Ruby is younger than other languages often used for prototyping, such as LISP, Smalltalk, Python and Perl. Because of this the libraries and reusable components available to Ruby at the time of writing are not as large as for instance the APIs available to Java or the cspan library for Perl. Even so the Ruby Application Archive has grown as the language's user group increases, and contains free libraries for many different problem domains as well as more general ones. General libraries include xml parsing, networking, database interfaces, GUI development, testing and distributed development. More specialized libraries include support for bio informatics, artificial intelligence and several areas of mathematics.

The biggest problem with the libraries of Ruby has been that they lack documentation. Early in the development of the example prototype a lot of time was spent installing and understanding the libraries. Documentation was thin; in some cases only Japanese versions were available. Another problem was that some of the libraries were unstable, early releases. A design by contract (DBC) [Meyer96][Ploesch97] module I wanted to use in the experiment turned out to be so unstable and performance heavy that it was unusable. This is of course not a permanent problem, both more and better English documentation and library quality will evolve as the language matures and gains a larger user group. A book was published in February 2002 that explains how to use the most common libraries [Feldt02].

2.5 Mixed tools for mixed approaches

The idea of mixing languages in general is not new. In applications you often see that different programming languages are used to make up different parts of the solution. For instance in a web application the dynamic pages may be written using a dynamic language like VB Script, the business logic in C++ and the database interface in C and SQL. In specification and design documents, different modelling languages and notations are used. For instance UML class diagrams to depict structure and architecture, Screen snapshots for user interfaces, ER diagrams for database design and DFD diagrams for data flow. In [Cunningham97] we can read about a successful integration of Java and JPython in evolutionary development. His idea is that there are parts of an application

where the requirements are more likely to change than other parts. Using JPython for these parts of the application makes it easier to apply changes due to the flexibility of this language. When the requirements are more unlikely to change, it is converted to Java in order to increase performance. That way more and more of the application is translated to Java during development. Because JPython is developed in Java, objects of the two languages can send messages to each other and the transition to a more static application is seamless. A pure Java version of Ruby (JRuby) has been developed, and is being used in a similar way. This process of a transition from a dynamic interpreted to a more static compiled environment can also be found in the Self language. From the same motivation several projects have also tried to introduce a type system and a compilation phase for Smalltalk [Graver90].

The motivation to mix Ruby and UML is not the same, although it can be deduced from the above. Ruby is used to experiment and prototype when uncertainty about the design is high. Do we have the right design? And will the design we have now solve the problems of what we are trying to solve? These are questions that drive the prototyping process. At the same time UML will at any time describe the structure of the prototype and the future production system. When uncertainty is reduced to an appropriate level we stop prototyping. It might be that we continue using Ruby, thus following an evolutionary approach, the prototype being a pilot system, or the system may be implemented partly or completely in another language because of performance requirements. Ruby has the possibility to interact with C, and with Java (using JRuby). Anyhow the system will be mainly based on the UML design, and for this reason it is important that this design is Ruby independent.

A motivation to use UML in prototyping is that even the prototype needs to be designed. In an experience report from the use of prototyping in the development of a large-scale industrial software system [Christensen98], the authors emphasize the importance of architectural prototyping. One of the most valuable outputs from their prototype (developed in BETA) was the architecture within and the UML object model describing this architecture. Further they emphasize the importance of the architecture of the prototype itself. This suggests that even the prototype must be described and designed in a more structured way. Prototyping is not just hacking together something that works. Being able to describe the design of the prototype, thereby the probable design of the future system in a language independent notation like UML, helped the developers to discuss architectural decisions with the stakeholders.

The final motivation comes from the notion of mixed approaches, given thorough treatment in this chapter. It might be that being able to easily make a transition from prototyping using Ruby to designing using UML, will make it easier, and inspire to, taking on a mixed approach. From a personal standpoint I have also found motivation in the Agile processes like XP and Agile Modelling, simply stating that you should not do more or less than what is bringing you closer to the target. Design is necessary and prototyping is sometimes necessary, rapid development of the prototype is imperative. Being able to extract the design from the prototype can also help develop the design rapidly, since you are doing things in parallel contrary to first developing a prototype and then develop the design.

The question revisited

The main problem of the thesis was formulated like this:

- To what extent is it possible to combine languages for specification and prototyping? And what are the implications of doing so?

Throughout this chapter the motivation for the idea has been built. The question above is deliberately general, and has to me decomposed into smaller questions to be answered in the analysis. This decomposition is discussed below.

UML is clearly a language for specification by design, but Ruby was not initially developed as a prototyping language. Time has to be spent to clarify if Ruby is a prototyping language, and to defend using it in a mix of specification and prototyping languages.

- Is Ruby a good prototyping language?

To answer this we have to look at what features languages classified as prototyping languages have, and what related features we find in Ruby. The answer is dependent on experiences trying to apply Ruby in a prototyping setting. If Ruby is found to be insufficient as a prototyping language, the hypothesis is falsified.

A part of finding the implications and benefits of mixing languages for prototyping and specification is to answer if UML generation of the prototype helps when discussing and learning about the problem domain. If there are no benefits of such an extraction of structure, the idea is less feasible.

- Did UML generation of the prototype help in user communication?

- What language was used in which situations?

These two questions are important because it tells us something about how each of the languages handles change and how one language is capable of dealing with the weaknesses of the other. The idea is dependent upon the assessment that a language for prototyping will solve problems that cannot be solved by the specification language, and the other way around.

When trying to take on a mixed approach, the effort to mix two languages can in itself inspire to take on such an approach. To explore this, the following two questions have to be answered:

- How do the translations influence the process?

- What inspires the decision to make a translation?

They have to be answered by looking at the experiences from the experiment. A weakness is that the answers are dependent on my own experiences that will, to a certain degree, be subjective. To further say something about how the process is influenced, qualitative methods are more appropriate. As mentioned in 1.3, resources were not available to apply such methods, and suggested as a next step in the grounded theory approach.

Experiences from developing and applying the translation model will focus on the first part of the main problem; to what extent can two languages of prototyping and specification be combined?

- What are the weaknesses and benefits of the translation model?

This is a question that, when answered, will say something about how close UML and Ruby could be brought together. The amount of extensions and special cases will indicate how big the differences between the languages are. There will be things that can be expressed in one language but not the other. This brings us to another important question:

- How should the languages be combined?

At this point this is not clear. One possible solution is to form a hybrid language that brings the best of both languages, for example a language in which structure is defined in UML and actions in Ruby. This would mean that this hybrid language is used for both experimentation and prototyping. Another solution is to keep the languages apart, use them together in the process, but one for prototyping only and the other for specification only.

3 TRANSLATION BETWEEN UML AND RUBY

A model of translation

To effectively combine Ruby and UML in a design effort, a model of how to translate between them is needed. It is obvious that several parts of UML such as use cases, sequence diagrams and activity diagrams don't have any direct representation in Ruby. UML class diagrams however have more in common with Ruby elements. The model will therefore describe how elements of Ruby are represented in UML class diagram notation and vice versa.

3.1 Perspectives

You can have three different perspectives when you draw UML class diagrams [Cook94]. The perspective you have when you make a class diagram affects how you should interpret them. The perspective influences what the class diagram describes. The three perspectives are described here:

- **Conceptual**. The conceptual perspective treats the UML class diagram as a description of the concepts in the domain of the software system. Such a diagram should be drawn without regard to the software that later will implement it (be software independent). Cook and Daniels call this the essential perspective [Cook94]. It is usual to have a conceptual perspective during the analysis phase.

- **Specification.** The specification perspective looks at software but not at implementation. It deals with designing the interfaces of the software, or the types of the system. The notion of a class in an OO language supports both interface and implementation [Fowler00], and thus these two concepts are often mixed together. Because a specification model goes through an evolution into an implementation model it can be difficult to say when you have a specification perspective and when you have an implementation perspective.

- **Implementation.** In the implementation perspective the focus is on software classes and their implementation. It can be argued that because the key to effective OO programming is to program to a class's interface and not its implementation the specification view is the better one to take. For a thorough discussion see the first chapter of [Gamma95].

So except for the conceptual perspective the lines between the two other perspectives are not clear, but important to have in mind when drawing class diagrams. I bring this up here because given the definition above it is the specification perspective I want to extract from a Ruby prototype into a class diagram, in other words the interfaces of the classes and the relationships between them. The risk is that when translating from a Ruby prototype you are effectively in implementation perspective and it is easy to end up with a class diagram that describes the implementation, not the interfaces of the system. On the other hand, when going from a UML class diagram to Ruby, it might be the implementation perspective of a specific part of the system you have described and want to translate. Because OO languages such as Ruby and UML support both interfaces and the implementation of them this duality can be met by saying that specification and implementation is two aspects of the same thing. In other words treat this as a feature of OO; that we have a mechanism that makes the leap between specification and implementation smaller, rather than a problem.

To separate between specification and implementation perspective in Ruby, I introduce a convention of how to organize Ruby code using modules and open class definitions that makes it easier to switch between perspectives. This is explained in 3.4.

3.2 Stepwise translation model

Throughout the description of the model I will use an example from the Ruby book [Hunt2000] and translate the elements step by step into a class diagram. This way each element is treated by itself while keeping focus on the whole picture. The system described is part of a jukebox program and will be extended as we go through the different sections to come.

3.2.1 Classes

Classes are important building blocks of both UML class diagrams and Ruby programs. As the name suggests, a class is a classification of objects that share the same attributes, operations, relationships and semantics. Further a class tells us the responsibilities its instances have within the software system.

```
class Song
  @@totalplays    #Class attribute

  def initialize(name, artist, duration) #Constructor
    @name = name
    @artist = artist
    @duration = duration
    @plays = 0
  end

  attr_reader :name, :artist              #Attribute reader
  attr_accessor :duration                 #Attribute accessor

  def durationInMinutes                   #Method
    @duration/60.0
  end

  def durationInMinutes=(value)
    @duration = (value * 60).to_i     #Uniform access principle
  end

  def play
    @plays += 1
    @@totalplays +=1
    name + ",” + duration +", This song: #@plays plays. Total #@@totalplays
plays”
  end
end
```

Listing 3.2-1 Jukebox song class

The Ruby code in Listing 3.2-1 is a declaration of a class called song. The keyword *class* and the name of the class mark the beginning of a class definition, anything between this and the *end* keyword marks the class body. This song has a class attribute called totalplays, which is the number of times this jukebox has played any song. Class scope is indicated by the two leading @ signs. Further the class has four instance variables: name, artist, duration (in seconds) and plays which indicate how many times this particular song has been played. An instance variable in Ruby has one leading @ sign. The instance variables are private by default in Ruby, one way of making them

readable from the outside is by declaring reader, writer or accessor (read and write) methods for them. Above name and artist are read only, and duration are readable and writable. A method declaration in Ruby is marked by the keyword *def*, and the body of the method is between this and the *end* keyword. The result of the last statement in a method body is the return value. The method durationInMinutes serves as a "virtual attribute", it seems like a normal attribute to the rest of the world, but is really calculated based on the duration instance variable. This is also called the Uniform Access Principle [Meyer97]. Note that the term "virtual attribute" bears no resemblance to a virtual function in OO terminology, such as the operations of an abstract Java class.

When this class is translated into the corresponding UML class, there are some things that are straightforward and other things that are more complicated. In the following sections each element of a class will be treated separately.

Attributes

The notation for an attribute in UML is: *visibility name: type = default value*. Class attributes like totalplays in the Song class are underlined. As Ruby is a dynamically typed language the type of an attribute cannot be directly read. We have two choices here; we can choose to not specify the type in the UML class diagram either since this is optional, or we can get the types of the attributes by looking at the context in which they are used. A third way is to use the reflectional possibilities in Ruby[2]. Reflection can only be used at runtime and thus this method to extract type information can only be done during execution. Here I will use the second method and simply say that name and artist are strings and duration plays and totalplays is of type integer. This is because the two first are used within string concatenation and the next three are assigned integers. In an implementation perspective class diagram we could include getter and setter methods for access to the attributes and make the attributes themselves private (as in the example). The interesting bits are that a Song instance can tell us its name, duration and artist, and that duration can be set from the outside, so it is enough to make them public attributes. Further we know that a Song keeps track of how many times it has been played as well as the total amount of played songs, so we include these as private attributes.

Translating the attributes specified in a UML class to the corresponding Ruby class is straightforward. The notation for attributes in UML was: *visibility name: type = default value*. Beginning from the left with visibility: Private attributes correspond to a Ruby instance variable - *@name* because all instance variables are private by default. Public attributes translate to an instance variable and a public getter and setter method, for readability the use of the shorthand method *attr* and its friends can be used.

- *attr_accessor :name* generates get and set methods for instance variable *@name*. It takes multiple arguments so several instance variables can be made public on the same line.

- *attr_reader :name* generates a get method for *@name*, it does not correspond to a public attribute because it can not be set. As with *attr_accessor* it takes multiple arguments

- *attr :name* generates a get method. It also takes a Boolean value to specify if a setter method also should be generated, thus *attr :name, true* equals *attr_accessor :name*.

[2] See reflection paragraph

Protected attributes can be declared by first making an accessor and then declare the get and set methods protected like:

```
attr_accessor :name;
protected :name, :name=              #name= -> the setter method
```

I will use the assumption that when visibility is left out in UML it is public. If in Ruby, an attribute has an attr_reader defined it is only readable and result in a private attribute in UML as well as an operation with the same name that gives us the value of it.

Type information is lost when translating to Ruby, because of its strong but dynamic and inexplicit typing. A convention is to write a comment for each attribute stating the expected type. If the attribute has a specified default value it must be set in the initialize method of the class.

Summary – translating attributes

From Ruby to UML:

- Include attributes that in Ruby are accessible from the outside in some way as public attributes in the UML class.
- Include private attributes and class attributes that are important parts of the state of the objects of the class.
- Read visibility from the use of attribute accessor methods. *attr_accessor* and *attr ‹name», true* result in a public attribute and *attr_reader* a private attribute and a get operation with the same name. Protected attributes are a combination of *attr* and friends and the *protected* keyword.
- Make operations of virtual attributes that are important.
- Deduce type information from the context the attributes are used in, or use reflection to get it during execution. Optionally it can be left out as UML allow unspecified type.

From UML to Ruby:

- The attribute becomes an instance variable with the same name
- Visibility is specified by declaring get and set methods, using the shorthand notation adds readability
- No specified visibility means public visibility
- Type information is lost; optionally use a comment to specify it.
- Default values must be set in the initialize method.

Operations

The UML syntax for an operation is: *visibility name (parameter list) : return-type-expression* {*property-string*}. The default visibility of a Ruby method is public, except for the initialize method, which is always private. Visibility in Ruby is declared by one of the keywords *public, private* or *protected*. You don't have to declare visibility for every method separately. A Ruby shorthand is that all methods following one of the keywords are assumed to be of that visibility. The initialize method is a kind of constructor of the class (it is called when an object is created using the new keyword) and thus the first method to be called. The next version of UML (2.0) will include object construction notation. If included, the initialize method is stereotyped «constructor».

Stereotype «constructor»:
Called upon object construction, the first method to be called

The parameters of an operation in UML are similar to the syntax of an attribute, but with the extra element of direction; *direction name : type = default value*. The direction can be in, out or in-out (both). Ruby does not support out parameters, if several values are to be returned they must be wrapped in a holder object. Parameters in Ruby are passed by value, but there are ways of getting the effect of by-reference parameter passing and thus Ruby also supports in-out parameters. Ruby also gives the possibility of variable length argument lists, not directly supported by UML. In these cases the argument list has to be wrapped in a container type like Set or Sequence in UML. This is also how Ruby internally handles variable length argument lists. Another special feature of Ruby is that a block of code can be associated with a method call and yielded for execution inside the method body, but this is a feature of the method call not the signature of the operation. Set methods follow the convention of *methodname=(argument)* so these get a leading operation name of get in UML. Later we will se another use of special syntax for a method[3]. We should give more understandable names for these names in UML, although *name* in UML is just a series of characters. Type information about return types and parameters has to be deduced in the same way as for attributes, either by looking at their use, or through the use of reflection at runtime. A Ruby method returns the result of the last statement in the method, or an explicit return statement.

The signature of an operation in UML was: *visibility name (parameter list) : return-type-expression* {*property-string*}. Visibility is translated to Ruby by using the leading keywords *private, public* or *protected* together with the name of an already defined method.

```
private :methodname1, methodname2, ...
```

As a convention all methods following a visibility keyword have the specified visibility. It is normal to list all private, protected and public methods in groups.

The types of parameters and return type are lost; the programmer must uphold this by passing expected objects and returning objects correct type. As for attributes, a comment above the method definition can be included using this syntax:

```
# returns:type, param1:type, param2:type...
```

The property string is simply a named value denoting the characteristics of the operation, such as {query}. This should also be a comment in Ruby.

[3] [ ] method of class SongList

Summary – translating operations

From Ruby to UML:

- All methods are public unless specifically stated otherwise.
- Initialize is stereotyped «constructor», this means that it is called first.
- Type of the parameters and the return type of the method are deduced from the context of use, or through reflection during execution. The value returned is either the result of the last statement or an explicit return.
- Methods that have several possible return types based on execution flow, or where the parameters can have different types, must be split into separate methods for each return type.
- Variable length argument lists must be wrapped in a set or sequence.
- Methods with special Ruby syntax, like assignment methods, should be given more general names.

From UML to Ruby

- Visibility is specified for each defined method or by grouping together methods after the corresponding visibility keyword.
- Type information is lost, must be upheld by the programmer. Optionally use comments.
- Property strings become comments.

3.2.2 Relationships

Relationships in the UML are the ways that things can connect to one another, logically or physically. There are several kinds of relationships, the most important being: associations, generalizations and dependencies.

Associations

In UML associations represent structural relationships between classes, implying that objects of one class are linked to objects of another. Their meaning is different based on the perspective, mentioned earlier. Take the following example:

Figure 3.2-1 Example association

In a conceptual perspective, an association represents a conceptual relation between classes. For example an association may tell us that an Order has to come from a single Customer, and that one Customer may have several orders. In the specification perspective, the associations represent responsibilities. The same association will now imply that there are one or more methods in a Customer instance that can tell us what Orders that specific Customer has made, and also one or more methods in an Order instance that tell us who placed the order. An implementation perspective would maybe say that there exists a set of references from one Customer to Orders, and a reference from an Order to the Customer that placed it.

Continuing the jukebox example, we see that we need a way to get hold the different songs available. We introduce a SongList class that holds a reference to all the songs:

```
class SongList
  attr_reader :songs
  def initialize
    @songs = Array.new
  end

  def append(aSong)
    @songs.push(aSong)
    self
  end

  def deleteFirst
    @songs.shift
  end

  def deleteLast
    @songs.pop
  end

  def [](key)                # Method for finding a song
    if key.kind_of?(Integer)
      result =  @songs[key]
    else
      result = @songs.find { |aSong| key == aSong.name }
    end
    return result
  end
end
```

Listing 3.2-2 Jukebox SongList class

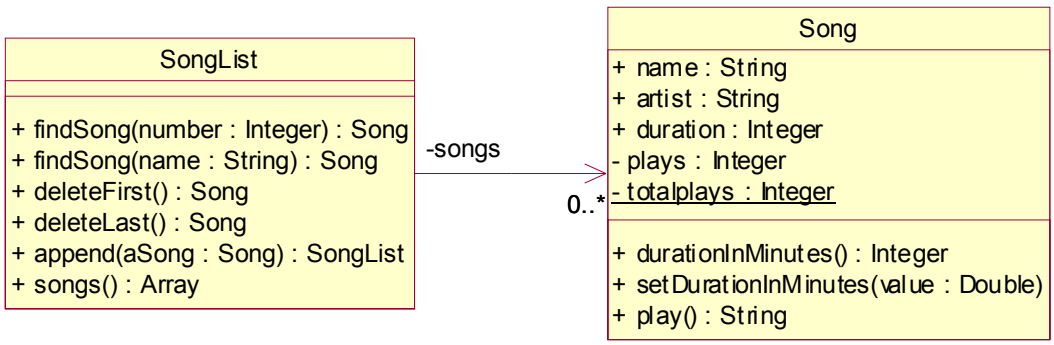


Figure 3.2-2 Jukebox association

The SongList has a private array holding references to Song objects. Songs can be added and deleted. [ ] is a method that finds a song for us, based on either an array index or a song name. Here we have an example of the use of blocks in method calls, the find method iterates over all the objects of an array and executing the block for each object. This is an example one of Ruby's many built in collection methods.

As I mentioned I want to extract the specification perspective when I go from Ruby to UML. In the example above I use the fact that a SongList has a method to look up Songs as an indication of the existence of an association between the SongList and the Song. Further I see that there is a collection of Song references in the attribute *@songs*, indicating a multiplicity of zero to many songs. Finally I know that there are no methods in Song accessing a SongList and thus the navigability is unidirectional in direction Song. We know nothing about how many SongList objects know about a song, so the multiplicity in the direction of SongList from Song can not be read from the Ruby code. Since the navigability is in direction Song, this is not necessary to know either except from a conceptual standpoint.

In both UML and Ruby we can have relationships cycling back to the class itself. A person may for example have a significant other who is also a person.

In UML an association can connect more than two classes. These are called n-ary associations. Such associations does not map directly to the concept of references in Ruby, since a reference cannot point to several different objects. In UML associations are viewed as instances of the UML metaclass Association that has an arbitrary number of association ends where you find objects it links together. We can implement an n-ary association in Ruby the same way. In the section about the association and composition extension (5.4.2) problems regarding associations are discussed further.

Summary – translating associations

From Ruby to UML:

- A method giving information about another class is an indication of an association with navigability in that direction.
- If an attribute references another class, this class knows about the other class. Navigability is deduced by looking at the direction of the reference.
- Multiplicity is deduced by the existence of only one or a collection of references.
- Some multiplicity information cannot be read from Ruby code. This coincides with the association being unidirectional.
- The role name of an association can be the name of the instance variable that holds the reference, or left out. Role names are typically introduced to make the model easier to understand.

From UML to Ruby:

- An association with fixed multiplicity results in one or more instance variables referencing objects of the class at the opposite association end. Fixed multiplicity implies assignment to these references upon object creation.

- An association with non-fixed multiplicity result in an instance variable referencing a container like Array, Hash or custom made object in turn referencing objects of the class at the opposite association end.

- The name of the instance variable holding the reference should be the same as the role name, or the name of the opposite class. Use plural form if the multiplicity is non-fixed.

- Navigability is assured by keeping the reference in the object(s) on the right end(s)

- N-ary associations (for example a ternary relationship) are constructed in this way:

1. Introduce a class that represents an association

2. Create references to the classes that the association ends point to

3. Make the association class navigable from the classes that the association ends point to if needed.

Aggregation

The UML defines a special kind of association called aggregation. An aggregation specifies a "whole/part" relationship where one class represents a larger thing (whole), which in turn consists of smaller things (parts). The aggregation is a "has-a" relationship between the whole and the part. The meaning of an aggregation is entirely conceptual and has no formal semantics other than the ones of an association. It is therefore difficult to give a Ruby equivalent of an aggregation. Specifically an aggregation does not change the meaning of navigation from the whole to its parts or the lifetime of the whole and its parts. In our jukebox example we might model the fact that the SongList has several Songs in it using an aggregation instead of an association in the UML model.

Composition

A composition is a variation of the simple aggregation that extends the semantics beyond that of an association. [Booch98] defines composition like this:

Composition: A form of aggregation with strong ownership and coincident lifetime of the parts by the whole; parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it; such parts can also be explicitly removed before the death of the composite.

Further a composite is a class that is related to one or more classes with a composition relationship. A composition relationship is expressed as an association with a filled diamond at the association end connected to the whole.

Having a composite allows us to treat a group of interconnected objects as one object because the parts and their relationships together specify the internal structure of the whole. A more exact definition of a composition can be found in [Bock96]. I summarize their definition here:

Composition is defined as a relationship that associates a whole and its parts, in which:

1. Each part is of a certain type
2. Parts may be connected in certain ways unique to the composite
3. Each part has an identifiable role in the composite
4. Parts may be assigned properties unique to the composite
5. Each kind of part may have more than one instance in the composite
6. Relationships may have parts or be parts themselves

As all of these characteristics of a composite can make it rather complex, expressing it in UML, let alone a programming language like Ruby, can be difficult. It is especially hard to spot composites when reading pure programming language code. Therefore I have developed a Ruby module and runtime obeying some rules of compositions that eases this task. The code for this module can be found in on the accompanying CD-ROM.

I have found it necessary to introduce two different types of composition relationships, with slightly different semantics. These are generalizations of the six kinds of compositions defined in [Odell94].

- Component-integral object composition – "Scenes are part of films"
- Material-object composition – "A cappuccino is partly milk"
- Portion-object composition – "A meter is a part of a kilometre"
- Place-area composition – "A peak is part of a mountain"
- Member-bunch composition – "A tree is part of a forest"
- Member-partnership composition – "Steven and Mary are a married couple"

The first is the default kind of composition, where the parts in the composition can not be exchanged for other parts or used by other composites. The parts are assumed created together with the whole and die with the whole. This kind of composition will often be used when we have fixed or at-least-one multiplicity. This variation covers all Material-object, Portion-object, Place-area and Member-partnership forms of compositions as well as those Component-integral object composites which parts are not interchangeable. The second variation ensures that all parts are always of a specific type, but can be exchanged with other parts. When used by one composition a part can be participating in another composition. In these cases the use of a composition should be reconsidered, [Odell 94] discusses relationships that are often mistaken for compositions. Typically

this will cover Component-integral object composites where parts can be changed and Member-bunch composites where parts also can exist in their own rights (however in these cases a qua-type will often be introduced). In UML the first type is specified using a simple filled diamond, the second type is stereotyped «weak».

Stereotype «weak»:
A non-invariant composition, the parts can be exchanged for other parts. If non-fixed multiplicity, parts can be added after construction of the whole.

The extension of Ruby I have developed covers both types of composites. To use the extension one must mix in the Composition module in each class that have relationships with parts in a composite, variable that are references to parts are marked. A default composition is exemplified in Figure 3.2-3, corresponding Ruby code in Listing 3.2-3.

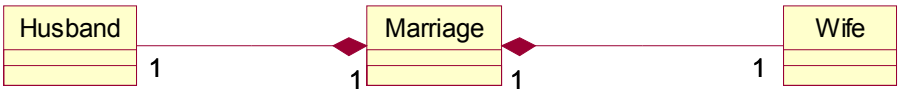


Figure 3.2-3 Default composition UML

```
require 'composition'

class Husband
end

class Wife
end

class Marriage; include Composition
  def initialize(husband,wife)
    @husband = husband
    @wife = wife
    comp :husband, :wife
  end
end
```

Listing 3.2-3 Default composition Ruby

Example of a weak composition follows in Figure 3.2-4 and Listing 3.2-4. The difference about the parts of a marriage and the parts of a bike is that the marriage is invariant in respect to the parts. When a part leave the marriage is over. The wheels of a bike can be changed with other wheels, but the bike is still a bike.

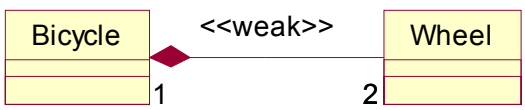


Figure 3.2-4 Weak composition UML

```
require 'composition'

class Wheel
end

class Bicycle; include Composition
  def initialize
    @frontwheel = Wheel.new
    @rearwheel = Wheel.new
    weakcomp :frontwheel, :rearwheel
  end
end
```

Listing 3.2-4 Weak composition Ruby

The Composition module consists of a per-object register of parts, private methods for checking if the parts have changed and a Runtime class. The Runtime class keeps a global register of what parts that are used by what wholes and is responsible for defining methods to check for violations on a per class basis.

The main trick is to redefine the "method_added" call back method whenever a new class is defined. This allows the Runtime class to wrap every public method being added to the class, if it is a class that uses the Composition module. It does this by aliasing the new method, redefine it to first call the original method with the parameters given, storing the result, call the methods that check for composition violations and finally return the result of the original method.

Whenever a new part is added to the composition, the Runtime checks if the part has been used by any other composites, raising an exception if this is true. It then stores the part-whole mapping in the global part register. Further it stores the object id of the part in the per-object register mapping the part (reference) name to it. If it is a weak composition only a mapping of the type to the name is stored. Further if this is the first time a part is added, a finalizer method is defined for the object ensuring that the register is cleaned up when the object is garbage collected.

The checks are performed when a public method of the composite is called. For a default composition an exception is raised if any of the parts has changed, for a weak composition if any of the parts have changed type.

It has to be mentioned that this extension does not cover all restrictions of a composite. Particularly since there is no way to explicitly destroy objects in Ruby, a whole can not destroy its parts when it is deleted. The user must be sure that nothing references any of the parts so the garbage collector destroys them. Of course it is not good programming to reveal parts of the composite to the outside, but the Composite runtime does not take care of this. The second issue has to do with associations. If the parts are stored in a collection object of some kind for a one-to-many composite relationship, each part is not registered on the per-object level, only the id / class of the collection object. The collection object also has to include the Composition module and keep a register of the parts, meaning that the constraint perhaps does not reside in the correct place. An idea to solve this problem is outlined in chapter 91. However the composition extension still adds semantics and readability to the composite concept in Ruby. Later in the development of the translation explicit typing was introduced to associations and compositions using the same extension as a basis. See 5.4.2 for how this was done.

Qua classes in Composition

Expressing a composite does not only include specifying that objects can consist of other objects, but also their internal structure. For example a car consists of engine and wheels, and the car has an internal structure defining how the engine is connected to the wheels and what wheels the engine powers.

A qua class is a subtype created solely because of a special relationship to another object. In a composition it is often necessary to introduce a qua-type to use the same kind of object in more than one situation. Using an example from [Bock94], an engine can be used by a car to power the wheels but also by a submarine to power the propeller. The Engine can then be qua typed into CarEngine and SubMarineEngine to more clearly express how they are used. A CarEngine powers the wheels and a SubMarineEngine powers the propeller. An engine that powers both wheels and propellers are avoided. Thus the parts are sub typed in context of the whole.

Since parts can consist of other parts and so on, UML diagrams grow more complex with each qua type added. It is suggested for UML 2.0 to model composite parts and their internal relations inside the class of the whole. In Ruby we can subtype another class from within the class itself, we can also add features to one object's class and create a new anonymous class. In the following example I have used the first method on all except the wheels to show both possibilities. Notice that the Ruby code not only specifies the different parts and their internal structure, but also how the parts are constructed by the whole. A feature not expressed by the UML diagram in Figure 3.2-5. The corresponding Ruby code is shown in Listing 3.2-5.

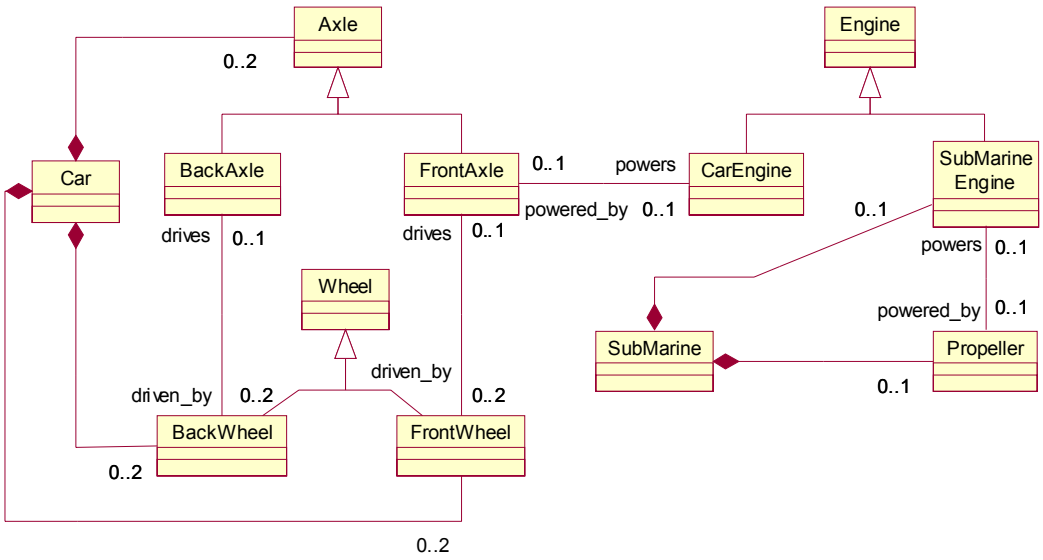


Figure 3.2-5 Composition using qua types

```
require 'composition'
class Axle
end
class Engine
end
class Wheel
end

class Propeller
end

class Car; include Composition
  class FrontAxle < Axle
    def initialize(left=nil, right=nil, powered_by=nil)
      @leftwheel = left
      @rightwheel = right
      @powered_by = powered_by
    end
    attr_accessor :powered_by
  end

  class BackAxle < Axle
    def initialize(left=nil, right=nil)
      @leftwheel = left
      @rightwheel = right
    end
  end

  class CarEngine < Engine
    def initialize(powers=nil)
      @powers = powers
    end
    attr_accessor :powers
  end

  def initialize
    # The car should construct the parts
    @lf_wheel = Wheel.new
    @lb_wheel = Wheel.new
    @rf_wheel = Wheel.new
    @rb_wheel = Wheel.new
    @b_axle = BackAxle.new(@lb_wheel,@rb_wheel)
    @f_axle = FrontAxle.new(@lf_wheel,@rf_wheel)
    @engine = CarEngine.new(@f_axle)
    @f_axle.powered_by = @engine

    # Extend the wheels in context of car
    class << @lf_wheel
      attr_accessor :driven_by
    end
    class << @rf_wheel
      attr_accessor :driven_by
    end
    class << @lb_wheel
      attr_accessor :driven_by
    end
    class << @rb_wheel
      attr_accessor :driven_by
    end
    @lf_wheel.driven_by = @f_axle
```

```
    @rf_wheel.driven_by = @f_axle
    @rb_wheel.driven_by = @b_axle
    @lb_wheel.driven_by = @b_axle

    # Define the parts as compositions
    comp :lf_wheel, :rf_wheel, :rb_wheel, :lb_wheel
    comp :engine, :b_axle, :f_axle
  end
end

class SubMarine; include Composition
  class SubMarineEngine < Engine
    def initialize(propeller=nil)
      @powers = propeller
    end
  end

  def initialize
    @propeller = Propeller.new
    @engine = SubMarineEngine.new(@propeller)
  end

end
```

Listing 3.2-5 Qua types in Ruby

Shifting focus back to the Jukebox example we see that the jukebox itself is missing. We also miss start, pause and similar buttons. Having gone through how we represent composition in UML and Ruby, one way of representing the Jukebox whole is listed in Listing 3.2-6:

```
require 'UML/composition'
require 'thread'

# assume the jukebox hardware vendor gives us a basic button
# interface to use for
# callbacks from the jukebox's different buttons
class Button

  def initialize(label)
    @label = label
  end

# callback for button pressed
  def buttonPressed
    # do start action...
  end
end

class JukeBox; include Composition
  class JukeBoxButton < Button
    def initialize(label,&action)
      super(label)
      @action = action
    end

# invoke the block stored in @action blocks are closures,
# so it will be called in scope of the jukebox where it was defined
# not the button. See [Hunt2000]
```

```
    def buttonPressed
      @action.call(self)
    end
  end

  class ProgramList < SongList          # a programmable list to be played
    @attr_accessor :current
    def initialize
      super
      @current = 0
      @playing = false
    end
    def start
      @playing = true
      @playthread = Thread.new do
        while @playing = true and current < @songs.size
          [current].play
          current = current + 1
        end
    end
    def stop
      @playthread.stop if playing
      @playing = false
      @current = 0
    end
    def pause
      @playing = false
    end
  end

  attr_reader :program, :songlist    # make program- and songlist readable
  def initialize(songlist=SongList.new)
    @songlist = songlist
    @program = ProgramList.new       # an array holding songs to be played

    @startButton = JukeBoxButton.new("Start") { program.start }
    @pauseButton = JukeBoxBotton.new("Pause") { program.pause }
    @stopButton  = JukeBoxButton.new("Stop") { program.stop }
    comp :program, :songlist, :startbuttong, :pausebutton, :stopbutton
  end

  def listSongs
    songList.songs.each {|song| print song.artist + ":" + song.name}
  end
end
```

Listing 3.2-6 Jukebox composition

The Jukebox itself has three buttons: play, pause and stop. These buttons are all JukeBoxButtons but have different actions when they are invoked (see generalization/specialization and dependency sections for translation details). It also contains a ProgramList, which is the current list of songs to be played, and the SongList, which holds the available songs in the jukebox. Figure 3.2-6 is the corresponding UML class diagram of the jukebox composite, using the translation rules for attributes, operations, associations and inheritance.

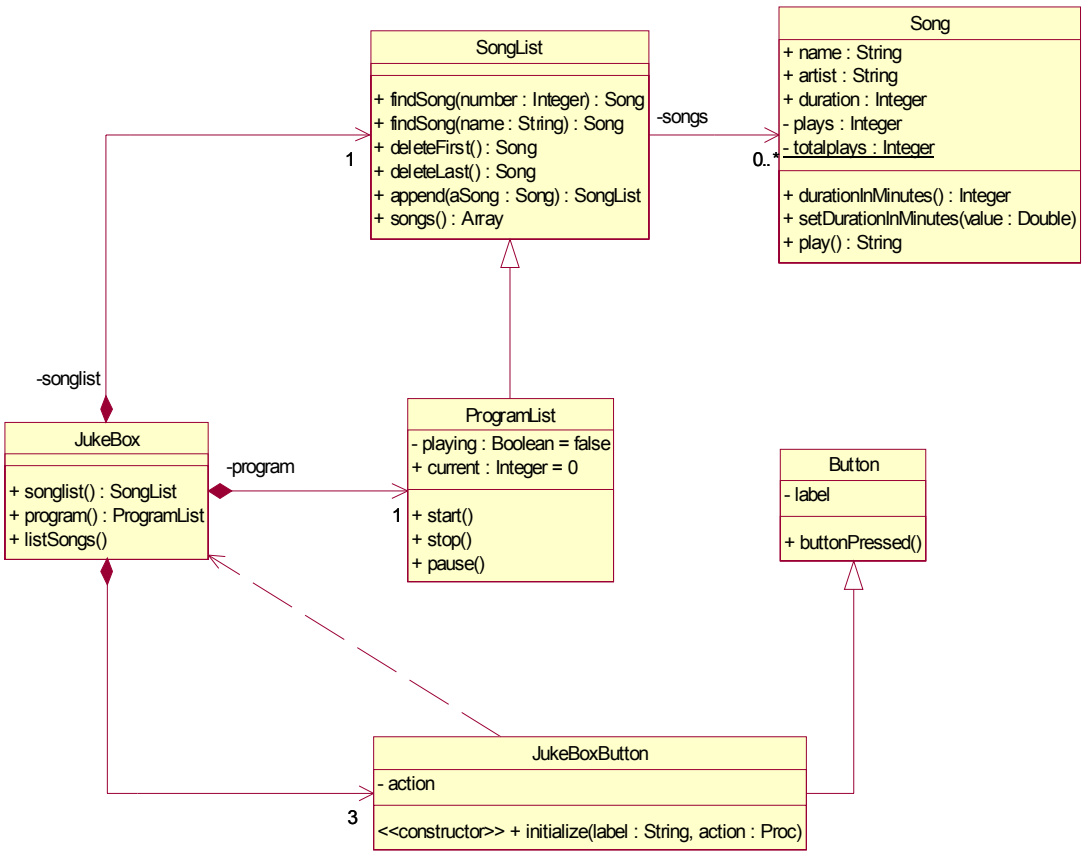


Figure 3.2-6 Jukebox composition UML

Summary - translating composites

From Ruby to UML

- We have a composition relationship between to classes when the composition extension is used. Classes that include the composition module are the composition whole referencing the parts.
- Associations marked *comp* are strong (default) composition relationships.
- Associations marked *wcomp* are weak composition relationships, stereotype them «weak».

From UML to Ruby

- Include the composition module for whole classes
- Mark the associations (references) with *comp* for unstereotyped- and *wcomp* for «weak» composition relationships
- Use nested class definitions or anonymous classes for qua classes in the UML diagram

Generalization

Inheritance specifies an "is kind of" relationship between a super class (the general class) and the subclass (the specific class). In the UML a subclass can have one (single-inheritance) or more (multiple-inheritance) super classes. The semantics of a UML generalization are that the specific inherits all the structure and behaviour of the general, and may or may not extend or modify the inherited behaviour.

In the extension of the jukebox that was performed in the previous section when dealing with compositions, inheritance was used several times. I now also extend the Jukebox program with a KaraokeSong class:

```
class KaraokeSong < Song
  def initialize(name artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end

  def play
    super + " [#{@lyrics}]"
  end
end
```

Listing 3.2-7 Jukebox inheritance

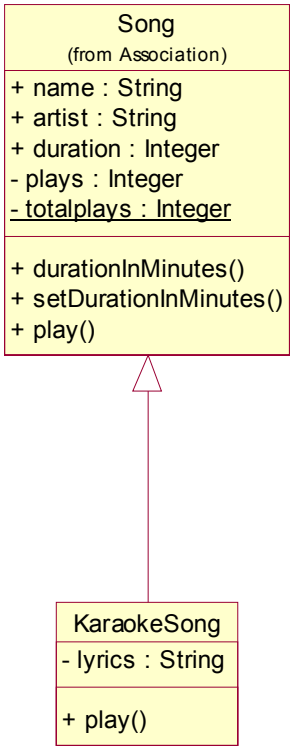


Figure 3.2-7 Jukebox inheritance UML

KaraokeSong is a subclass of Song. Apart from the regular information about a song, a KaraokeSong also has lyrics that are displayed when the song plays.

Ruby specifies inheritance using the < operator: *class ‹class name› < ‹superclass name›*. Ruby does not support multiple-inheritance, instead it supports including modules. A module that is included in a class is called a mixin. The following paragraph explains modules in detail.

The Ruby Module

A module is a way of grouping together methods, classes and constants. A module differs from a class in two ways:

- A module cannot be instantiated.
- It implements the mixin facility, allowing them to be included in other classes or modules.
- It can be included in another namespace, for example the top level of a ruby program.

```
module Amodule
  def methodA
    #...
  end
end

class Asuperclass
  def methodB
    #...
  end
end

class Aclass < Asuperclass
  include Amodule
  def methodC
    #...
  end
end
```

Listing 3.2-8 Mixin example

Listing 3.2-8 is an example of a module used as a mixin to a class. It is now possible to invoke all three methods on an instance of Aclass. To prevent name clashes when several modules are included a module defines a namespace. A class or module can mix in an arbitrary number of modules, and because a module defines a namespace, many of the drawbacks of multiple-inheritance are avoided. What happens when a class includes two modules containing a method with the same name? The answer is that the same rules as when redefining methods applies: the objects will respond to the last definition, in this case the instance method of the last module to be included. To avoid this confusion all instance methods are available using the namespace of the module: *‹module name›.‹instance method›*. Since the class inherits implementation it is not like an interface class, where the class realizing the interface has to provide implementation for the interface class' operations. For this reason the inheritance features of Ruby have been called "single inheritance with implementation sharing".

How Ruby represents the mixin of a module into a class gives us a hint about how we can represent it in UML. When a module is included in a class (client), the module's constants, class variables and instance methods are bound to an anonymous super class. Anonymous meaning inaccessible, one cannot navigate to this super class using the *super* keyword. Objects of this class will now delegate unknown method calls to the anonymous super class. If several classes include the same module, their anonymous super class is the same. A consequence of this is that when a module's behaviour is changed, the behaviour of all the classes that include the module is also changed. If the module has instance variables, these will become instance variables of the class that includes the module. This is because an instance variable is bound to the instance when it first is used. In case of a module this will happen when the first call to an instance method using the instance variable in that module is made. The module itself does not have instance variables since it does not have instances. Another way of saying this is that the instance variables are injected into the client class.

A mixin in Ruby has so much in common with the semantics of generalization in the UML, that we can indicate it with the same notation. A note here is that this brakes with the idea of a "is kind of" relationship. The mixin of a module does not mean that the class is a kind of that module; rather that it gets access to the behaviour of the module.

I choose to stereotype a module used for mixins with «mixin» in UML.

Stereotype «mixin»:
- Can be inherited from, but can only inherit from other <<mixin>> classes
- Cannot be instantiated

The example from Listing 3.2-8 translated into UML is shown in Figure 3.2-8 Mixin in UML:

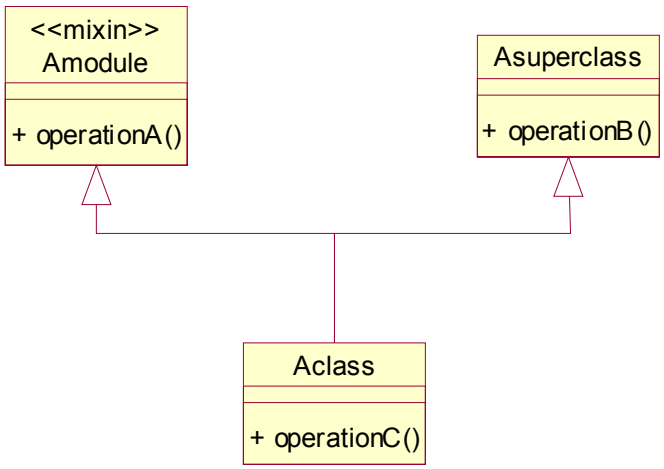


Figure 3.2-8 Mixin in UML

Both classes and modules can contain nested class or module definitions. This was used for qua classes in the previous section. Within the scope of the module or class the nested classes can be instantiated with the normal syntax. Outside the scope you can instantiate the same classes with the following syntax *‹name›::‹class name›.new*.

To include a module at the top level you first have to import the contents of the file containing the module definition using *require 'filename'*. The module itself is then included using *include ‹modulename›*, in which case the module's constants, class variables, and instance methods become available at the top level. Now we do not have to use the *Modulename::Constantname* or *Modulename::classname.new* syntax to access properties of the module. The instance methods of the module are brought into the current namespace. Module methods, or singleton methods in Ruby terminology, can be invoked without including the module first using *‹module name›.module_method*. This corresponds to how class methods can be called without making an instance of it first. An example might be in place (Listing 3.2-9)

```
module Math
  def sin(x)
    #...
  end
  module_function :sin     #same as def Math.sin … end
end

Math.sin(1)                #Here the module function is called
include Math               #The module is included
sin(i)                     #The instance method is called
```

Listing 3.2-9 Module function example

The instance method and module method are two different methods: The call module_function makes a copy of the instance method not an alias.

Include is useful for providing the mixin functionality, but it is also used as a way of bringing the constants, class variables, and instance methods of a module into another namespace as was done above. A special case is when a module is mixed into a class using the *extend* keyword instead of *include*. Doing this will make all the module's instance methods class (singleton) methods of the class instead of instance methods.

Summary – translating generalization/specialization

From Ruby to UML:

- When the class uses the < operator the left hand class is the specific class and the right hand class the general class.
- Ruby only supports single inheritance, but multiple inheritance of implementation is possible through the use of mixin modules.
- Mix in of modules are indicated by an *include* keyword. A client can include several mixin modules.
- Modules being mixed in become classes stereotyped «mixin» in the UML diagram with a specialization between the client class and the module.

From UML to Ruby:

- Generalization relationships are put into the class definition of the specific using the < operator.

- Multiple - inheritance cannot be directly mapped to Ruby. Such situations can be solved using mixins.

Abstract classes and interfaces

Several object oriented programming languages support the concept of an abstract- and interface class. In the UML abstract classes are specified with the class name in italics. An abstract class can't be instantiated, but specifies a set of operation signatures that must be implemented by specific classes. Thus an abstract class is an interface specification. The abstract class does not have any direct mapping in Ruby. The module concept in Ruby is not the same as an abstract class, because operations in an abstract class do not have any implementation. Because abstract classes and interfaces are important concepts in possible target languages such as C++ and Java, it would be a good thing to have the same concept in Ruby.

Using the Module class as basis, I added the possibility to define methods as abstract (Listing 3.2-10). I also aliased the include method to implements for readability. To define an abstract class you define a new module and the abstract methods. If a class that implements this module does not provide implementation of these methods an exception is thrown. Now we know that any class that implements a given "abstract" module, respond to the methods defined in it.

```
class Module
  def abstract(*refs)
    for ref in refs
      name = ref.id2name
      class_eval << EOD
        def #{name}(*args)
          raise NotImplementedError, "#{name} not implemeted"
        end
      EOD
    end
  end
  alias implements include
end
# Example of an abstract module
module IStack
  abstract :push, :pop
end
# Example of a class using this module
class ConCreteStack; implements IStack
  def initialize
    @stack = []
  end
  def push e
    @stack.push e
  end
  def pop
    @stack.pop
  end
end
```

Listing 3.2-10 Abstract class Ruby extension

The UML makes a distinction between an abstract class and an interface. An interface cannot be instantiated, does not have any structure (so they may not have any attributes), or implementation (so they may not have any methods, which provide the implementation of an operation). An abstract class like the interface cannot have instances, but it can have attributes as well as implementation of some or all operations (methods). Another difference is that interface span model boundaries. It can be used to specify the interface for logical abstractions (i.e. classes) and physical abstractions (i.e. components). These differences are important to map to Ruby. The semantics of the interface dictates that if a Ruby module is meant to represent an UML interface it cannot provide implementation for it's methods (marking all of them *abstract*), or have instance variables. If on the other hand the module should correspond to an abstract class it can have methods, variables and abstract operations. To make the difference easily readable interface module names begin with an "I" and abstract module names do not. This has been done above, resulting in the following UML diagram.

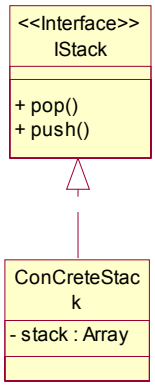


Figure 3.2-9 Interface realization

Summary – translating interfaces and abstract classes

From Ruby to UML

- A module that defines operations as abstract is an interface if it does not have other methods or variables. It will also have a leading "I" in the module name.
- A module is an abstract class if it defines operations as abstract. It may have other methods and variables.
- A class that uses *implements* to include a module results in a realization relationship between it and the abstract class or interface in the corresponding UML diagram.

From UML to Ruby

- Abstract classes become modules that define the same operations as abstract
- Interfaces become modules that define all operations as abstract, the name should have a leading "I" for readability.

- The class that realizes the interface or abstract class should include the module with *implements*.

Dependencies

The general dependency does not have rigid semantics in UML. The common idea is that if class A has a dependency relationship with class B, class A is dependent on class B and its interface but not the other way around. Making rules for translating dependencies is therefore not possible without more detailed semantics for them. However to express certain implementation details dependencies can be used. The following stereotypes for dependencies might come in use when working with implementation models:

- «use». If a class makes use of another class, either by making a temporary instance of it or get it passed to itself via an operation parameter, and it is important to indicate this in the model, a «use» stereotyped dependency can be used.

- «create». If one class creates an instance of another class, a «create» dependency can be used to indicate it in the model.

- «raises». If a class raises an exception of a certain type, a «raises» dependency can be used between the class and the exception class. Optimally the dependency should be between the operation that raises the exception and the exception class.

- «use scope». This is the result of a Ruby specific implementation, where closures and blocks are passed as parameters to other classes. Closures bring with them the local scope and when they are executed that scope is used. Optimally it should be drawn from the operation that ultimately executes the closed block to the class whose instance sent the closure and scope in the first place. The dependency in Figure 3.2-6 is an example of this. This feature is exotic and should only be used in implementation models.

Again these stereotypes are not part of the translation, only suggestions to the use of dependencies in implementation models.

3.3 Reflection in Ruby

Ruby has powerful reflection capabilities. Reflection is the possibility to introspect, or examine aspects of the program from within the program itself at runtime. Some of the things we can find out about our program using reflection are what objects it contains, the current class hierarchy, the contents and behaviours of objects and information about methods. In the below example we exploit the features of reflection to get information about the types of attributes and methods. It turns out that this is very simple. Class Object, which every class in Ruby is derived from, has a method called class (or its alias type). The Object.class method gives us a string representation of what class an object is instance of. Taking the Song class as an example we can do the following:

```
s = Song.new(”Wild thing”, ”Jimi hendrix”, 300)
puts s.class
puts s.durationInMinutes.class
puts s.name.class

This produces in order:
Song, Float, String
```

Exploiting reflection in languages like Ruby makes it possible to make flexible and self-aware applications, but is also a key feature when extending the language. It gives us an opportunity to change the behaviour of the language without changing the interpreter itself. Reflection plays a major role in the composition and association extension as well as the type logger that was developed.

3.4 Perspectives revisited

In the beginning of this chapter I introduced the concept of perspectives. The idea being that when doing analytical work you can apply three different perspectives; conceptual, specification and implementation. Most often it is the specification perspective that is the most interesting when designing applications and it was argued that implementation and specification are difficult to separate because they are two aspects of the same thing. When developing prototypes in Ruby, all the details of the implementation are laid bare, making it difficult to extract the specification perspective. I therefore introduce a convention of how to organize Ruby code so that the specification view and the implementation perspective are separated.

As explained modules can be used to create namespaces and these modules can be included in other modules to make the constants, classes, and mix-in modules locally available. Further Ruby class and module definitions are open, meaning if a class or module is defined more than one place in a namespace (module) the definitions are merged. Conflicts between the definitions, for example if a method is defined in two separate definitions of a class, are resolved so that the last definition parsed is the valid one. These two features, in addition to organizing code in files, are the keys to the specification – implementation organization.

The specification perspective concentrates on the interfaces of the software, treating the classes as black boxes whose instances respond to different messages. The important things here are the relationships between class instances and the interfaces of their class- and instance operations. This information corresponds to class with attributes and empty method definitions in Ruby, as well as any documentation related to these elements. The idea is to wrap this information in a module called "Specification", and place this in a separate file, or set of files, with distinct names such as "jukebox_spec.rb". Listing 3.4-1 shows the specification view of the jukebox previously used as example in this chapter. To save some space only the specification that corresponds to Listing 3.2-6 Jukebox composition is included.

```
# File jukebox_spec.rb
require UML/composition

module Specification

  class JukeBox; include Composition

    # A button only used in jukeboxes
    class JukeBoxButton < Button

      # Attributes
      # private
      @label
      @action

      # Constructor, takes label of button as string
      # and the action to be performed as code block
      def initialize(label,&action)
      end
```

```
      # Callback, called when the button is pressed
      # invokes the action passed to initialize
      def buttonPressed
      end
    end

    # A programmable list to be played
    class ProgramList < SongList

      #Attributes
      #private
      @playing
      @playthread

      #public
      attr_accessor :current

      # Starts playing the program
      def start
      end

      # Stops playing the program
      def stop
      end

      # Pauses the program when playing
      def pause
      end
    end

    # Attributes
    # private, composite parts
    @startButton
    @stopButton
    @pauseButton

    # public, readonly composite parts
    attr :songlist    #Songs in the jukebox as SongList
    attr :program     #Current programmed songs as ProgramList

    #Constructor, make composite parts
    def initialize(songlist=SongList.new)
    end

    # List all songs in the jukebox
    def listSongs
    end
  end
end
```

Listing 3.4-1 Jukebox specification view

The implementation perspective is then placed in a module called "implementation". This module includes the "Specification" module and adds the implementation details such as classes and actions only relevant to implementation. Several implementations can be made of one specification; all of them include the same specification module. These implementation modules are also placed in a separate file, or set of files, with simple names such as "jukebox.rb".

```
# file jukebox.rb
require 'jukebox_spec'
require 'thread'

module Implementation; include Specification
 … code as in Listing 3.2-6
end
```

Listing 3.4-2 Jukebox implementation view

When running the prototype, the files making up the implementation perspective of the application are put together and run, for example by using test cases. The relationship between the specification perspective and implementation perspective modules is depicted using the package diagram below.

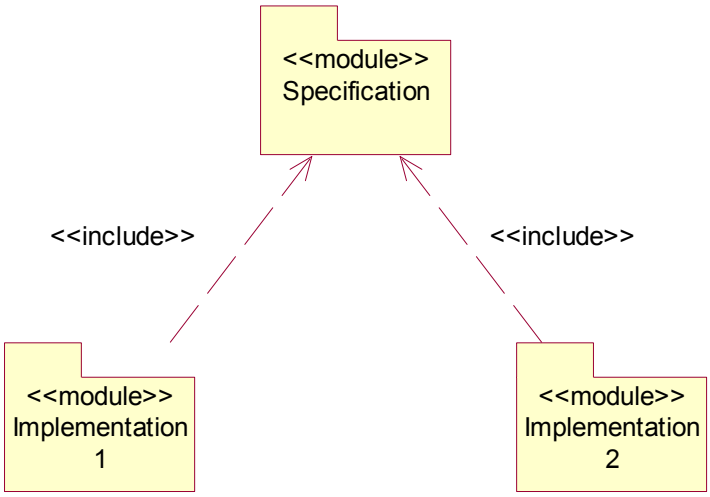


Figure 3.4-1 Specification and implementation module relationship

The problem with this organization of code is that we introduce yet another document that must be maintained together with the different prototypes and UML diagrams. Another solution is to include only elements relevant to a specification perspective when translating a prototype into UML. By tagging classes and other elements as implementation specific in Ruby, these can be left out when doing the translation or hidden in the resulting diagrams. Separating between a specification and implementation perspective in the first way will perhaps result in better organization, and the ability to make several implementations of the same specifications, but requires more work to maintain. Leaving the separation out is quicker but may introduce ambiguities about what is specification and what is implementation.

4 CASE STUDY

Description of experiment

Having developed a method for combining UML class diagrams and Ruby in the last chapter, this chapter describes the details of the experiment where the mapping was applied. It will answer how the problem domain was chosen and what elements of uncertainty and complexity that existed. Further the requirements of the application are presented, followed by an explanation of how translations were done. The last section of the chapter is a textual presentation of the resulting application design.

4.1 Choice of problem domain

When developing an example application or performing a case study, the choice of the problem domain is important. The problem was defined in collaboration with Agresso, because employees of Agresso were to be example users and domain experts.

Agresso delivers software, targeting middle sized to large sized organizations. Their main focus is solutions for economy, collaboration and e-business, with high emphasis on security and performance. The flagship product is Agresso Business World (ABW), an integrated solution supporting many different aspects of business administration.

Because ABW covers so many different problem domains, a choice was made to find a case within one of those. Optimally the best example would be a problem requiring a solution with very high levels of dynamics, where Ruby hopefully would excel. A typical example of such a problem within the range of ABW is workflow management. But the complexity of such an application was regarded as too high to serve as a limited example in this thesis. Also we made the realization that if Ruby was to be used as a general prototyping and specification tool, the example would not necessarily have to be tailored for Ruby.

The final choice was to focus on resource hour and absence registration. In ABW this is a complex module, with much flexibility, but still a reasonably sized problem to serve as an example. I would also have good access to people that had experience with these areas of ABW, either as developers or requirement engineers. In parallel the same case was used in an ongoing project, researching the migration of ABW to the Microsoft .NET development platform. The task in the experiment was to specify the business logic of a client independent component interfacing the Agresso database, running on an Oracle 9 platform. Client independence here means that it should be possible to use the logic from a range of client applications such as web, windows32, web services or xml and mobile applications. This also meant that no user interface was to be developed. The decision about not implementing a user interface was also influences by the fact that problems related to user interface and design [Meyers93] are not solved using either UML or Ruby.

4.2 Complexity and uncertainty in the problem domain

In this section some elements of complexity and uncertainty will be identified. It was important to choose and design the experiment with some elements of both. This was because there had to be reasons for both experimentation and analytic behaviour when doing the experiment.

Complexity

An hour and absence registration can at first glance seem to be a fairly simple system to design and implement. However, a closer look reveals some of the elements of complexity that had to be dealt with:

- An object oriented design supporting the task of delivering timesheets and register absence has to be developed.
- The tables and rules of the Agresso database related to the problem have rigid rules and are not easily mapped to an object oriented design.
- The hour and absence data reference, and is referenced by, other data in the Agresso database.
- This data varies from company to company.
- The different clients can request and provide varying amounts of data
- There can be many errors in the data, validating a timesheet can be a complex task.
- The different clients can have very good or very limited ability to validate data.
- There is a conflict between how flexible the application should be in dealing with invalid data and the ability to raise detailed exceptions and expect clients to deal with them.

There was no common timesheet or absence component that could serve as an example. Each client had different implementations suited to the particular client's needs, the common ground being the database. The data in the database are transferred into other parts of the Agresso solution by server processes. In some respect the task was to find the things common to every client and reengineer the best from the clients. I did not have access to source code for the clients.

Uncertainty

The elements of uncertainty are somewhat inherent in the complexities listed above. I chose not to use a detailed specification of the Agresso database, even if such a specification existed. The uncertainties rose from the following facts:

- I had to rely on myself and other people to understand the rules and tables of the Agresso database. I could not guarantee that these other people fully understood it or if misunderstandings occurred.
- The peoples' view of the existing systems in ABW influenced how they envisioned and used the system themselves. The ABW time and absence module is complex and I could not be sure if they understood everything they told me.
- People told me conflicting things about how they envisioned the system to be. This because they worked with different clients and applications.

4.3 Requirements

This section describes the requirements of the hour and absence system. The requirements presented here are based on communication with domain experts (Agresso users and developers) and what was learned during the experiment. These requirements are therefore not necessarily the same as when the experiment was started. Rather they were iteratively defined during the development of the prototype and specification, and user communication. Further they are influenced both by the layout of the Agresso database and the process of delivering and maintaining absence and hour data. The description begins with a textual explanation followed by a description of the general use cases that the system must support.

4.3.1 Narrative description

The hour and absence registration is the individual employees' responsibility. It is assumed that the employee enters data on a regular basis, often every week. Data on how periods are distributed in time can vary; the employee must deliver a set of hour or absence data each period.

The hour data is organized in time sheets. These contain information about how many hours the employee has worked in the period of question, and how these hours are distributed on different jobs. The job description contains information about what project the job is organized under, what work order (job assignment) the job fulfils and what kind of activity the job included. The combination of project, work order and activity follows the rules defined in the project management module, defined in the database. The minimum information to enter is project. If the project has work orders, work order information is mandatory. The same applies for activities. Based on the realization that an employee will work on the same type of jobs every so often, charge codes are defined containing legal project, work order and activity combinations. These codes must be assigned to an employee before that employee can register hours using the codes directly. Internally every job is assigned a charge code, even if the data is entered without the use of the code. Codes that do not exist must be generated upon storing a time sheet.

Each employee has a certain amount of hours that must be distributed every week, based on a percentage of a full week (normally 40 hours). This information is available in the organization global data (called client in ABW, there can be many clients in a company) and in the employee register for that organization. The amount of hours distributed per period and per day must be calculated. This is so the application complies with overtime and flexi time rules. The type of hours must be registered. Examples of different hour types include regular hours, overtime, billable hours and time off due to positive number of flexi hours. The hour types are can be defined freely, and is also tied to organizational data.

The employee can edit and draft a timesheet for later amendments, until a certain date where it has to be delivered. This means that it must be possible to store illegal or incomplete data in the database. Special temporary charge codes must be generated to store illegal values. When delivering the timesheet must be complete and legal, any temporary charge codes are deleted. It is possible to deliver a timesheet after the delivery date, but the employee should be made aware of any outstanding deliveries.

The time sheets must be approved before further processing. The person responsible for approving is defined on a per project basis. Hours on a project can be defined to be the resource's responsible, the project responsible, the work order responsible or it may be that hours does not need approving. This way it is possible that several persons are involved in approving one

timesheet. If every job on the timesheet is approved it is sent for further processing. If the approver rejects one or more jobs, the timesheet must be corrected and redelivered by the employee.

Absence data is in many ways similar to hour data, at least internally in the database. The absence data is stored in the same tables as hour date, differing with special projects and hour types. Absence types vary on how they affect the flexi time account, absence projects have additional activities that define what type of absence it is. Examples are doctor appointments, vacations, maternity leave and sickness. Often employees will tend to add more notes to an absence registration than a job description.

Delivering, correcting and approving timesheets can be helped by a task list. Especially managers responsible for approving many different timesheets can benefit from seeing how many timesheets that need approving and have quick access to them. At the same time employees will benefit from being reminded when a timesheet delivery is overdue or when a timesheet is rejected and need correction.

4.3.2 Use cases

The use cases identified are shown in Figure 4.3-1, each use case is described using a template from A. Cockburn's "Structuring use cases with goals". This use case diagram is depicting the scope of the experiment rather than an initial use case diagram that was developed before starting the experiment.

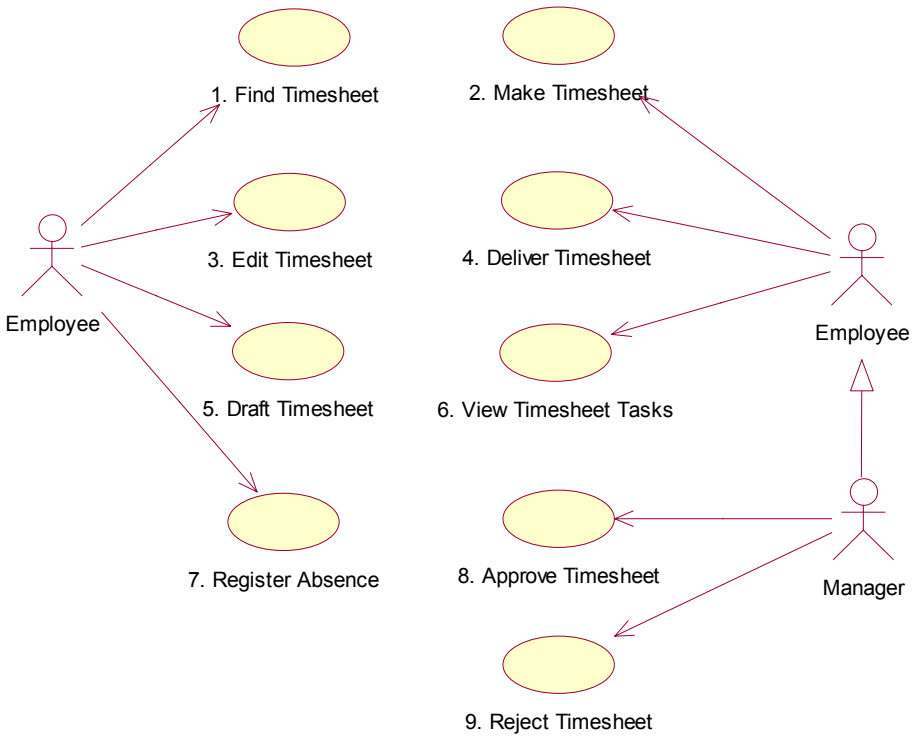


Figure 4.3-1 Use case diagram

The actors in the use case diagram are typical client application users. Employee is a resource of the company that is responsible for maintaining hour and absence data. A manager is every resource that can be responsible for approving hours on a project. This can be another resource's manager, a project manager or a work order issuer. A manager can also initiate the same use cases as an employee, indicated by the generalization relationship.

Name	1. Find Timesheet
Initiator	Employee
Goal	Find a timesheet for a given period
Pre conditions	One or more timesheets has previously been stored or drafted. Period information is stored in the database.
Main success scenario	1. Data identifying the resource and the period in question is supplied. 2. The timesheet is loaded
Exceptions	1. a) Either the resource number or period number is illegal 1. No timesheet is found, exception is raised 2. a) No timesheet exist for the period and resource. 1. No timesheet is found, new one is made

Name	2. Make Timesheet
Initiator	Employee
Goal	Make a new timesheet for a given period
Pre conditions	Period information is stored in the database
Main success scenario	1. Data identifying the resource and the period in question is supplied 2. A new empty timesheet is made and loaded
Exceptions	1. a) Either resource or period information is illegal. 1. No timesheet is made, exception is raised 2. a) A timesheet already exists for the given period. 1. That timesheet is loaded

Name	3. Edit Timesheet
Initiator	Employee
Goal	Amend or change details of a timesheet
Pre conditions	A timesheet is loaded, through use case 1 or 2. Project, work order, activity and hour type information is defined in the database. The loaded timesheet is editable, meaning

	in a draft or to be corrected status.
Main success scenario	1. A job description is entered 2. The data is validated 3. Hour totals are calculated 4. Repeat process
Exceptions	1. a) A job description is to be deleted 1. Indication of what job to delete is supplied 2. That job is deleted 3. Go step 3. 2. a) The data is invalid 1. Raise exception for client to catch, store job description anyway

Name	4. Deliver Timesheet
Initiator	Employee
Goal	Deliver a complete timesheet
Pre conditions	Timesheet made and complete
Main success scenario	1. Timesheet is delivered 2. The whole timesheet is validated 3. Charge codes are found 4. Timesheet header is stored 5. The job descriptions are stored 6. Approve tasks are generated 7. Relevant deliver task is deleted
Exceptions	2. a) Timesheet found invalid 1. Timesheet is saved as draft (UC 5) 2. Exception is raised 3. a) No matching active charge codes 1. A charge code is made 6. a) The timesheet has some rows that need approving and others that do not. 1. An approve task is generated for every row that needs to be approved. 2. The rows not to be approved are stored with a status so they can not be changed later.

Name	5. Draft Timesheet
Initiator	Employee
Goal	Draft a timesheet for later amendments
Pre conditions	Timesheet made
Main success scenario	1. Timesheet is drafted 2. Timesheet header is stored status draft 3. Temporary charge codes are made 4. The job descriptions are stored

Exceptions	None

Name	6. View Timesheet Tasks
Initiator	Employee
Goal	View the task list related to hour and absence
Pre conditions	None
Main success scenario	1. Task list displayed 2. Timesheets related to each task are easily available
Exceptions	1. a) The task list may be empty 1. An empty list is returned

Name	7. Register Absence
Initiator	Employee
Goal	Register time away from work
Pre conditions	Absence project and activities are defined. Period information is available
Main success scenario	1. Absence information is supplied 2. Hours to distribute / flexi time account is calculated
Exceptions	2. a) The absence type does not affect flexi time by definition. 1. Flexi time account not changed

Name	8. Approve Timesheet
Initiator	Manager
Goal	Approve hours for a specific project in a timesheet
Pre conditions	Timesheet validated, delivered and approve task generated.
Main success scenario	1. The timesheet is loaded 2. One or more job descriptions are approved by the manager 3. Relevant approve tasks are deleted 4. The timesheet is saved for further processing
Exceptions	2. a) The manager decides to reject a job description – Use case 9.

Name	9. Reject Timesheet
Initiator	Manager
Goal	Send a timesheet back to the resource for correction.
Pre conditions	Timesheet validated, delivered and approve

	task generated
Main success scenario	1. The timesheet is loaded 2. The manager selects one or more job descriptions for rejection 3. Approve tasks are deleted 4. Correct task is generated 5. Timesheet is saved
Exceptions	None

4.4 Translations

The translations were carried out following the rules defined in the translation model. This section will describe how the translation model was used in the experiment and define some terms used in the analysis.

Apart from some tools developed for upholding composition-, and association semantics and to deduce type information from test cases during execution, the translation was done manually by reading Ruby or UML and mapping it to the other language.

- Definition: Translation, translating from Ruby to UML or vice versa

During the development of the experiment four complete translations, meaning all of the Ruby code developed was translated to UML and back into Ruby after possible changes. In addition some smaller parts of the solution were translated Ruby-UML and UML-Ruby several times.

- Definition: Ruby-UML translation, translating Ruby to UML, keeping actions as comments on the UML operations.

- Definition: UML-Ruby translation, translating UML to Ruby, inserting commented actions into the methods and removing or adding anything that has been changed since the Ruby-UML translation.

The starting point of the development was a conceptual model in UML, developed on basis of the initial requirements and communication with users. From this a Ruby skeleton was generated. Different situations triggered new translations, often the need to communicate with users and then trying to find solutions.

- Definition: Behaviour shift, changing mode of operation from an analytical one to an experimental one, or vice versa

- Definition: Experimental behaviour, learning about the problem by trying out solutions in the form of prototyping. Ruby was the main language for experimental behaviour.

- Definition: Analytical behaviour, learning about the problem by analysing the reality and information from domain experts using specification. UML was the main language for user communication and analytical behaviour.

When performing the translation, weaknesses and problems with the translation model was discovered and it was changed accordingly.

4.5 Communication with domain experts

The employees in Agresso R&D are familiar with UML through a RUP centred course they followed in the last half of 2001. The plan behind this course was to increase the use of UML in Agresso. However I believe that UML is not systematically used on an overall basis in the company. The people acting as users in my case had varying experience in UML and object oriented development. Some were application developers and others were requirements engineers communicating with end users of ABW and not participating in implementation of applications.

Agresso started a research experiment, using Ruby as an action specification language in translational code generation from UML. This research was later stopped in favour of another technology. It is my impression that focus on Ruby has been abandoned and the language is not systematically used in development. Some people in Agresso, however, know Ruby and apply it in personal work. Some of these people participated only very early in the experiment. This was either because they started to work at other subsidiaries of the company located in England and the Netherlands or went to work in other companies. None of the users involved in the later stages of the experiment was familiar with Ruby.

The most usual form of user communication was informal conversation or mail exchange between me and one or two users about specific areas of the domain or the existing agresso solution. When complete translations were done, more formal discussion and reviews were held.

4.6 Resulting design

Rather than explaining the resulting design in detail, I will concentrate on a couple of special cases in it. The full design of the component in UML together with the Ruby source code for it is available on an accompanying CD-ROM. In addition the UML design can be found in the appendix.

It is recommended to browse the Ruby code from RDoc (start ./ruby/Final/doc/index.html from the CD-ROM) when looking at the UML models, because this makes it easy to navigate to specific parts of the code related to specific parts of the UML model. The code and models is also thoroughly documented.

The first special case to be treated is found in the implementation view of the model. As mentioned in the translation model, Ruby has the capabilities of mixing in modules, also referred to as multiple implementation inheritance. This has been applied when the task list of a user is updated by changes in time sheets. An observer pattern has been applied, and we can see an example where using modules can implement a pattern without requiring the client classes to give any additional implementation. The module Observer provides implementation securing that the answer to an update method. This module is mixed in by the TaskList class, that redefine the update method to take a TimeSheet or TimeSheetRow as an argument and find, create, delete or update a relevant task based on the data provided. The module Observable provides methods for adding and deleting observers. In addition it injects structure in the form of a relation to all the current Observers to the client class (see "The Ruby Module" in 3.2.2 for how this works).

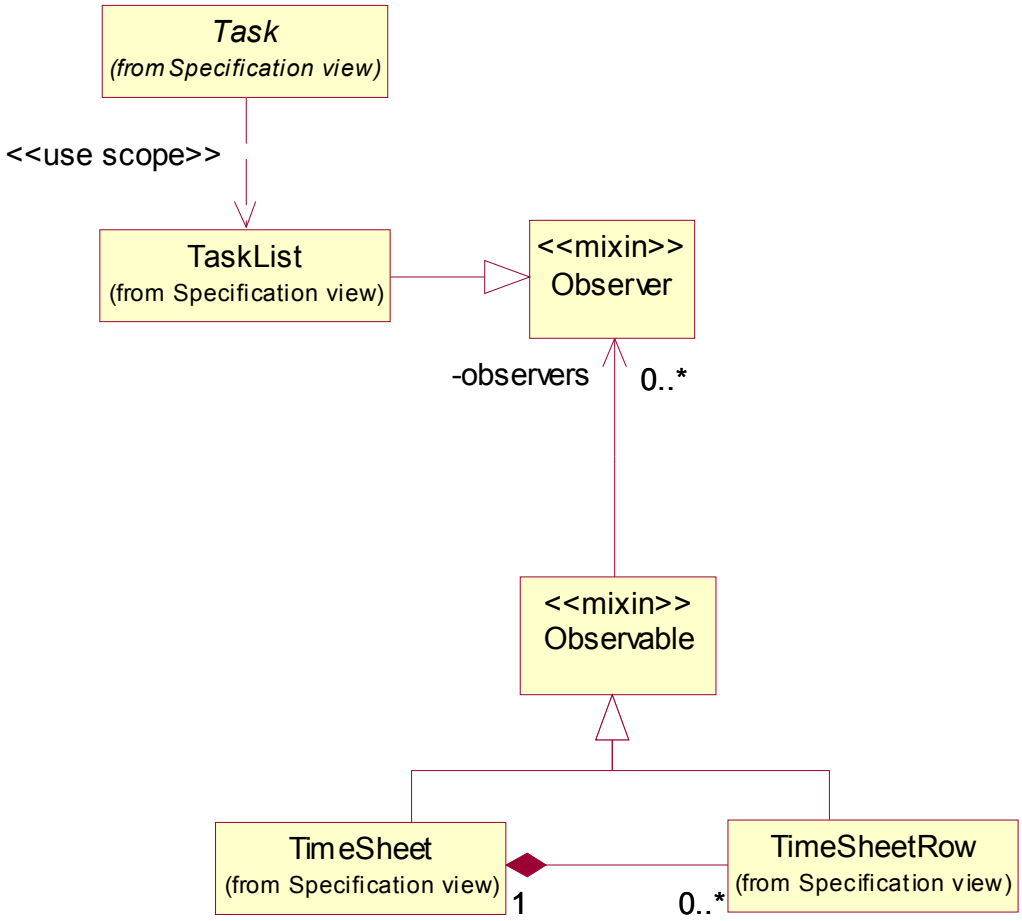


Figure 4.6-1 Use of mixins in the solution

As treated in 3.2.2, module mixin is represented in UML as the client class inheriting a stereotyped class. A simple interface realization is not enough since a mixin provides implementation and can inject structure to the client class. Another special case in Figure 4.6-1 is the "use scope" dependency from Task to TaskList. This is because TaskList calls a method in Task using a block, and that method yields the scope of the TaskList to delete itself if it is completed. The use of such special Ruby specific cases should be avoided if the goal is a language-independent design. However this is the implementation model where generality has been sacrificed over rapid development. It is absolutely possible to change the implementation to be more general.

The second case I want to treat in this section is due to a design dilemma rather than the use of Ruby or UML special features. Ponder Figure 4.6-2 and Figure 4.6-3.

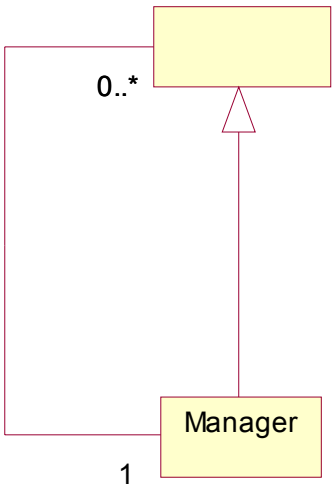


Figure 4.6-2 Resource specialization

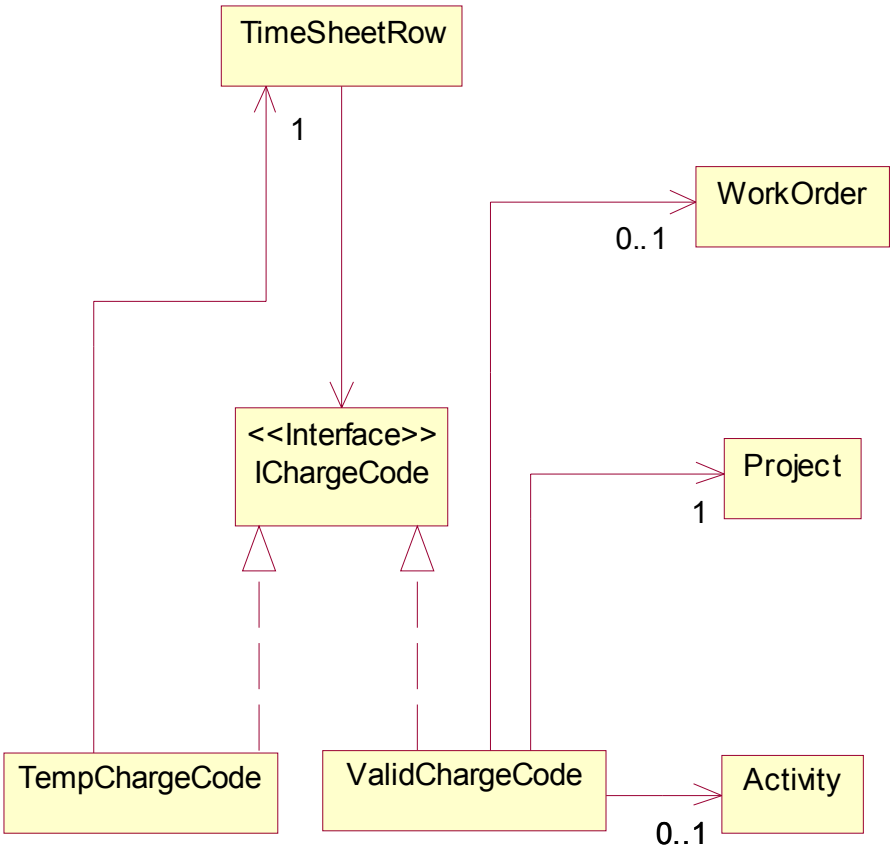


Figure 4.6-3 IChargeCode Interface

Figure 4.6-2 shows how that Manager is a specialization of Resource, because it has an association to the resources that it manages. Figure 4.6-3 shows a similar slightly more complicated case, but different in the sense that generalization has not been used. It also violates UML because a class cannot have an association to an interface. This is according to the textual description of meta-class AssociationEnd in the UML1.4 specification stating that the association end points to classes whose *instances* can be reached over through the association [OMG00]. Thus an AssociationEnd can not point to an interface or even an abstract class because they will never have instances. The above is violation of text, but probably not intent and is fixed in UML 2.0 [OMG02].

Why is not generalization used here too, stating that TempChargeCode and ValidChargeCode is special chargecodes? Well TempChargeCode and ValidChargeCode are very different when it comes to structure. They don't share any structural elements at all. All they share are methods to tell if they are valid, validate them and calculate their sequence number (which is a database issue), and these are implemented differently in both classes. The TempChargeCode has a relation to a TimeSheetRow as there it exists only in relation to one of these. Further it keeps temporary values as string attributes as opposed to the ValidChargeCode that has relations to a valid project, work order, activity combination. ValidChargeCode objects are reused in several TimeSheetRow objects. A TimeSheetRow always have an association to one of these two charge codes, hence the illegal association (as a side note this is perfectly legal in for instance Java).

The alternative of introducing an abstract class and specialize it seemed like overkill. In Ruby we would not even need the interface class because of dynamic typing. This is also an example of something that in a relational database is a perfectly reasonable solution might be difficult to map to an object-oriented design.

5 ANALYSIS OF THE RESULT

Analysis and discussion of the translation and the experiment

This chapter discusses the findings relevant to the problem of the thesis 5.1 discusses how Ruby performed as a prototyping language, 5.2 analyses the experiences gained from using the combination of UML and Ruby in the experiment, 5.3 regards the translation experience and finally 5.4 looks at problems that rose due to difference in the nature of Ruby and UML.

5.1 Experiences using Ruby as prototyping language

Promoters of Ruby claim that the language is very good for rapid prototyping [RubyWeb]. This is also a claim of several other interpreted scripting languages. Rapid prototyping can mean two things [Budde91 p8.]:

1) To be able to quickly produce working experimental versions of the final application facilitated by Very High Level Languages and powerful interactive development tools.

2) When used as the only valid document in the system development process an unsystematic, trial – and – error approach to software development.

As the latter has a negative ring to it I presume that the promoters mean rapid prototyping in the first sense. This definition is very general as there are many approaches to the use of prototypes in software development [Budde91]. In the experiment I wanted to test how Ruby performed in a real prototyping effort. I wanted to objectively answer the question: Is Ruby a good prototyping language? This question forces a subjective answer. It is better to ask: What features of Ruby make it a good prototyping language? To answer this we must first look at what the general features of prototyping languages are, and what I found evident in Ruby when developing the example.

Many languages claim to be prototyping languages. While some are custom made for one type application or problem domain, others are more general and aim to solve larger problem sets. Ruby falls into the last category.

- A main feature of a prototyping language is that it should have a fast turn around, or code – test – correct cycle. This is typically a feature of interpreted languages as discussed in 2.4.3. An important issue in prototyping is fast development and low cost. A fast turnaround is only a part of this issue.

- Prototyping languages are often semantically rich, small in the sense that they require less typing to get things done over languages typically used to develop production systems and they have large general and domain specific libraries.

- Since the requirements changes more often when prototyping, the prototyping language must easily cope with changes of varying implications to the whole prototype. Of course the programmer must uphold the rule of low coupling and high cohesion himself, as this is not the responsibility of the programming language. The language can provide maintainability by being concise and semantically rich, but be readable.

- Another desirable feature is to be possible to test small parts of the requirements atomically, without regards to the rest of the prototype, and larger parts of the requirements by combining tests or run the whole prototype. Often the prototyping environment includes some way to interactively browse and run code; a perfect example of this is the Smalltalk programming environment [Goldberg84]. Another way is to provide tools for unit testing and test driven development, much used in the Extreme Programming paradigm [Kent99].

- When prototyping the language should not require you to make decisions to early on, meaning you should be able to defer some decisions not relevant to the problem at hand. This also corresponds to the ability to cope with change, and be able to run smaller parts of the prototype isolated. An example can be deferring the decision about the details of the interface of an operation until you are fairly sure that you need the operation and what would be the best interface.

Above I listed typical features found in prototyping languages. Which of these and others was found in Ruby and used in the hour registration system example? Ruby was used in combination with UML in a design effort, where Ruby played the role as an action specification language and to develop a running prototype. In the following subsections I compare the typical features of prototyping languages to the features of Ruby

5.1.1 Developing, running and testing code in Ruby

Ruby is an interpreted object oriented language (may also be referred to as a scripted OO language). Being interpreted there is no compilation phase, compilation time and runtime is the same. This facilitates the fast code – test –correct turnaround mentioned. I was able to write arbitrary amount of code and run or test it in several ways, isolated or together with other parts of the application. Basically I ran code in three different ways: From the editor (emacs C-c C-c), through test cases and the interactive ruby interpreter. In the first case I added test at the bottom of the file that was open in the editor. I will refer to this kind of tests as inline tests. Listing 5.1-1 below shows an example where the Period class is tested at the bottom of the file. Ruby executes the statements relevant to the top-level object sequentially (there is no main method like in Java). The definitions are also executed. For example it is possible to conditionally define classes, modules and methods.

The test code should run only if the file is run directly and not when it is required or loaded into other files. To ensure this the test code is wrapped in an `if __FILE__ == $0 … end` construct, this compares the name of the file ruby is parsing to the name of the program or main file. A good thing about this way of running code was that it was fast, it was not necessary to resort to the command line or the interactive environment. A problem was that it was tiresome to maintain the tests since they coexisted with the prototype code, and they were distributed throughout the files. Inline tests were good as a kind of ad-hoc testing of code. It was used several times when new methods were added to a class, to see if they did what was intended. The inline tests were not meant to test every possible side effect of the methods.

```
class Period ... end
class PeriodList ... end

if __FILE__ == $0
  def print(per)
    p per.id
    p per.dateFrom.to_s
    p per.dateTo.to_s
```

```
    p per.description
  end

  # Test creation, next and prev
  per1 = Period.new(200250); print(per1)
  per2 = per1.next; print(per2)
  per3 = per2.prev; print(per3)

  # Test ==
  raise("== wrong!!") if !per1 == per3

  # Test transition to next year
  3.times do
    per2 = per2.next
  end

  # Test currentPeriod class method
  per5 = Period.currentPeriod
  if !(per5.dateFrom < Time.now < per5.dateTo)
    raise("currentPeriod does not work")
  end
end
```

Listing 5.1-1 Inline testing

Many of the inline tests were moved to separate files and incorporated into larger test cases for better maintainability. Writing tests that find and report problems is time consuming, a lot of the test code consists of if conditions and expressions reporting errors or raising exceptions. The RubyUnit and Test::Unit libraries provide a framework for easier unit testing in Ruby. Unit testing is heavily used in extreme programming and test-first development. I did not systematically employ test-first development when doing the example system. This was because studying test-first development in prototyping was not a focus of interest when I started to work on the example. At a later time I found out that employing unit tests through the use of these frameworks was a good way to run the code produced and check if it did what it was supposed to do. Listing 5.1-2 shows a rather large test case for the TimeSheet class at one point of the development. Using these test cases the number of possible errors that the prototype handled increased during development. Relevant cases were run whenever methods were added or refactored.

```
require 'time'
require 'period'
require 'chargecode'
require 'resource'
require 'test/unit'

# Tests depend on database condition!
# Assumes there is no timesheet stored for
# test user wanda weir - 87010101 in the
# current period. Ensured by tear down

class tc_timesheet < Test::Unit

  def set_up
    res_finder = ResourceFactory.new
    @wanda = res_finder.getResource("87010101")
    @per = period.currentPeriod
  end
```

```
def test_initialize
  @sheet = @wanda.timesheet(@per)
  @todistr = (@per * @weir.percentage) / 100
  assert(@sheet.rows.empty?,
         "There were rows in the database, or error!")
  assert_not_nil(@sheet.voucherNo,
          "No voucherNo retrieved or generated")
  assert_equal(@sheet.resource,@wanda,
          "Initialize stores wrong resource")
  assert_equal(@sheet.@per,"Initialize stores wrong period")
  assert_equal(@sheet.toDistribute,@todistr,
          "To Distribute wrong!")
  assert_equal([0,0,0,0,0,0,0],@sheet.dayTotals,
          "Wrong daytotals!")
  assert_equal(0,@sheet.periodTotal,"Wrong periodtotal!")
end

def test_add_valid_row_and_validate
  @row = TimeSheetRow.new(@sheet)
  @sheet.addRow(@row)
  @row.setCcAttributes("WEB2","WEB2")
  @row.hourType = HourType.new("REG")
  assert(!@sheet.empty?,"Row was not added")
  assert_equal(0,@row.seqNo,"Row did not get right seqNo!")
  assert_instance_of(ValidChargeCode,@sheet.chargecode,
           "Valid attributes made invalid chargecode!")
  assert_not_nil(@sheet.approver,
           "WEB2 has approver for project, sheet has not!")
  assert(@sheet.needsApproval?,
          "Needs Approval attribute is wrong")
  assert(@sheet.approver,@sheet.chargecode.project.manager,
          "project mgr approves WEB2, wrong in sheet!")
  @row.hours = [7,7,7,7,7,0,0]
  assert_equal([7,7,7,7,7,0,0],@sheet.dayTotals,
          "Did not update sheet.dayTotals!")
  assert_equal(35,@sheet.periodTotal,
          "sheet.periodTotal not correct!")
  assert_equal(@todistr - 35,@sheet.toDistibute,
          "sheet.toDistribute not correct!")
  @row.approve
  assert(!@sheet.needsApproval?,
          "Sheet was approved, but still needs approval!")
  assert_nothing_raised{@row.chargecode.validate}
end

def test_add_invalid_row_validate_and_save
  @row.setCcAttributes("WEB2","WEB2","ACT")
  assert_instance_of?(TempChargeCode,@sheet.chargecode,
           "Invalid Attributes made valid chargecode!)
  assert_raises(ChargeCodeException,
           "Invalid cc did not raise error on validation!"){
           @row.chargecode.validate
           }
  assert_raises(TimeSheetException,
          "Managed to deliver an invalid timesheet!"){
          @sheet.deliver
          }
  @row.setCcAttributes("WEB2","WEB2")
  assert_nothing_raised{@sheet.deliver}
end
```

```
  def tear_down
    #clear the database
    db.connect do
      db.transaction do
        db.clear_rows(@sheet)
        db.execute("DELETE FROM atsheader
                    WHERE resource = '87010101' AND period = "+
                    period.currentPeriod.id)
      end
    end
  end
end
```

Listing 5.1-2 Example test case

The interactive ruby interpreter (irb) was also used to run code. Irb is a command line tool that evaluates ruby statements on the fly, outputting the result of every statement. It is particularly useful for experimentation, refactoring and debugging. In Ruby it is possible to extend or change already defined classes because class definitions are not closed. This makes it possible to start irb, load in classes from files and then call methods on them, redefine methods and add new ones on the fly. Using the introspective features of Ruby, the state of the running program is available for querying. I found myself using irb mostly when trying out new ideas for methods. For example at one point I wanted to introduce virtual attributes in the TimeSheetRow class to easier answer questions like: "Do this row need approval?", "Has this row been approved?", "Has it been rejected?" and similar. Listing 5.1-3 shows and irb session where I add two methods that answer if the row has been approved and if it has been rejected.

```
> irb
irb(main):001:0> require 'time', 'chargecode', 'resource'
true
irb(main):002:0> class TimeSheetRow
irb(main):003:1> def approved?
irb(main):004:2> return false if(workflow != 'N' and approver)
irb(main):005:2> true
irb(main):006:2> end
irb(main):007:1> def rejected?
irb(main):008:2> true if(workflow == 'R')
irb(main):009:2> end
irb(main):010:1> end
nil
irb(main):011:0> res = ResourceFactory.create.find("87010101")
#<Resource:0xa0ad0b8>
irb(main):012:0> ts = res.timesheet(Period.currentPeriod)
#<TimeSheet:0xa0a6ad9>
irb(main):013:0> row = TimeSheetRow.new(ts)
#<TimeSheetRow:0xa0a3908>
irb(main):014:0> ts.addRow(row)
#<TimeSheetRow:0xa0a3908>
irb(main):015:0> row.setCcAttributes("WEB2", "WEB2")
#<ValidChargeCode:0xa097b18>
irb(main):016:0> row.approved?
false
irb(main):017:0> row.approve
nil
irb(main):018:0> row.approved?
true
```

```
irb(main):019:0> row.rejected?
false
irb(main):020:0> exit
>
```

Listing 5.1-3 Irb session

Above I have discussed how I ran and tested smaller parts of the prototype during the development. Because there is no compilation phase the turnaround was immediate. There were several options for doing this, each better suited in some situations. There are other tools still that I did not use, such as the ruby debugger, xmp (outputs all statement results) and ri (Ruby interactive). Combining test cases tested a larger part of the prototype. A GUI for the application was not developed; the reason was to limit the experiment to business logic, which is the main application of UML in implementation near UML class diagrams. The use of UML class diagrams in user interface design is not wide spread, and a decision was made not to spend time developing a GUI that would be sole Ruby in the same way as other UML diagrams has no Ruby representation. Of course it would be interesting to develop a GUI to see how Ruby performed here, because it is an important aspect of a prototyping language. But I will suffice to refer to the many GUI libraries available in the Ruby Application Archive [RubyWeb].

5.1.2 Dynamic typing

Ruby is a dynamically and strong typed language. This means that the programmer does not have to declare types of variables, but the application maintains type safety at runtime, raising errors if messages are sent to an object that violated its type. Static typing has two benefits over dynamic typing:

1) Type errors are caught at compile time, eliminating a common source of errors.

2) The compiler can optimize machine code based on type information, which is why statically typed languages have better performance than dynamically typed languages.

Programming language theory often promotes static typing because of these reasons [Ghezzi98 p137] [Wirth74]. At the same time stronger typing rules reduces flexibility and the size of problems that can be solved using the language [Ghezzi98 (p139)]. In prototyping you often need that flexibility, which may be the cause that most prototyping languages are dynamically typed. Non functional requirements like performance and memory consumption are often less stressed or not regarded in prototyping [Sommerville95 (p147)], so the second issue above is less important at least when doing throw away prototyping. Dynamic typing requires less typing and thus faster development. In some statically type languages type checking is still done at runtime when working with generic objects leading to type conversions through casting.

Listing 5.1-4 shows an example where dynamic typing requires less typing and is more readable. If at a later time you realize that MyType could be generalized into MyGeneralType, you will have to change both the type of 'o' and the cast to MyType for all the class' clients. In the dynamically typed language o can be an instance of any class as long as it responds to callMethod.

```
Java:
Iterator iter = aList.iterator
While (iter.hasNext()){
  MyType o = (MyType) o.next();
  o.callMethod();
}

Ruby:
aList.each{|o| o.callMethod}
```

Listing 5.1-4 Iterator comparison

Of course the generalization should be realized earlier or an interface should have been used. But when prototyping such changes happens more often as the programmer learns about the problem domain, and such flexibility is good to have. It also makes it possible to defer such decisions to a later time.

By the use of test cases we are able to test and catch more than just type errors. In the TimeSheet test case in Listing 5.1-2 there is an example of this. When assigning charge code attributes to a row, an assertion is made to check if the type is ValidChargeCode when valid attributes are fed to the set method and TempChargeCode when the attributes are invalid. A static language would only check if the charge code implements the interface IChargeCode. This shows that extensive use of unit testing can weigh up for lack of compile time type checking.

The issue of typing will be discussed further in 5.4

5.1.3 Libraries and reusability

Like Smalltalk all of Ruby's built in classes and modules are open to the programmer. Because everything in Ruby is an object the developer can inherit and reuse classes like Integer, String and Array as well as Meta classes like Module and Class. This is also a way of extending and tailoring the language for specific use. As explained in 3.2.2 I made use of this when extending Ruby to support composition semantics and abstract classes.

A lot of the built in elements, libraries and tools available to Ruby come in the form of modules that can be mixed into classes. I found that the concept of mixins is very good for reusability and faster development. This because the classes that mixes in the module inherits implementation, not only specification. In the hour registration example I could make use of a generic Observable module that implements the Observer pattern [Gamma95] by mixing it into the TimeSheet and TimeSheetRow classes. Other patterns can also be implemented by the use of modules.

5.1.4 Summary

Many of the features found in languages traditionally used for prototyping can be found in Ruby. There are several ways to run arbitrary large parts of a prototype through ad-hoc testing, test cases and interactive interpretation. The interactive environment is good for experimentation with new ideas.

Ruby's dynamic typing and semantic richness require the programmer to type less and produce code fast, at the expense of compile time type checking and performance. Examples of rich semantics include Ruby's collection support, the possibility to pass blocks to methods and yield them from within and strong introspective / reflective possibilities.

Dynamic typing can also give more flexibility and make it easier to adapt changes and defer decisions. UML is superior to Ruby when comparing the ability to gain overview of the structural interrelations of the prototype. Incorporating larger changes to requirements resulting in structural changes is better to do in UML than in Ruby for that reason.

Ruby supports reuse through freely available components and libraries. Documentation and quality of these components grows in parallel with the user group of the language. In Ruby a client class can inherit implementation from a super class or one or more mixed in modules. Many of the libraries available come in the form of modules. This makes it easy to wrap up generic reusable code in modules and use them in different projects. An example is that several of the design patterns described in [Gamma95] can be implemented as modules. Ruby is also a very extensible language, every object of the environment is available to the developer, who can reuse these elements by inheriting their classes or extend the classes by making use of Ruby's open class definitions.

5.2 Experiences combining UML and Ruby

A main goal of the experiment was to draw experiences on combining a language for prototyping or experimentation with a language for specification. The last section tried to answer if Ruby can represent the prototyping language of such a combination, the findings was that Ruby has a lot of features typically found in other prototyping languages. This section will focus on the combination of UML and Ruby. To restate the principle of limited reduction [Mathiassen96]:

- Relying on an analytical mode of operation to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures.
- Relying on an experimental mode of operation to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

The interesting part of using a combination of tools is how it performs in regards to the principle. The question is: How will use of one tool countermeasure the weaknesses of the other and vice versa. The section begins with comparing Ruby and UML as basis for user communication and then how the languages were used when coping with change to the requirements.

5.2.1 User communication

During the experiment both Ruby and UML was used in user communication. This section will compare how the two languages were used in different situations. The questions to be answered are:

- Did UML generation of the prototype help in user communication?
- Did the users understand action specifications in Ruby?
- Which language was easier to use in communication?

As described in section 4.5 the users had varying experience with UML. Therefore is unfair to compare UML and Ruby on the basis of the users' understanding of them. Rather I will focus on how they understood the languages considering their background with them.

UML in user communication

UML was used from the very beginning of the experiment implying that an analytical mode of operation was the first to be applied. The first models produced were business process models using the activity diagram notation and use case diagrams augmented with textual step logic. These models were developed to learn about the problem domain and define the scope of the application. The models were then used as basis when developing a business concept model class diagram. It is not in the scope of this thesis to analyse the efficiency of such models. However the users expressed great interest in these models, actively making suggestions of amendments and changes. For example one user with database development background immediately began explaining how the data flow would be upon seeing the activity diagram in Figure 5.2-1. Object oriented design literature often advice careful use of such process oriented models because they can inspire developers to think about implementation to early [Richter99], which is just what this user did. Also note that the ability to draft save an invalid timesheet for later amendments is not included in the diagram, and was overseen both by me and the users until much later in the prototype development (see 5.2.2).

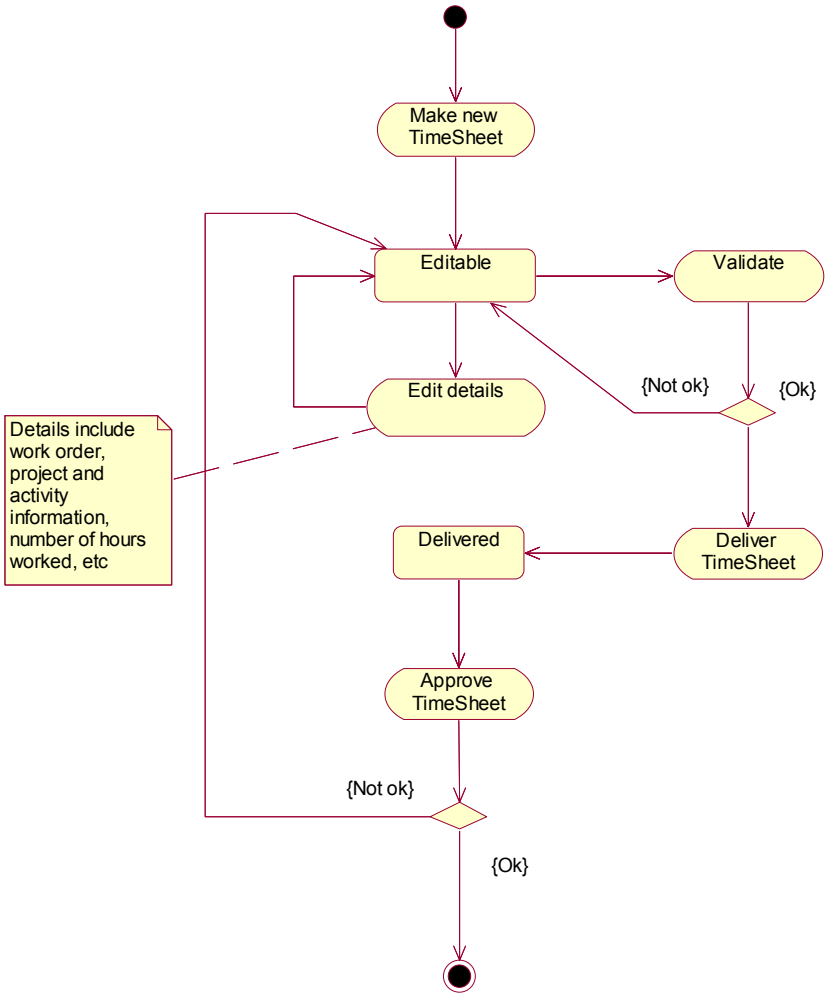


Figure 5.2-1 Example of early activity diagram

A review of the business concept model (Figure 5.2-2) was held when the scope of the case was decided upon. Questions from some users like: "what does that diamond mean?" (One user pointing to the composition relationship between Resource and TaskList) and "is a manager also a resource?" revealed some uncertainty about UML notation. But overall the users expressed that they understood the UML class diagram notation, except for extensions in the form of stereotypes I had introduced.

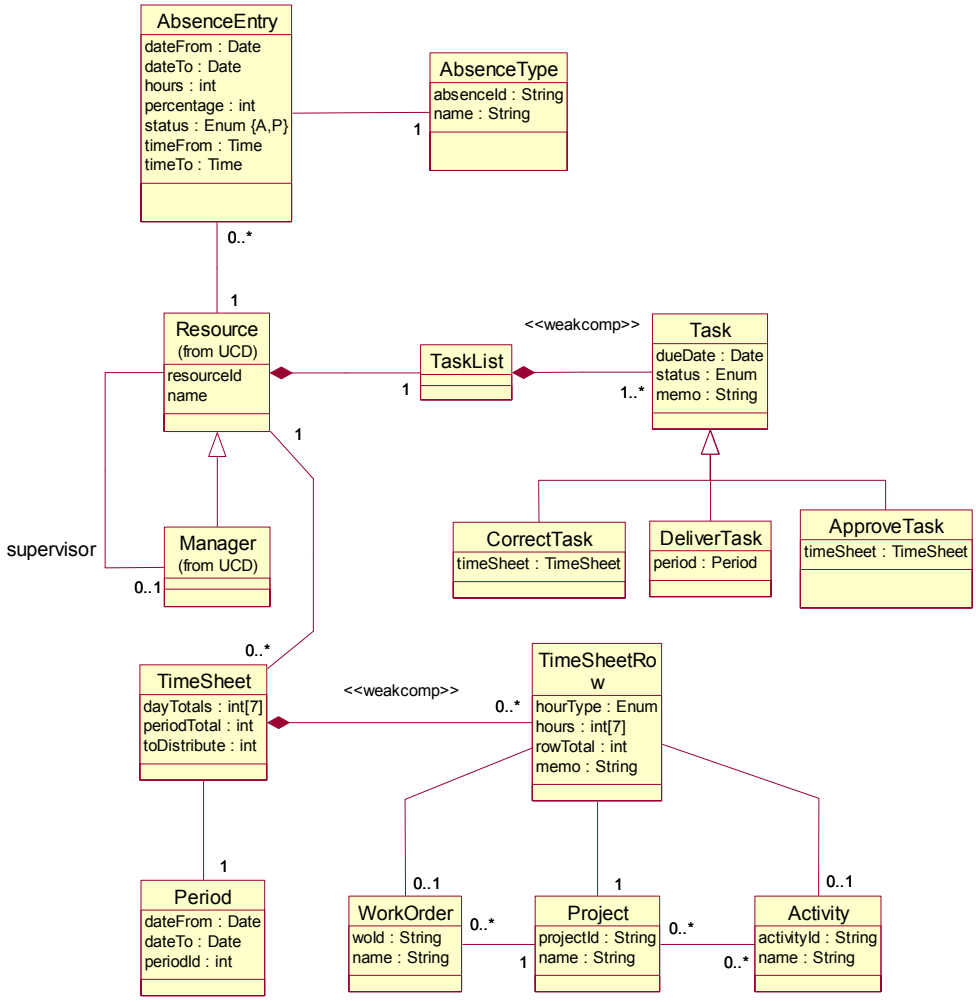


Figure 5.2-2 Business concept model

The discussion also revealed errors and amendments to the model, such as "An activity is always tied to at least one project", "you should make the association between a resource and the timesheets a qualified association based on period, because the resource has only one timesheet per period" and "at some point the task list is empty". Here also, some users with developer background started talking about implementation issues very early with statements like "this class here has to have operations that validates if the combinations of instances of these are legal". This might be an indication of inexperience separating analysis and implementation models. Other users with more of an analysis background were more interested in the overall picture, what classes represented and what relations they had with others. These users often related the concepts from

the model with real life concepts with questions like "if I have a paper that is my TimeSheet, the TimeSheetRows is each job description on that paper, right?"

Conversations and reviews like this with a UML class diagram as basis was held throughout the experiment. Using the model it was easier for me to pin down the area of attention and for the users to have the same focus.

Ruby in user communication

Ruby did never gain the same role as the UML diagrams in user communication. Primarily I used it to explain what was going on inside the operations specified in the UML diagrams. Users having developer background, seemed to be able to read what was being done quite easily. However some specialized concepts of Ruby, like code blocks and mixin modules, required explanation. Input from these users was good when trying to increase readability of the Ruby code. Sometimes code was written too dense resulting in reduced readability. This code could be extended with more keywords or written in another way since Ruby allows many ways of doing the same thing.

When using Ruby in combination with a generated class diagram, the RDoc tool was used to browse the code. RDoc generates an html based, Smalltalk inspired, code browser that can be used through a web browser (Figure 5.2-3). Combining this with the convention of one class diagram per Ruby file, the relevant documentation for a diagram was easy to find in RDoc. This way we could easily click to see the method of an operation of interest from the class diagram.

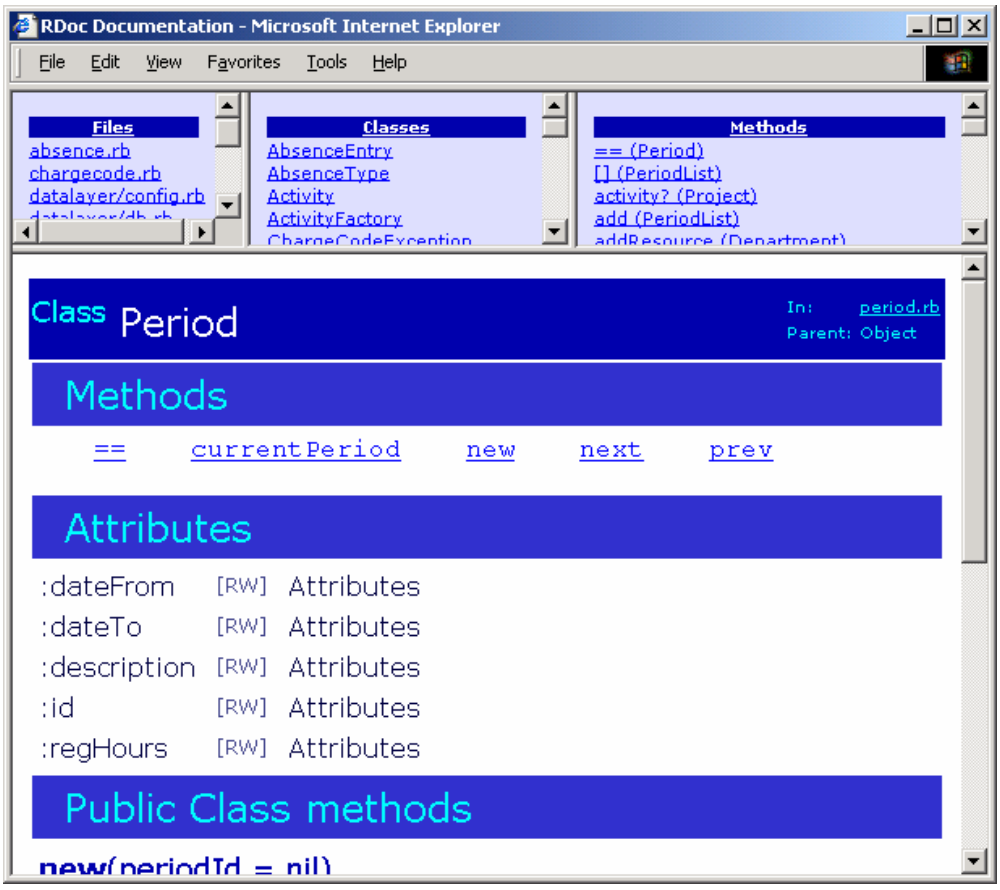


Figure 5.2-3 RDoc screen

Using the RDoc browser also made it easier to follow call sequences of operations. However the call sequence can not be seen at a glance, you have to go into the method of each operation to see what other operations are called, find that, repeat the process, and retrace back to the beginning. If a specific scenario was interesting, UML sequence or activity diagrams were used to clarify the situation. The problem with such diagrams is that they can become rather cluttered and complex, and that you have to introduce several similar diagrams to show all possible solutions. The Ruby code shows all situations, but has the problem with being distributed throughout the code. Listing 5.2-1 shows a set of related operations in Ruby, whose UML sequence diagrams would be complex.

```
# File time.rb TimSheetRow class, line 268
  def setCcAttributes(attr)
    val = ChargeCodeValidator.new(self)
    if (attr.kind_of?(Integer))
      @chargecode = val.getCcById(attr)
    elsif (attr.kind_of?(Array))
      @chargecode = val.getCcByValues(attr[0],attr[1],attr[2])
    end
    findApprover if(@chargecode.valid? and hourType)
  end
# File chargecode.rb ChargeCodeValidator class, line 115
  def getCcByValues(pro=nil,wo=nil,act=nil)
    begin
      validate(pro,wo,act)
    rescue ChargeCodeException
      returnTempCc(pro,wo,act)
    else
      returnCc(pro,wo,act)
    end
  end
# File chargecode.rb, line 128
  def getCcById(id)
    DB.connect do |db|
    if(id == 0)
      row = db.get_tmp_cc_details(@ts_row)
      return returnTempCc(row[0],row[1],row[2]) if row
    else
      return ValidChargeCode.new(id)
    end
    end
  end
# File time.rb TimeSheetRow class, line 200
  def findApprover
    delete_all_observers
    @approver = chargecode.project.findApprover(self)
    add_observer(@approver.tasklist) if @approver
  end
```

Listing 5.2-1 Browsing related operations

The users with analysis background did not seem to understand Ruby code without thorough explanation. “I do not understand programming languages” or “I have never programmed before so I do not understand this” clearly states that they thought Ruby code looked like any other programming language code they had seen, and that they were not eager to learn to read it either.

Combined use

The way UML and Ruby was used in communication with the users somehow reveals the nature of each of the languages. UML hides details to focus on the area of attention or the bigger picture. Ruby on the other hand lays the details bare; the details overshadow the bigger picture even if it is there. The users were more interested in the Ruby source when the RDoc, which also does shield implementation and details until asked for, was used to present it.

On the other hand, when details are asked for, when we are interested in exactly what should happen to the elements of the system, when for example the details of a timesheet is changed, Ruby shows every action involved, and allows more completeness than UML sequence, collaboration or activity diagrams. By the use of unit tests Ruby also gives examples of both successful scenarios and erroneous scenarios and how these are handled.

There have to my knowledge not been studies on how unit tests also can be used for documentation and specification, but it was interesting to see that test cases could be used in this manner. There are plans to incorporate test cases, used as documentation for classes and operations, into the next version of RDoc. By combining Ruby prototype source code with corresponding UML class diagrams, we get both the overview of the structure and the details of actions.

5.2.2 Coping with change

As mentioned in section 5.1.2, dynamic typing makes some changes easier to do. The example there was of a structural change in inheritance hierarchies. Generally I found it easy to apply changes to the prototype on a small scale in Ruby. It was easy to express new ideas, because of the rich syntax and semantics of the language, and to check implications of them running the test cases. But faced with larger structural changes I found I was forced to make a translation to UML to gain overview of the implications.

One example of this came because of a misinterpretation of information from one of the users. Originally my understanding of what one user communicated, was that in ABW the manager of a resource is the one responsible for approving that resource's timesheets. Further that a legal combination of project, work order and activity had to be entered for each row in the timesheet. In the prototype, the approver of the timesheet was therefore assigned to be the manager of the resource, if that resource had a manager. Test cases were also written on this assumption. The relevant structure is expressed in the class diagram below (Figure 5.2-4).

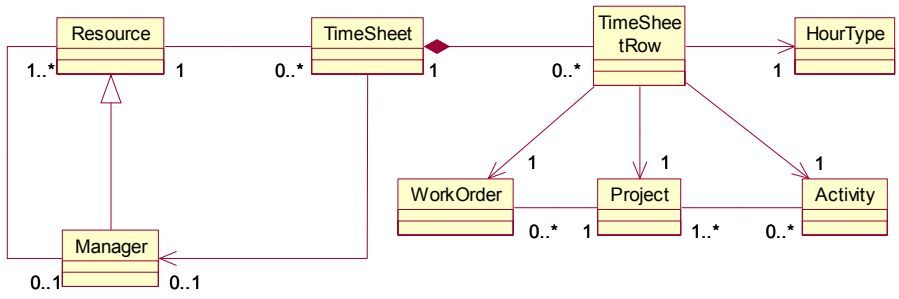


Figure 5.2-4 Class diagram of original structure

The structure was translated to Ruby, and methods for finding and setting the approver association was based on the association between a resource and the manager. Methods checking legal projects were introduced at the TimeSheetRow class and delegated responsibility of checking work order and activity combinations to the project class.

Other classes such as the TaskList and the database layer used the TimeSheet class and became coupled to this structure, using the approver association. At a later time when I verified the structure in a conversation with the users the misinterpretation was discovered. The project was supposed to have information about what kind of hours to approve and who the approver is. The approver could be the project manager, the work order responsible or the manager of the resource. Further normal-, overtime-, both or none hours had to be approved.

This meant that each TimeSheetRow could have different approvers. At first I tried to implement the changes in Ruby alone, the plan being to translate into UML and verify again together with the users. Before that more changes was introduced because the users wanted to be able to draft save invalid time sheets for later amendments. I then decided that it was too difficult to gain enough overview in Ruby alone to make such structural changes with the dependencies already there. I made a translation of the structure that I had in the Ruby prototype and developed a new one, which moved the approver association to TimeSheetRow, and introduced the concept of a charge code. The charge code held information about the project, work order and activity combination and performed validation it. The result of this is shown in Figure 5.2-5

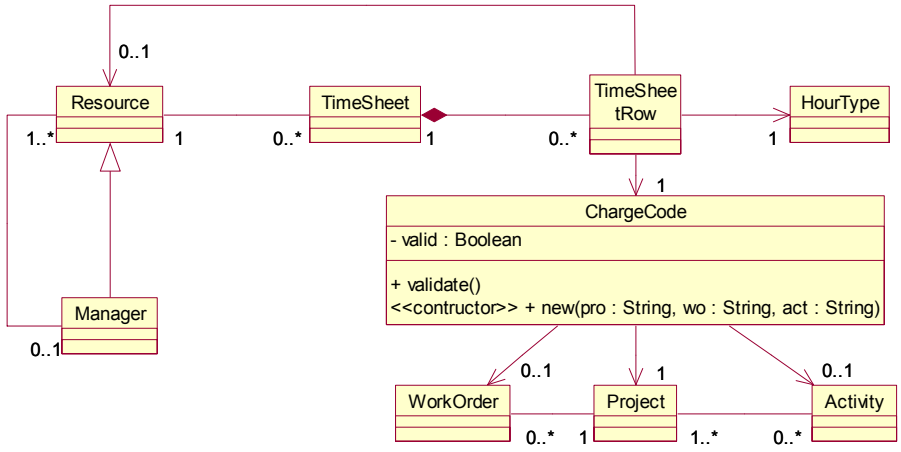


Figure 5.2-5 Class diagram of changed structure

When I returned to Ruby some methods had to be rewritten and new introduced, as well as a complete rewrite of the test cases. The ChargeCode class began to be too complex and was specialized into a ValidChargeCode and a TempChargeCode. A ChargeCodeValidator class was introduced that could validate input in context of a row and make and return a proper type of ChargeCode. This was easier to do directly in Ruby since this change did not have too many dependencies yet. Translated back into UML the final structure (without attributes and operations) is shown in Figure 5.2-6.

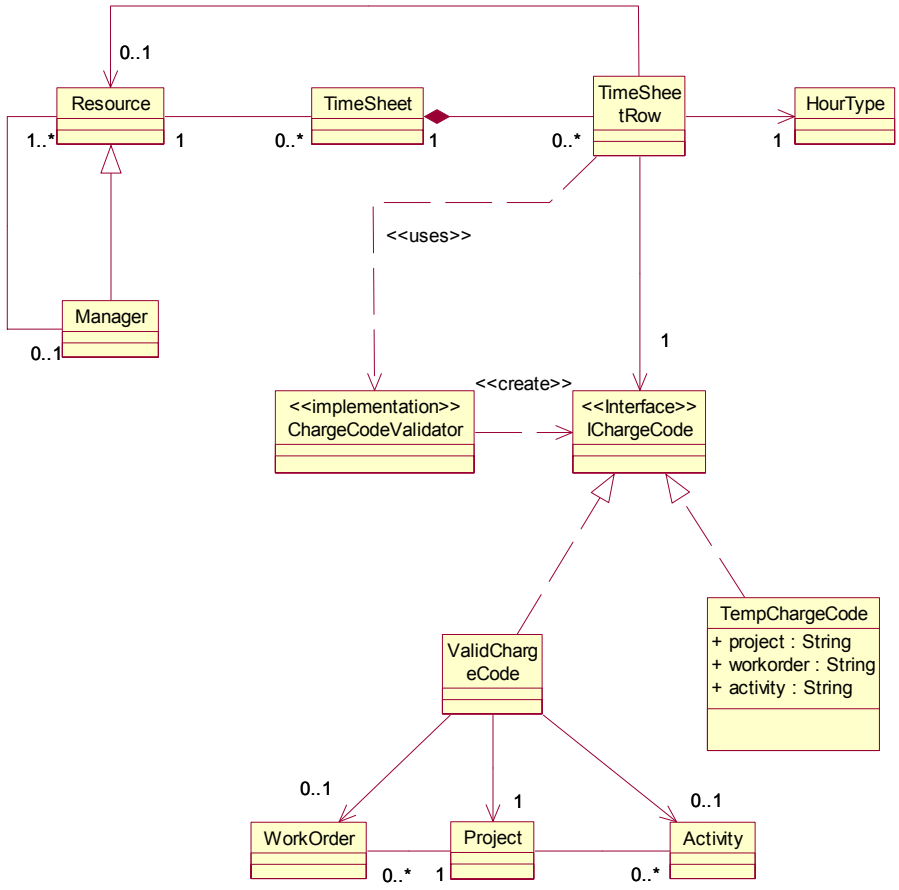


Figure 5.2-6 Class diagram of final structure

As shown in the example above, it can be hard to apply larger changes, especially relating to structure, in Ruby alone. I experienced what I would describe as a "spaghetti code" syndrome when trying to incorporate large changes directly in the prototype. Early assumptions that turn out to be wrong can be a source of larger changes, and this tend to happen more often in the learning or prototyping phase of development. This above example shows that when trying out ideas to meet uncertainty the complexity will rise as more is learned about the problem domain. Having the possibility to go to UML, suppressing details and regaining overview is a good countermeasure to this problem.

5.2.3 Summary

UML was used to a greater extent than Ruby in user communication. It was easier to gain overview and pin down the area of attention using a graphical model in UML than a textual model in Ruby. The reason for this is assumed to be the information shielding capabilities of UML not only its graphical representation. Users without development experience did not easily understand Ruby source code without thorough explanation. On the other hand, users familiar with other programming languages were able to quickly understand details of operations written in Ruby.

By using information shielding tools like RDoc, Ruby and UML was used together in some situations. Tracing call sequences and action details in RDoc at the same time as having the overview in form of UML class diagrams.

As mentioned in section 5.1.4 change in the small was easy to do in Ruby, on the other hand larger structural changes made the prototype hard to maintain. Using Ruby to try out the structure and requirements revealed tacit and overlooked requirements early. UML was superior to Ruby when applying larger structural changes. Literature states that evolutionary development and prototyping has a problem of instability because of many changes applied to the prototype reducing maintainability [Boehm84], [Sommerville95 ch8]. I found that by using UML to generate class diagrams from the prototype somewhat countermeasures this in the form of regained overview and maintainability compared to continuing a more evolutionary development using only Ruby.

5.3 Experiences translating concepts back and forth

This section will discuss the Ruby-UML-, and UML-Ruby translation. The following questions will be in focus here. In what way will the translations influence a development process? What are the weaknesses inherent in the translation model that was developed and what are the benefits?

5.3.1 Influence of the translations

As found, Ruby is a good prototyping language. Code can be written and executed quickly due to its dense style, dynamic typing, flexibility, reusability and interpreted execution. This also makes it a good language for experimenting with different solutions and designs. Interpretation means that there is no compilation phase and arbitrary large parts of the prototype or application can be tested on its own. What happens when UML is coupled with a language that has these capabilities?

The answer depends on how Ruby and UML are used in the development process. An issue with many modern CASE tools is the notion of round trip engineering. Round trip engineering is the seamless integration between design diagrams and source code. The programmer generates code from a design diagram, changes that code in a separate development environment, and then recreates the possibly altered diagrams back from the source code. Round trip engineering is also made an issue by many object-oriented development processes' emphasis on iterative development [Booch94], [Booch99], [Reenskaug96]. In many ways this is what the Ruby / UML combination is trying to achieve. But there is a problem with the combination and how it can support round trip engineering.

The problem is that ideally the Ruby prototype code, and the UML representation of it, should always be in sync with each other. This is sometimes called simultaneous round trip engineering. Without this, the source code and the UML diagrams will at some point not describe the same application. The smallest change in either Ruby or UML should be represented in the other language. With proper tool support this should be possible. TogetherSoft's Together ControlCenter (www.togethersoft.com), is one example of a highly sophisticated tool with heavy emphasis on round trip engineering support between different programming languages and UML.

The problem with this for a dynamically typed language is that some information in a prototype can only be deduced based on sample input data. In other words some of the translation has do be done at runtime. In a test-driven process based on test cases, keeping a perfectly synced source code / UML representation of the application effectively introduces a translation phase, which has much in common with the overhead compilation introduces to the code – test – correct

development cycle. The translation phase will even out one of the biggest advantages that interpreted languages have over compiled languages when used in prototyping, the quick turnover.

When developing the example prototype, I did not have access to any tools automating the translation process, every translation was done manually. The first translation done was from the UML-Ruby translation of a conceptual model. The model was extended with actions and tested. As discussed in the previous sections changes in the small and experiments were done in Ruby and larger changes, as well as discussions with the users, were what forced translations into UML. Obviously simultaneous round trip engineering, as described above, was not the case. To find the reason for this we can again look to the principle of limited reduction. In this thesis UML is used as a language whose strength is to describe the results of analytical behaviour, and Ruby a language that facilitate experimental behaviour. The driving force behind the translations is the behaviour shift from analytical to an experimental one, and the other way around.

The example scenario provided in 5.2.2, is an example of how a translation was first triggered when experimenting with Ruby revealed that there could have been a misunderstanding about the requirements, and a shift to analytical behaviour occurred. Translations back to Ruby was forced when the structural changes applied in UML, behaving in an analytical way, was to be tried out and tested, thus shifting focus again to an experimental behaviour. The effect is that the use of each of the languages is affected by changes in the developer's behaviour, and as each mode of operation requires the other to countermeasure the effects, translations occur naturally.

Translational tool support is needed because, as manual translation is slow and also error prone, the developer can become reluctant to shift behaviour, even if it is required. As tool support was not available during the experiment, it may be that a shift between analytical and experimental behaviour did not occur as often as it would if a tool was available. Because I was able do develop and test quickly in Ruby, I sometimes found myself trying to apply changes that really should have been done in an analytical way. This way I progressed too far with Ruby, experimenting with the changes, instead of analysing the implications of the changes. The result was that much of what had been done in these situations had to be thrown away, or done in another way, when a translation was performed and better solutions were found.

The implication to the process is that a translation might seem to slow down progress, and the developer goes too far in an experimental mode of operation because of this illusion. Having tool support for the translation, and perhaps rules and guidelines for in what situations a translation should be performed, can be an alternative to simultaneous round trip engineering. This way fast development turnaround in Ruby can be upheld, and at the same time inspire translations when a shift in behaviour is needed.

5.3.2 Loss of information, round trip or one way?

Another problem is related to loss of information in the translation, either from UML to Ruby or from Ruby to UML. Only the combination of the two languages will form a complete picture of the whole application. Each language on its own will not contain all the information in the other language. The two languages are tools meant to solve different challenges in software development.

Starting with the translation from UML to Ruby, the information loss is related to type information. Since in UML the type of attributes, operation parameters, return values and associations are really constraints on the application, these restrictions are not transferred to the corresponding Ruby source code. It may be that the application never violates the restrictions, but there is no way to explicitly uphold them without changing or extending Ruby itself. This is because type checking is

performed at runtime, as explained in 5.1.2. As stated in 3.2, type information can be included as comments in the Ruby code, but this only helps understanding the code. Section 5.4.2 discusses schemes for extending Ruby to support explicit typing of the most important elements, without changing the interpreter itself.

Another problem when developing the translation model itself, was how UML concepts would be represented in a programming language like Ruby. The semantics of for example an association in the UML is much richer than the solution in the translation. The UML 1.4 specification [OMG00], defines an association as an instance of the metaclass Association, having two or more ordered AssociationEnds, which in turn are connected to a Classifier (Class). The ends have attributes telling if it is navigable, aggregated (tightly connected), what the scope of the target is, the multiplicity of the target, if it is changeable, the visibility and a collection of optional ordered additional attributes. In addition, an association can be specialized by other associations, or be a full-blown class (AssociationClass). The solution in the translation model is one, or a combination of several, special object reference(s). This is because it is the most common solution for implementing an association, given in a specification model, in a programming language. It is also the most common solution made by code generators in UML tools. Very detailed associations, having features like explicit multiplicity or several association ends, has to be implemented as classes that uphold the same constraints. A problem with this solution is that it can be hard to develop heuristics on how to translate such classes and references back and forth between Ruby and UML automatically. This is easier to do manually, especially if the programmer and translator is the same person. Another example is the composition relationship, an association with specialized semantics, which required an extension of Ruby for its constraints to be supported.

Ruby's job in the combination except for being a prototyping language is to specify actions and executable statements. The translation from Ruby back to UML does not keep the action specifications. The current version of UML (1.4) does not contain enough to specify the same as Ruby actions. The metamodel of UML has a package named Common Behaviour that does define Actions; the metamodel is shown in Figure 5.3-1, copied from [OMG00 (p150)].

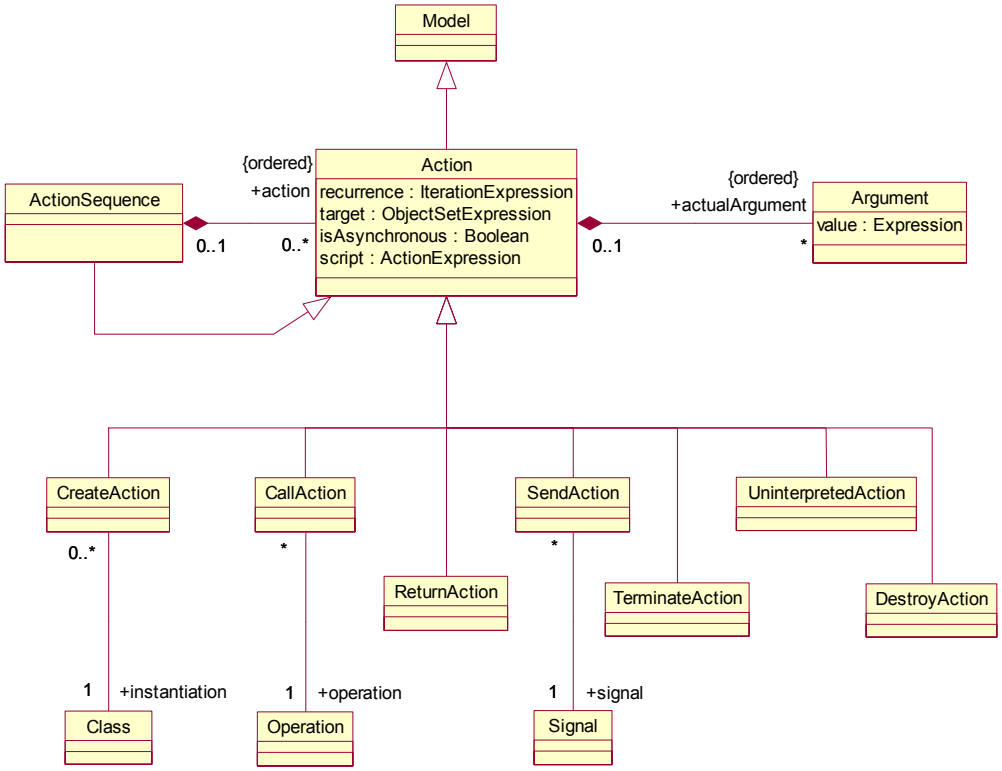


Figure 5.3-1 Common Behavior – Actions

These metaclasses provides the infrastructure in UML sequence diagrams, state machine, use cases and collaborations. But they are still not enough to represent the diversity of different actions found in Ruby. One will very seldom see the use of these metaclasses to describe actions. The current work on UML 2.0 includes an extension / replacement of the current action metamodel in UML, an effort to rigorously define precise semantics for different types of actions [ActionSemantics]. A classification of actions to be supported is:

- Read and write actions: These actions allows for reading or changing information of a class. One can read, or set the values of-, variables or attributes, links from one object to another can be destroyed, created or read (navigated). Creating or destroying an object instance is also an example of a write action.

- Computation actions: Computation actions take input values, do some kind of mathematical evaluation on them, and produce output values. Finding the square root of a number is an example of a computation action.

- Messaging actions: These actions exchange messages between objects. One kind of message action is calling an operation or method. An object can also send a message to itself, i.e. calling one of its own operations.

- Composite actions: As its name suggests a composite action is a way of putting together other simpler actions to perform more complex tasks. One example of a composite action

is the conditional action, which based on a conditional test decides if another set of actions should be executed. This is typically implemented with if – then – else clauses. Another example is the loop action, which can execute a series of other actions as long as a condition holds, i.e. a while or for loop. A composite action can itself contain other composite actions.

- Collection actions: These actions operate on some kind of collection of elements. An example is an iterate action that applies the same action on all elements of a collection, such as raising the salary of all employees in a department with 100$.

- Exception actions: A condition that rises as part of an action, often because an error occurred such as an attempt to pop an empty stack or divide a number by zero.

The idea of the Action Semantics RFP is to first define the semantics for these types of actions, so that later mappings can be made to different programming languages. The main goal apart from an opportunity to specify precise actions in UML is a higher degree of complete code generation from the language to programming languages. It will also become an important tool in the process of defining executable UML. With the introduction of executable UML, the language can solve one of the challenges that Ruby does in the UML / Ruby combination, namely the ability to test and debug models at an early stage in the development. The question is if it proves to be as good as Ruby or any other prototyping language when it comes to experimentation. Pure execution abilities may not be enough.

The actions of Ruby may well be mapped to the Action Semantics thus bringing a UML / Ruby combination closer to round trip engineering without information loss. The challenges will also here be about the struggle between dynamic and static typing.

The solution I used to minimize information loss due to the inability to express actions precisely in UML was to include the methods (actions) of operations in the documentation for the operations. If operations where removed in UML so were the methods in Ruby, if a new operation was added, the Ruby code making up the method was added after translation back into Ruby. All the other actions survived the transition to UML and back to Ruby. This was only a way to keep the information, clearly better solutions has to be found. In an imaginary future tool, automatic translation of Ruby actions into UML action specifications, if they become rigorous enough, must be the best solution.

5.3.3 Summary

The benefit of the combination of UML and Ruby is that they complement each other to describe a more complete specification of a system than each would have alone. Ruby has the ability to concisely specify actions and give a hint about operation implementation as well as being an experimentation and prototyping language, UML define structure and perform as a tool for analytical behaviour.

There is a problem with simultaneous round trip engineering in that it introduces a translation phase affecting Ruby's quick code – test – correct development cycle. The translations between the two languages seem to be triggered by change in the developer's mode of operation from an experimental one to an analytical one or from an analytical one to an experimental one. Without translational tool support and guidelines there may be a problem with the developer staying in one mode of operation for too long, using the wrong tool to solve a certain problem.

Because of differences in the two languages information loss occurs both when translating to and from one language. Steps can be taken to minimize information loss.

5.4 As one, or separate but together?

In the last section I discussed problems that occurred because of using the combination in the development process. One of the goals of this thesis was to find out how close these two, sometimes very different, languages could be brought together. Was the combination a new language, in the sense that it tightly coupled took the best from each of the languages and formed a new better hybrid language? Or was the result that each language has strengths, the sum of problems to solve bringing them together? In other words should the two languages be treated as one, or separate but together? I will also discuss how Ruby can be extended to become a better companion for UML.

5.4.1 Problems keeping them apart

I have already discussed many differences between the languages both in the translation and in the prior section. It is important to know why these exist.

The main reason is that each language originates from two different disciplines in software development. Ruby has evolved as a programmer's tool to solve everyday tasks quickly and with minimized effort. Later, its features have inspired users to use it in other ways, for prototyping and also as the main language for developing larger and larger systems. The way it is most often used, the focus is still on problem solving and less on specification, modelling, and as a basis of intra developer- and developer - user communication. The prototype produced using the language is the communication basis, not the language itself.

UML comes from a totally different paradigm, the result of a controlled adoption of the best of many other modelling languages and development processes. It is used as a tool to easier understand the reality, by simplifying it in models. By the help of these models we communicate and learn about the reality. The UML is mainly a common notation for modelling.

Recently Extreme Programming has got much attention, with heavy focus on experimentation, and controlled efforts to reduce the problems usually affiliated with evolutionary development. It is out of the scope of this thesis to discuss what methodologies and methods may be the best, this is a question that gets a lot of attention by others. It will suffice to say that languages like Ruby and test-driven development are important tools in Extreme Programming. UML is also used in Extreme Programming, as hybrid processes like Extreme Modelling and Agile Modelling with greater focus on modelling have evolved. UML has traditionally been used the most in model driven processes like RUP, or so called heavy weight methodologies, which high focus on documentation and specification.

Where these two languages come from, and how they traditionally are used, explain many of the differences between them. Trying to use either of the languages in other ways, like Ruby for modelling, or UML for experimentation, will naturally introduce challenges. This is because they are not necessarily designed to be used in this way. Trying to use Ruby to model a large system, for example, will be difficult because it is in ways too flexible, and does not have the ability to describe a whole system on its own. On the other hand using UML to experiment with a solution can be a problem because it does not have that flexibility. Trying to bring the two together to form a new language revealed tensions, from which I draw the conclusion that you are probably not meant to succeed in doing it without changing both, or one of the languages, in a way so that their strengths

disappear. Instead, to find the similarities, and bringing them closer, but not completely together, using each language to solve the problems they are good at, seemed to be the solution. In the next subsection I will explain what compromises I did to bring the languages closer.

5.4.2 Compromises and changes made to Ruby

Apart from the fact that Ruby is not able to describe as much as UML, simply because UML has a vocabulary to depict many other aspects of a system apart from only its classes and objects, the biggest difference is obviously dynamic versus static typing. In a specification static type restrictions are important constraints and give the reader more information than non-explicit typing.

There are three possible solutions to this problem between Ruby and UML. The first is to remove typing from UML, thereby changing the very foundation of the language. For example you would not be able to draw an association from one classifier to another because this is effectively static typing. UML is a graphical language and I have no solution for how one might draw dynamic typing. When you think of dynamic typing you think about change, which is difficult to depict using static pictures. The second solution is to explicitly type everything in Ruby, which is easier. But dynamic typing is there for a reason also. Taking it away would take away the possibility to exploit dynamicity for better flexibility and forcing earlier perhaps wrong decisions which are bad for experimentation (see discussion 5.1.2). The third, and final, solution is to make a compromise; keeping UML typed and introduce explicit typing on some aspects of Ruby (I avoid saying static typing here because this requires a compilation phase). The same has been done for other dynamically typed languages [Graver90], but the inspiration to change typing has often been to improve performance.

Following the third solution I realized that the most important typing restriction is on associations. This is because if an error happens here, meaning an association gets assigned an object of illegal type; it will not be discovered before it is used. This association (reference in Ruby) may then be passed to other objects as parameters again assigning them to associations and passing them on. Suddenly it is used somewhere by a totally different object than the one that should have discovered the error, and a name errors occur. This error can be hard to trace back to the origin of the problem. Of course this is also true for compositions since they are specialized associations. By introducing explicit typing of associations the difference between an association and an instance variable will be more apparent and an important restriction on the application will be upheld. The same argument can of course be made on instance variables / attributes and local variables in methods. The difference is that, in the normal case, the problem will often be discovered locally if violated, so I chose not to type them.

Since there is no compilation phase, the check has to be done at run time. The difference is that the error will be discovered right as the violation happens. As a basis I used the same extension that was developed for the composition in the translation, and the inspiration was the techniques used in a DBC for Python module explained in [Ploesch98]. It was a matter of finding syntax for specifying associations and types explicitly and extending the wrapper for every public method to check if a violation had happened. Compositions were also checked in this way in addition to the other checks. I decided on the following syntax: *assoc :symbol, class, [multiplicity]*. Assoc is really a method call that ties an instance variable symbol to the specified class. The check was really simple and put into the Composition module, now renamed Typecheck. It creates a local hash that keeps the symbol – class mapping of every association and composition. The implementation of this and the check itself is given in Listing 5.4-1.

```
def AssociationViolation < StandardError
end

#module Typecheck
def assoc(sym,type,*refs)
  @assocs[sym.id2name] = type
  if refs
    @mplicity[sym.id2name] = refs
  endif
end

def checktype(sym)
  if !instance_eval("{#sym.id2name}.kind_of? @assocs[sym.id2name]")
    raise AssociationViolation("Illegal type for " + sym.id2name)
  end
end

def checkmplicity(sym)
  if mplicity[sym.id2name]
    object = instance_eval("{#sym.id2name}")
    if !object && mplicity[sym.id2name][0] > 0
      err = true
    endif
    if object.kind_of? Enumerable
      error = true if object.length < mplicity[sym.id2name][0]
      if mplicity[sym.id2name].length > 1
        if mplicity[sym.id2name][1].kind_of? Integer
          err=true if object.length > mplicity[sym.id2name][1].length
        endif
      endif
    endif
  endif
  raise AssociationViolation("Illegal multiplicity:" + sym.id2name)
end



def checkall … end #iterates over all associations and compositions making
the necessary checks.
```

Listing 5.4-1 Implementation of type checking

To use this, client classes must include the module Typecheck, define instance variables and then declare them to be associations. Included on top level there is an option to turn this check on and off, because they can slow down the execution of a program quite dramatically. The extension has been tested on some small cases, but not on the example application. The nice thing is that now it is very easy to see what instance variable references are associations or compositions and what are attributes.

As discussed before another problem with dynamic typing arises when the translation is to be performed because type information is only available at runtime. If there were some way to get a hint about the type of elements in Ruby without studying the code manually it would be a step in the direction of automatic deduction of this information. Of course exact type information is not possible for this would require the application to be tested with every possible input value. On the other hand if we use test cases and view them as examples of what types will exist in the system we can further extend Ruby to keep a track of the types and log them during execution. This technique can also be found in JIT compilers for languages like Self, where the information about what types

exist in the system is used to guess what types will exists later. Combine this with the static information collected with RDoc or similar tools, translation become more complete from Ruby to UML. The idea here is to keep a simple metamodel of the classes, its attributes and operations and the operation's parameters during runtime, keeping the current type information and writing the result to a file when finished. The metamodel does not need to include other information that is made available by RDoc. The metamodel used is depicted below in Figure 5.4-1

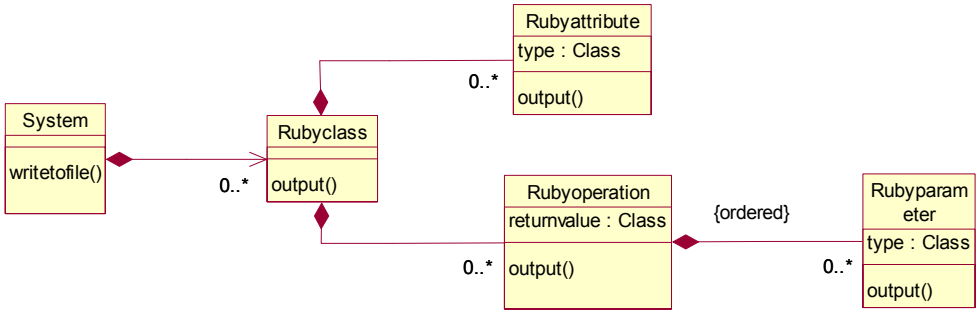


Figure 5.4-1 Metamodel for type deduction

When the type of an element changes the inheritance hierarchy has to be traversed until a common type is found, and that new type will be stored instead. Using reflection this is easy to do, a recursive algorithm that does this, and a method that triggers the inheritance hierarch traversal, is shown in Listing 5.4-2.

```
# Trigger
def logtype(type)
  unless @type
    @type = commonsuper(@type,type)
  end
end

# Traversal
def commonsuper(a,b)
  if a == b
    return a
  else if a.superclass == b
    return b
  else if a == b.superclass
    return a
  else
    return commonsuper(a.superklass, b.superklass)
  end
end
```

Listing 5.4-2 Traversing the inheritance hierarchy

A situation that can happen in Ruby is that the programmer deliberately designs a method to return values of two different types, not in the same type hierarchy. This may be to make a generic method that should return a known value type based on the input. These situations are rare, but one example can be the *eval* methods from the Ruby library, that returns the return value of the ruby string passed to the method. In such cases the algorithm above would return Object as type.

What you would want in UML, in such situations, is a series of similar operations with different signatures. The same applies to Ruby methods with variable length argument lists, and optional arguments. Another example of something that is difficult to translate is the use of code blocks and proc objects as parameters. This is because these features have special behaviour when in comes to the use of scope, remember the jukebox composition in chapter 3 for an example of this. This only shows that there will be differences, even if compromises are made. The natural thing would be to avoid the use of the most exotic Ruby features if the goal is an action specification. Of course these can have great value when experimenting, but like Perl there are many ways to do the same thing in Ruby and in more UML friendly ways as such.

By representing this type information in xml it can probably be coupled with the static information from RDoc also in xml. In addition when we have an easy way to tell the difference between attributes, associations and compositions, translations between the languages are easier to do.

5.4.3 Summary

My first assessment of how close the two languages could be combined into a new better hybrid language failed because of the tension and differences between the languages. This tension is mainly there because the two languages are designed for two different areas of application in software development. Instead of removing features from the languages to be able to more easily combine them or make them more similar, I arrived at the conclusion that it is better to embrace the differences as examples of when to apply each of the tools, while making compromises to inspire combined use of them. In this way each of the languages shortcomings will inspire the use of the other language. The effect is that, not necessarily consciously, the use of mixed tools will force a mixed process.

6 CONCLUSION

The main goal of the thesis has been to explore the idea of mixing two languages, one typically used for specification and object-oriented modelling, the Unified Modelling Language (UML), and one for rapid development of prototypes, Ruby. The goal has been to elaborate on the possibility and implications of such a mixed language approach in the early stages of software development. The motivation for the idea has been the fact that a mixed approach of specifying and prototyping has proven to be better than a pure evolutionary or analytical approach. The main question was formulated like this:

- To what extent is it possible to combine languages for specification and prototyping? And what are the implications of doing so?

The approach to answer this question was to develop a functional (one-to-one) mapping between UML class diagrams and Ruby, a translation model. This translation model was then used in a real life setting to develop the design of a component supporting client independent hour and absence registration. By following this approach the goal was to examine the feasibility of such a mixed language approach and draw conclusions based on experiences from using it in practice.

There are always sides about a research effort that rise uncertainty about the feasibility of the result. It can be weaknesses in the research method used or in the way the experiment is carried out. These are also evident in this thesis. The two next paragraphs outline the problematic issues.

First there is no triangulation in the methods used. This means that the experiences from the qualitative work done are not backed up or checked by quantitative methods. My conclusions are based on my own experiences. Although one strives to achieve objectiveness when doing research, my interpretation of the experiences will in some respect be subjective. Important issues may not have been touched upon, and less important issues may have received too much attention. A good way to verify the conclusions I have made would be to perform a similar study on a group of developers using quantitative research methods as mentioned in 1.3.

Second it the choice of case and problem domain is specific. In this case it was reasonably sized and there existed applications already that did the same thing. In real life, prototyping would probably not have been applied by developers in this organization because the problem domain is well known. The users or people serving as domain experts in the experiment are not real end users. They are employees of a company that deliver similar solutions to end users. The implication of this is that may have had different reactions than real end users. It is difficult to say if the mixture of UML and Ruby would perform better or worse in other scenarios.

In answering the general problem of the thesis, the first obvious conclusion is that only a subset of UML can be mapped to Ruby. UML covers many areas of object oriented development that Ruby can not hope to cover. Even if both Ruby and UML class diagrams are object-oriented, and have many common elements such as the notion of classes, objects and inheritance, there are also here profound differences stemming from what the languages are designed to do. They come from two different worlds in terms of how they where developed, what methodologies and processes they usually are used in and the simple fact that one is a programming language and the other is a modelling language. How much of these differences that could be overcome?

By developing the translation, every element was treated step by step. The conclusions from this process are that the common entities were straightforward to map. This included class, attributes and operations. The problematic differences were a clash in Ruby's dynamic typing and UML's static typing. UML allows for unspecified type information in attributes and operation signatures, so this clash did not affect them. Using Ruby's introspective capabilities type information can still be obtained, although a problem with completeness rises here because the information is dependent on sample input data. Some kinds of relationships in UML like associations and compositions were not possible to map directly to Ruby, and the language had to be extended to support the semantics of these. These extensions were possible to develop without changing the Ruby interpreter itself. There are also elements of UML such as aggregation and dependencies that have tacit semantics, and were not possible to represent in Ruby. The experience from developing the mapping was that a large part of UML class diagrams could be mapped to Ruby by extending Ruby when needed. Ruby on the other hand has some exotic features that are difficult to represent in UML. These features are implemented to make Ruby flexible and include mixin modules and scooping differences.

Actions are also something that Ruby has extensive support for in its execution model, and UML has little support for in its current version. Therefore as explained in 5.3.2 information loss occurs when translating from Ruby to UML. UML 2.0 will have better support for actions, but it is unlikely that it will be possible to specify actions at the same degree of detail as in Ruby, even with this extension.

So, to what extent could Ruby and UML be mixed? The answer would have to be to a certain degree. As concluded in 5.4 the two languages should not be tried put together to make a new hybrid language, but still be used as separate languages making compromises to inspire the use of both of them.

The experiences from the field experiment help us in answering the second part of the problem; what are the implications using a mixed language approach? I sum up the most important experiences in short below:

- The prototype description using both UML and Ruby give a better view of the prototype than each of the languages would have alone. Ruby can precisely define actions and give hints about operation implementation and UML describe its structure and give better overview due to its graphical representation. UML perform better in user communication.

- The decision to perform translations is triggered by shifts in the developer's mode of operation, from an analytical to an experimental one or the opposite. The possibility introduced by the translation to do this quickly inspires a mixed approach and do behavioural shifts. However there is a need for further guidelines to when translations should be made. Such guidelines could be part of risk evaluation applied in mixed approaches.

- Ruby makes prototype development quick. It is possible to quickly test parts of the application. However the quick code – run – correct development cycle that make Ruby a good prototyping language is somewhat broken by the introduction of the translation phase, much like a compilation phase would.

- Applying changes in the small are easy to do in Ruby because of its flexible nature, and possibility to defer decisions. Larger structural changes are better to do in UML in

cooperation with the users. Being able to go back to UML to regain overview helps keeping a good architecture of the prototype, and countermeasure the problems of maintainability of an evolutionary approach.

The general conclusions to draw from these experiences are that the rapidity of prototype development is hindered by the translations. Still, the chances of going astray, in the wrong direction, are countermeasured by always focusing on the architecture of the prototype. This is dependent on how often the translations are made. There is still a risk of behaving in an experimental or analytical mode of operation for too long. This thesis has not given any suggestions to when translations should be performed, only that it is a good idea to do it. The mixed approaches emphasize risk evaluation as a method for shifting mode of operation. If the risk is high that we know to little about the problem domain to develop a good design, the development of a prototype is suggested. On the other hand if the risk is low that the design is wrong, no further prototyping or design elaboration has to be done and development can continue to the next cycle. Such risk evaluation can be taken down to a smaller level when using the mixed language approach. If the risk is high that the prototype has changed much from the design a translation is done and the design is re-evaluated together with the users. If there is a high risk that the design is wrong, a translation is done and it is tried out in practice.

This shows that having a way to combine two languages for specifying and prototyping does not solve anything on its own. It merely makes it easier to apply a mixed approach because there are ways to shift focus more rapidly. An additional effect of the effort is that prototyping and specifying are brought closer together and happen in parallel.

7 FURTHER WORK

The road ahead

The motivation for using a mixed language approach and the implications of combining Ruby and UML has been treated by this thesis. This last chapter outline how the result can inspire to do further work.

Ruby had to be extended to conform to UML semantics. I feel that further work must be done to refine these extensions. The Ruby extension for associations deserves more thorough elaboration. A problem with a programming language is that it offers to specific ways of realizing associations [ActionSemantics]. The extension developed still does not cover all attributes of the Association and AssociationEnd meta-classes. Especially this relates to one-to-many relationships. The solution of using references with special constraints only to represent such associations is probably not the best solution. The Ruby Application Archive contains a hierarchy of container classes that can be used in the extension developed. These classes will represent associations as objects, not references and will cover more of the attributes in the Association meta-class. Such incorporation will be straightforward and solve many problems, but the idea to do this came so late in the development of the thesis that it was not accounted for.

A problem with the combination of UML and Ruby, and dynamically typed languages in general is that there is a lack of tool support for such a combination. Having developed a model of translation and extensions to Ruby needed to realize such a mapping it is interesting to point out a few ideas about how such a tool could be. First there exist UML tools that support dynamic languages. An example of this is Object Domain, which uses Python as an internal scripting language to access UML elements. This tool also support generation and reverse engineering of Python code, although it has no support for problematic issues such as associations. A future tool for a Ruby, UML mixed language approach would have such support, with round trip engineering like Together Control Center. An ongoing Ruby project that could spawn the development of a UML / Ruby tool is the Freeride project. The goal here is to develop a complex Ruby integrated development environment (IDE) with support for developing, running and documenting Ruby code. For instance it looks to support interactivity like the Smalltalk environment [Goldberg84]. RDoc and test driven development support is also meant to be supported.

RDoc is perhaps the starting point of UML integration. It already has support for xml output. By incorporating the type information and association extensions developed in the thesis the output should be able to be a basis for UML generation. Many UML tools use XMI to exchange models. XMI is also based on xml so it will be possible to translate the output from RDoc to XMI via XSL or similar technology.

ArgoUML is an interesting open source UML tool. Already having support for Java generation and reverse engineering and XMI import, people are looking at ways to integrate more languages. The open source tools platform, Netbeans (http://www.netbeans.org), is very interesting. It already has support for the UML meta-model, giving programmatic access to UML elements. Poseidon for UML (http://www.gentleware.de) is a version of ArgoUML running on top of the Netbeans platform. New languages can be supported by Netbeans by specifying their meta-model in the OMG MOF language and provide implementation of the interfaces that are generated from this

model by Netbeans. Using the Netbeans platform for a Ruby / UML integration tool might be possible.

All in all there are many possible directions in developing further support for the Ruby / UML mixed language approach. But perhaps the first step would be to perform more quantitative studies about the feasibility of the approach as mentioned in the conclusion. This thesis has built the motivation, provided a model for the translation, and pointed out, as well as resolved, some of the difficulties of the approach.

BIBLIOGRAPHY

[ActionSemantics] Alcatel, I-Logix, Kennedy-Carter, Kabira Technologies Inc., Project Technologies Inc., Rational Software Corporation and Telelogic AB *Response to OMG RFP ad/08-11-01 Action Semantics for the UML* OMG, 2000

[AgileManifesto] Agile Alliance, *Agile Manifesto* http://www.agilemanifesto.org/

[Ambler01a] Ambler, S. *Agile modeling and eXtreme Programming (XP)*
Essay, 2001 http://www.agilemodeling.com/essays/agileModelingXP.htm

[Ambler01b] Ambler, S. *Artifacts for Agile Modeling: The UML and Beyond*
Essay, 2001 http://www.agilemodeling.com/essays/modelingTechniques.htm

[Ambler01c] Ambler, S. *How the AM Practices Fit Together*
Essay, 2001 http://www.agilemodeling.com/essays/practicesFitTogether.htm

[Andersen86]
Andersen, Niels E. et al. Chapter 3: *Hvad er systemudvikling? Professionel systemutvikling forudsetter teori* from *Professionel systemudvikling: erfaringer, muligheder og handling.* Teknisk Forlag, København 1986.

[Budde91]
Budde. *Prototyping – An approach to evolutionary system development.*
Springer Verlag, 1991.

[Bock94]
Bock. Odell, J. *A Foundation for Composition.*
Journal of Object-Oriented Programming Vol7, No6. Oct 1994

[Bock96]
Bock. Odell, J. *A User-Level Model of Composition.*
Report On Object-Oriented Analysis and Design Vol2, No7, May/June 1996

[Boehm84]
Boehm B. W., Gray T. E. Seewaldt T. *Prototyping versus specifying: A multiproject Experiment.* IEEE Trans. Software Eng., SE-10(3):290-303 1984

[Boehm88]
Boehm, B W. *A spiral model of software development and enhancement.*
IEEE Computer: Innovative technology for Computer professionals, 1988

[Booch94]
Booch, G. *Object Oriented Analysis and Design with Applications.* (2nd Edition), The Benjamin Cummins Publishing Co. Inc., 1994

[Booch98]
Booch, G. Rumbaugh, J. *The Unified Modelling Language User Guide.*
Addison - Wesley Pub. Co., 1998

[Booch99]
Booch, G. Jacobson, J., and Rumbaugh, J. *The Unified Software Development Process*
Addison - Wesley Pub. Co., 1999

[Christensen98]
Christensen M et al. *Architecture of prototypes and architectural prototyping – The dragon experience.*
University of Aarhus 1998

[Cook94]
Cook, S., and Daniels, J. *Designing Object Systems.* Prentice Hall, 1994

[Cunningham97]
Cunningham, D. *User-centered evolutionary software development using Python and Java.*
Proceedings of the 6th annual Python Conference 1997

[Esterby-Smith91]
Esterby-Smith, M., Thorpe, R., Lowe, A. *Management research: An introduction*
SAGE Publications Ltd. 1991

[Feldt02]
Feldt R., Johnson L., Neumann M., *Ruby developers guide*
Syngress Publishing Inc., 2002

[Fowler00] Fowler, M., Scott, K. *UML Destilled. A Brief Guide to the Standard Object Modelling Language.*
Addison-Wesley Pub. Co. 2000

[Gamma95]
Gamma E., Helm R., Johnson R., Vlissides J.
Design Patterns: Elements of reusable object oriented software.
Addison-Wesley Pub. Co. 1995

[Ghezzi98]
Ghezzi, C., Jazayeri M., *Programming Language Concepts*
3. Edition. John Wiley and Sons, Inc. 1998

[Goldberg83] Goldberg, A., Robson, D. *Smalltalk-80. The Language and its Implementation.*
Addison-Wesley Pub. Co. 1983

[Goldberg84]
Goldberg A. *Smalltalk: The interactive programming environment.*
Addison-Wesley Pub. Co. 1984

[Graver90]
Graver, Johnson J. O., Ralph E. *A type system for Smalltalk.*
The Seventeenth Symposium on Principles of Programming Languages 1990

[Hunt00]
Thomas D., Hunt, A. *Programming Ruby: The Pragmatic Programmer's Guide.*
Addison Wesley, 2001, see also http://www.rubycentral.com/book/index.html

[Kent99]
Kent, B. *Extreme programming explained: Embrace Change.*
1. Edition Addison-Wesley Pub. Co. 15.10.1999

[Mathiassen90]
Mathiassen, L. Stage, L. *Complexity and uncertainty in software design.*
Proceedings of the IEEE International Conference on Computer Systems and Software Engineering, p482-489. IEEE Computer Society Press 1990.

[Mathiassen92]
Mathiassen, L. Stage, L. *The Principle of Limited Reduction in Software Design.*
Technology and People, 6, 2, 1992

[Mathiassen95]
Mathiassen, L. Stage, L. *Prototyping and Specifying: Principles and practices of a mixed approach.*
Scandinavian Journal of Information Systems, 7, 1.1995

[Mcgrath95]
Mcgrath, J. *Methodology matters: Doing research in behavioral and social sciences.*
From Human-Computer Interaction: Toward the Year 2000 p152 - 169, Morgan Kaufmann Inc. 1995

[Meyer96]
Meyer, B. *Building bug-free OO software, an introduction to design by contract.*
Object Currents, SIGS Publication, Vol1, No3. 1996

[Meyer97]
Meyer, Bertrand. *Object oriented Software Construction.*
Prentice Hall, 1997

[Meyers93]
Meyers, B A *Why are Human-Computer Interfaces Difficult to Design and Implement*
Technical report CMU-CS-93-183, School of Computer Science,
Carnegie-Mellon University, Jul 1993.

[Mills80]
Mills, H. D., O'Neill D., Linger, R. C., Dyer, M., and Quinnan, R. E.
The management of software engineering IBM Sys. J., 24(2) 1980.

[Odell94]
Odell, J. *Six Different Kinds of Composition.*
Journal of Object-Oriented Programming Vol5, No8. Jan 1994

[Palsberg91]
Palsberg, J. *Object oriented type inference.* OOPSLA 1991

[Ploesch97]
Ploesch, R. Kepler, J. *Design by contract for python.*
Proceedings of the Asia Pacific Software Engineering Conference, 1997

[Ploesch98]
Ploesh, R. *Tool support for design by contract.*
Proceedings of the TOOLS-26 conference, 1998

[Raymond99]
Raymond, E. S. *The Cathedral and the Bazaar Musings on Linux and Open Source by an Accidental Revolutionary*
Rev 1.40 O'Reilly & Associates Inc 1999

[Reenskaug96]
Reenskaug, T. *Working with Objects: The OOram Software Engineering Method*,
Manning Publications, 1996

[Richter99]
Richter, C. *Designing flexible object-oriented systems with UML.*
Macmillan Technical Publishing, 1999

[RubyWeb]
Official Ruby Web Site. www.ruby-lang.org

[Simon57]
Simon, H. *Models of Man: Social and Rational*
John Wiley and Sons 1957

[Sommerville95]
Sommerville, I. *Software Engineering.*
Fifth edition Addison-Wesley Pub Co 1995.

[Spivey92] Spivey, J.M. *The Z notation: A Reference Manual*
2. Edition Prentice Hall 1992

[Suchman87]
Suchman, L. A. *Plans and situated actions. The problem of human machine communication.*
Cambridge University Press 1987

[Sutherland97] Sutherland, J. *Scrum web page*
http://www.contolchaos.org
http://www.tiac.net/users/jsuth/scrum/index.html
http://www.jeffsutherland.org/scrum/index.html 1997
See also: Beedle, M., Devos, M., Yonat, S., Schwaber, K., Sutherland, J.
Scrum: an extension pattern for hyper productive software development
http://www.controlchaos.org/scrum.pdf

[OMG00] Object Management Group *Unified Modeling Language Specification* Version 1.4 First Edition OMG ad/00-03-01, 2000

[OMG02] Object Management Group *Unified Modeling Language: Superstructure* Version 2.0 alpha R4 (draft) ad/2002-08-26, 26. August 2002.

[Wirth74]
Wirth N. *On the design of programming panguages.*
Proc IFIP Congress 1974, 386-393, North-Holland, Amsterdam

APPENDIX

A. Ruby source

The source code is available on the accompanying CD-ROM. The Rdoc output is also available for easier browsing. In addition the UML wrapper extension and test cases are available. Use Rdoc to browse the Ruby code. Rdoc is run through a web browser, start ./ruby/Final/doc/index.html on the CD-ROM.

B. UML diagrams

The UML diagrams are available on the accompanying CD-ROM as a Rational Rose model. For readers that do not have Rational Rose available the diagrams for the final solution are also presented here. The diagrams have been split compared to the ones in the Rose model to better fit the pages.

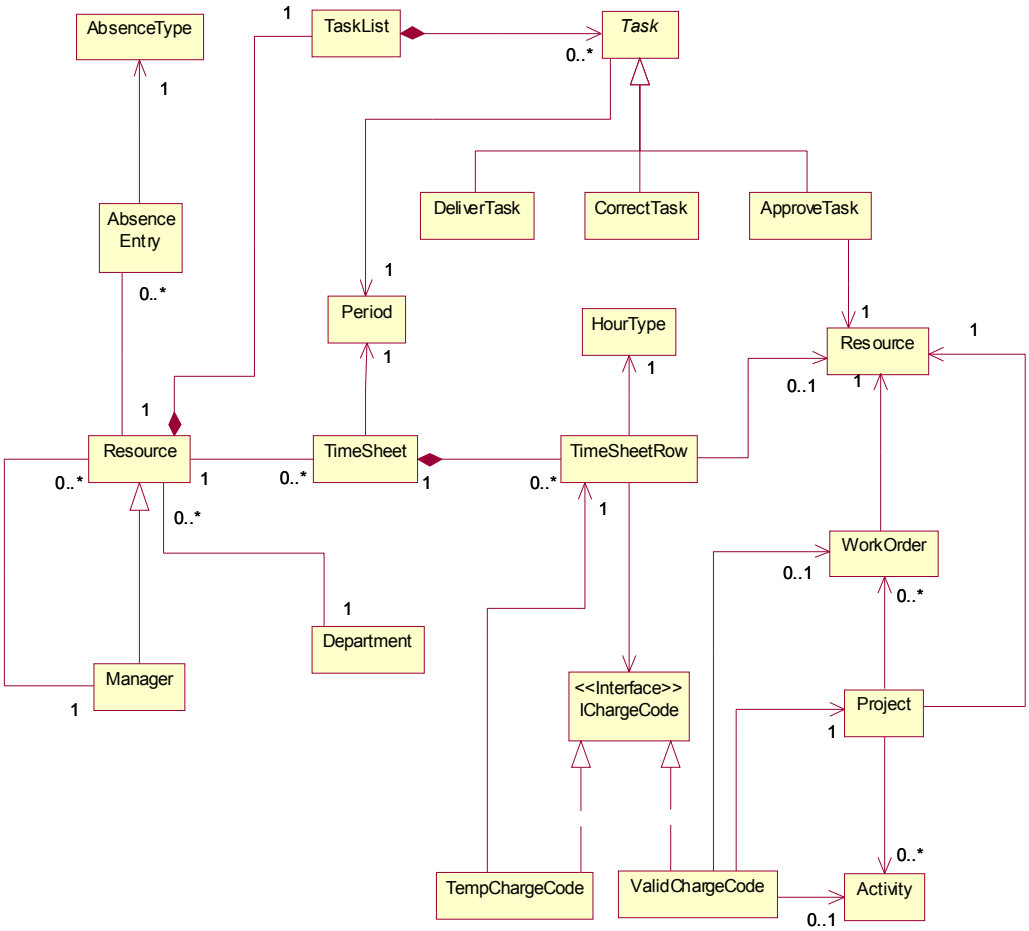


Diagram 1 Specification structure

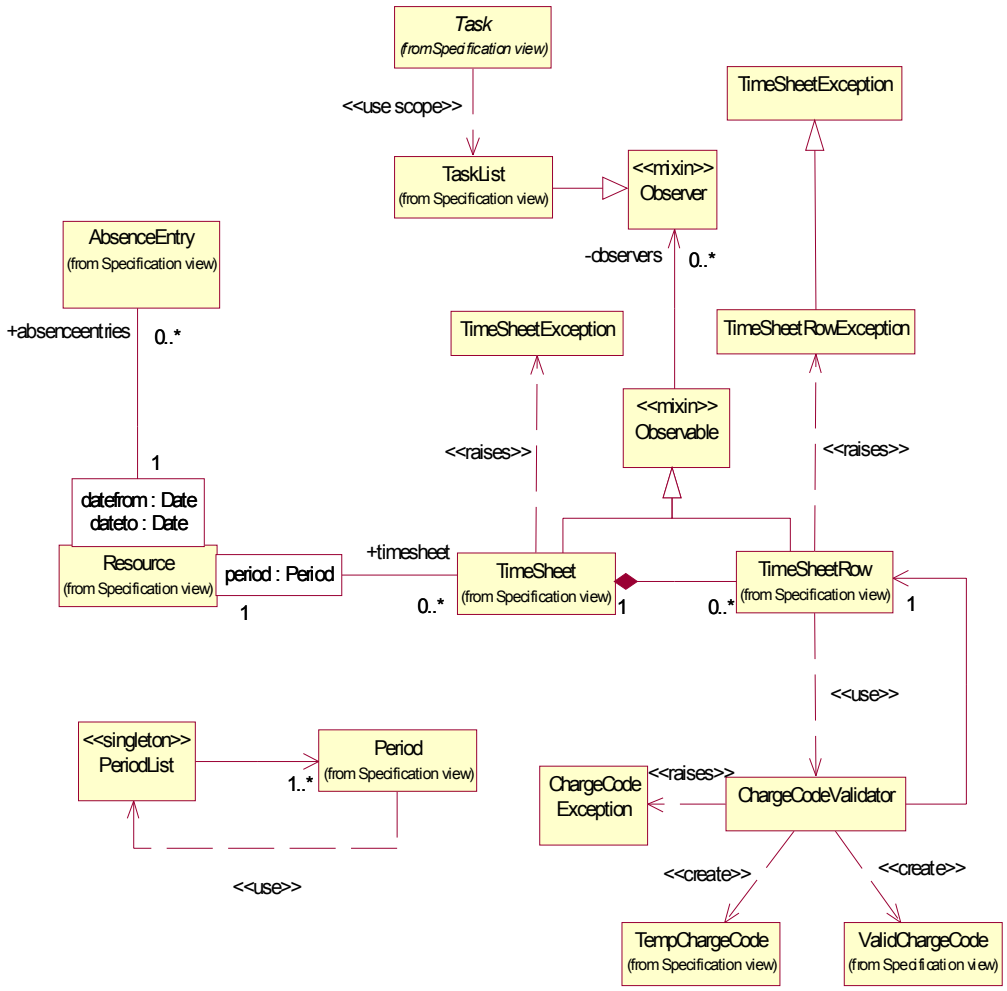


Diagram 2 Implementation structure extensions

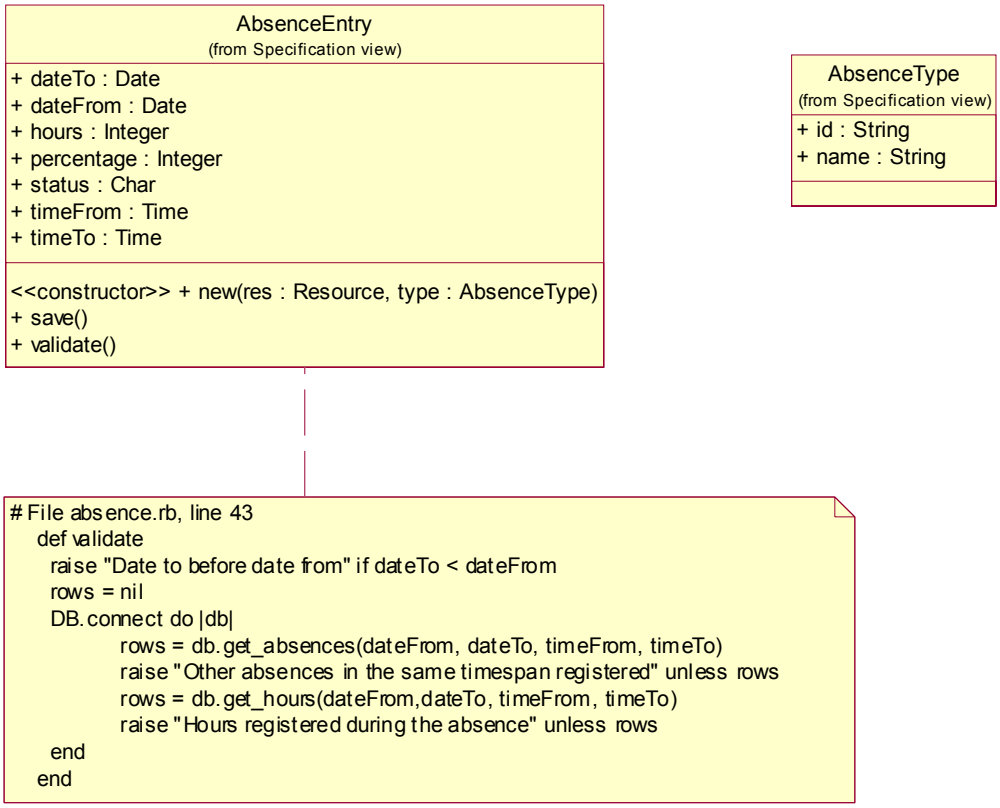


Diagram 3 Absence details 1

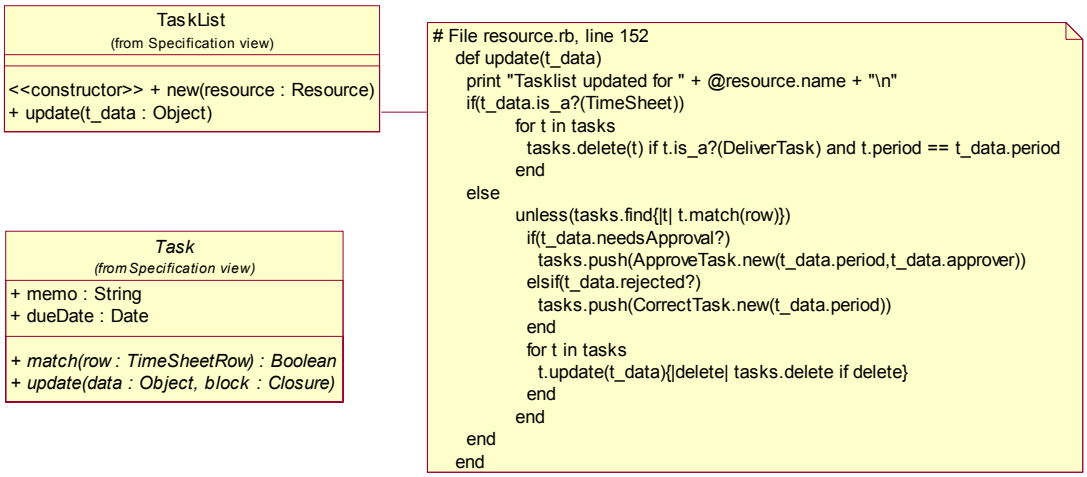


Diagram 4 Absence details 2

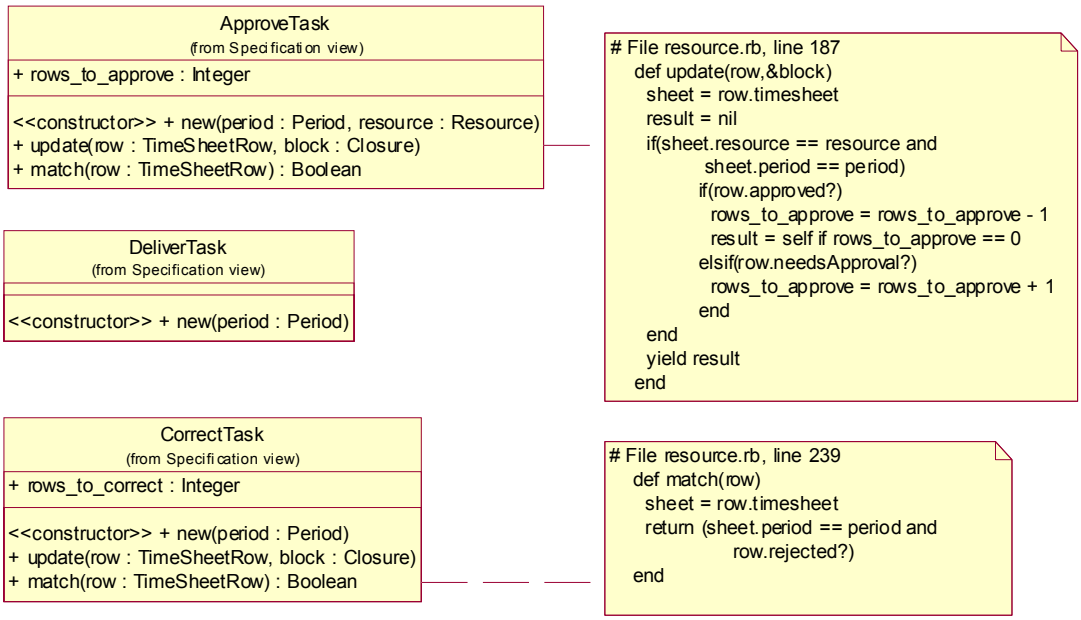


Diagram 5 Absence details 3

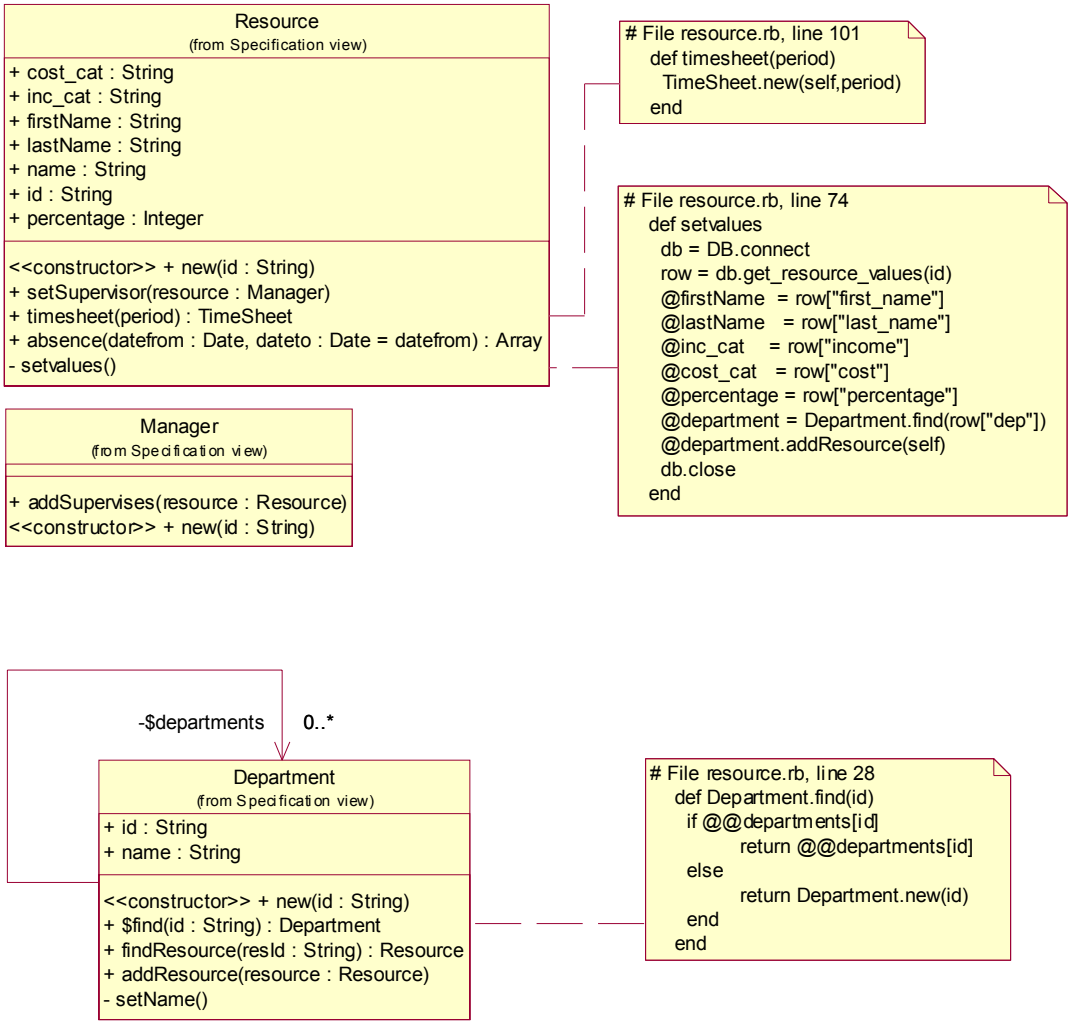


Diagram 6 Resource details

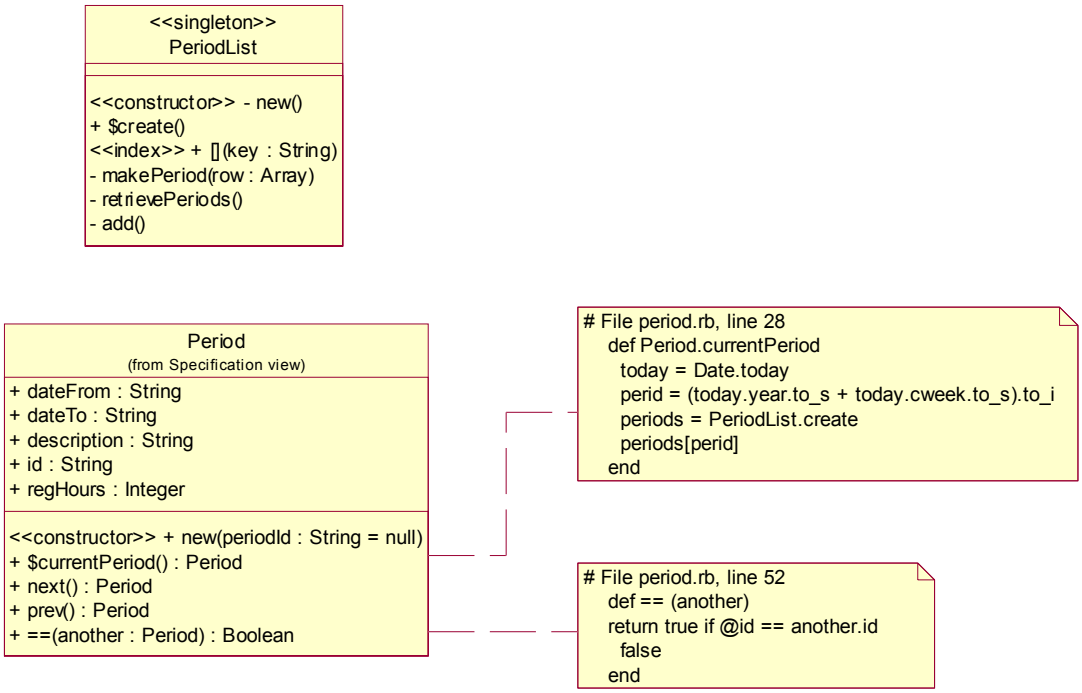


Diagram 7 Period details

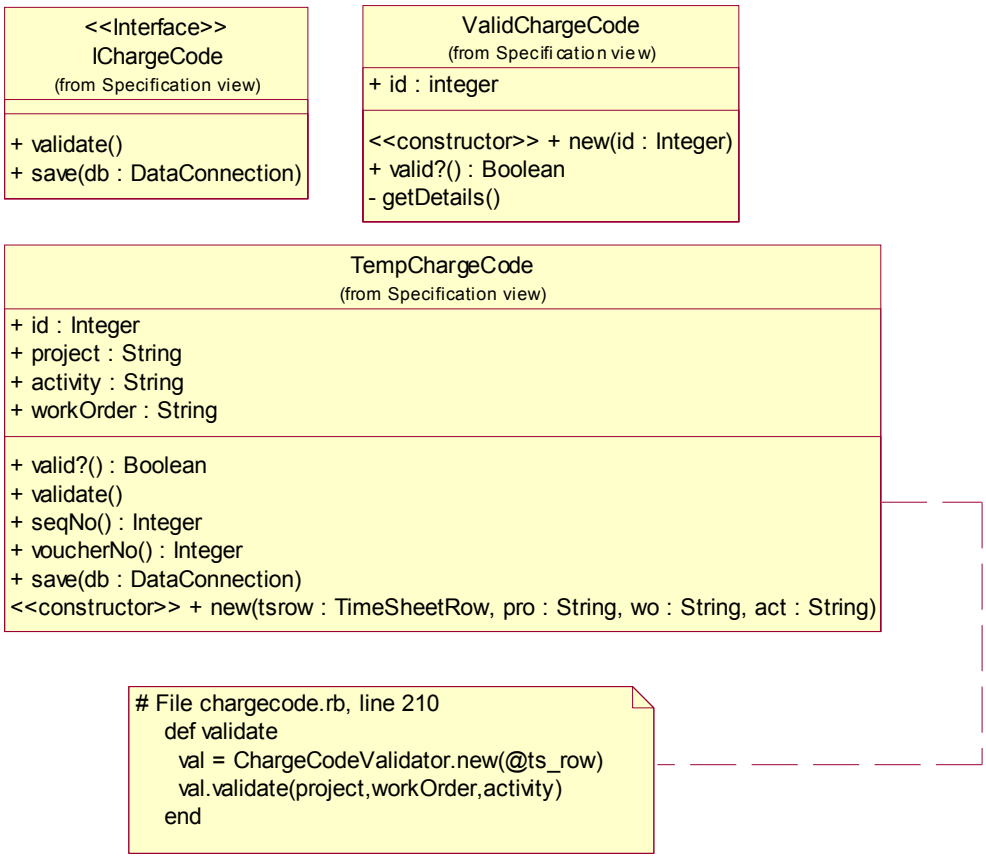


Diagram 8 Charge code details 1

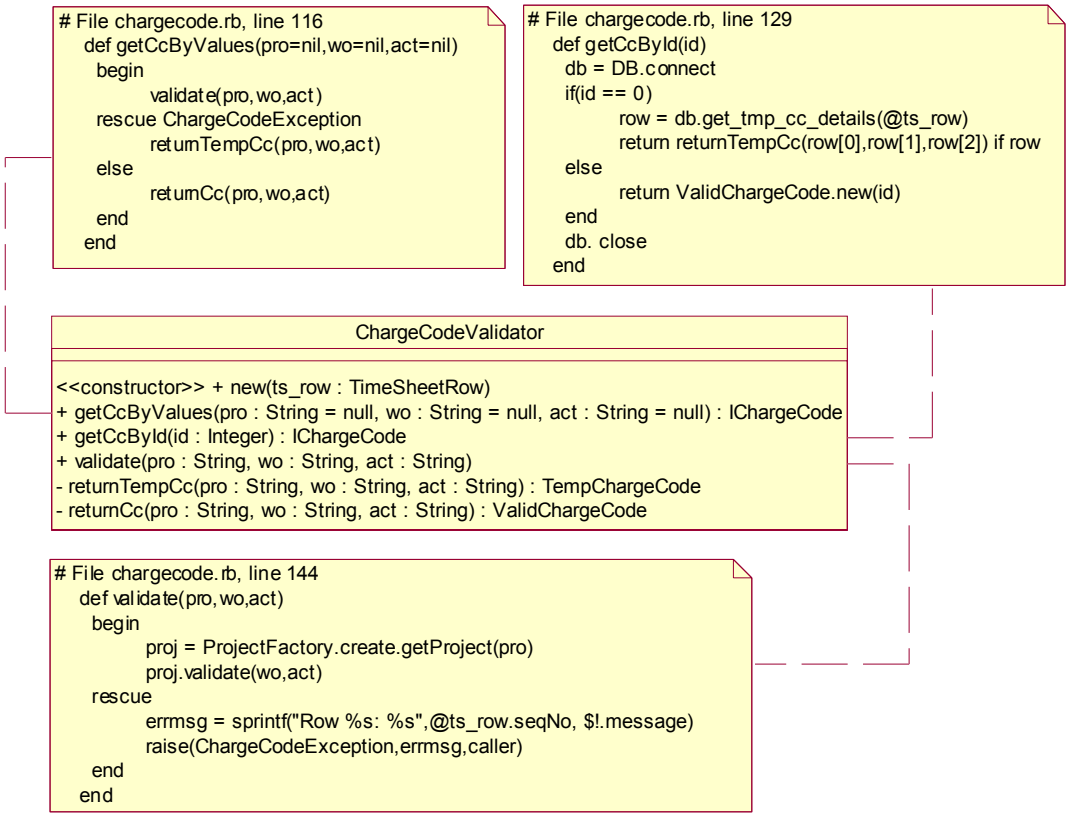


Diagram 9 Charge code details

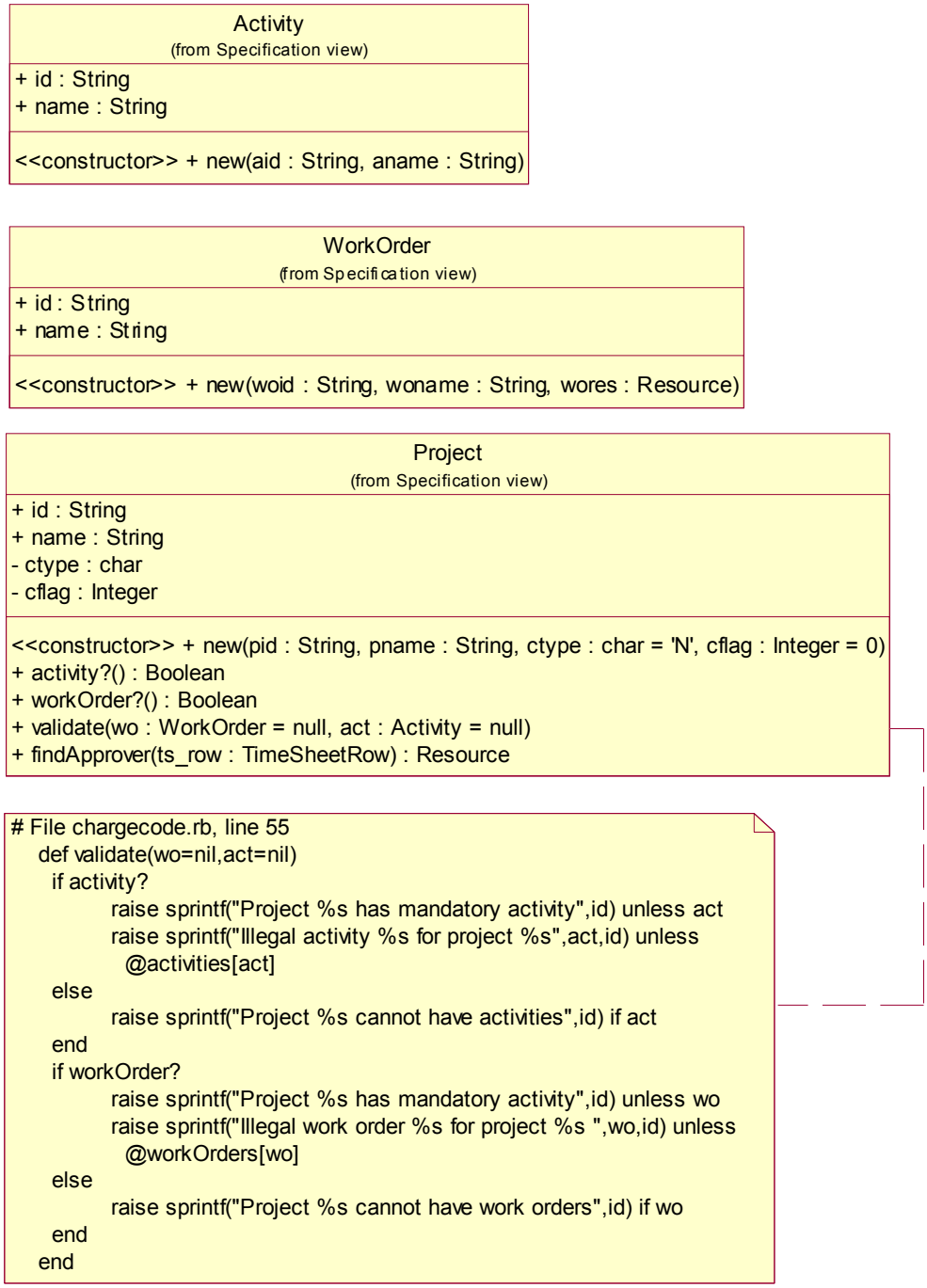


Diagram 10 Charge code details 3

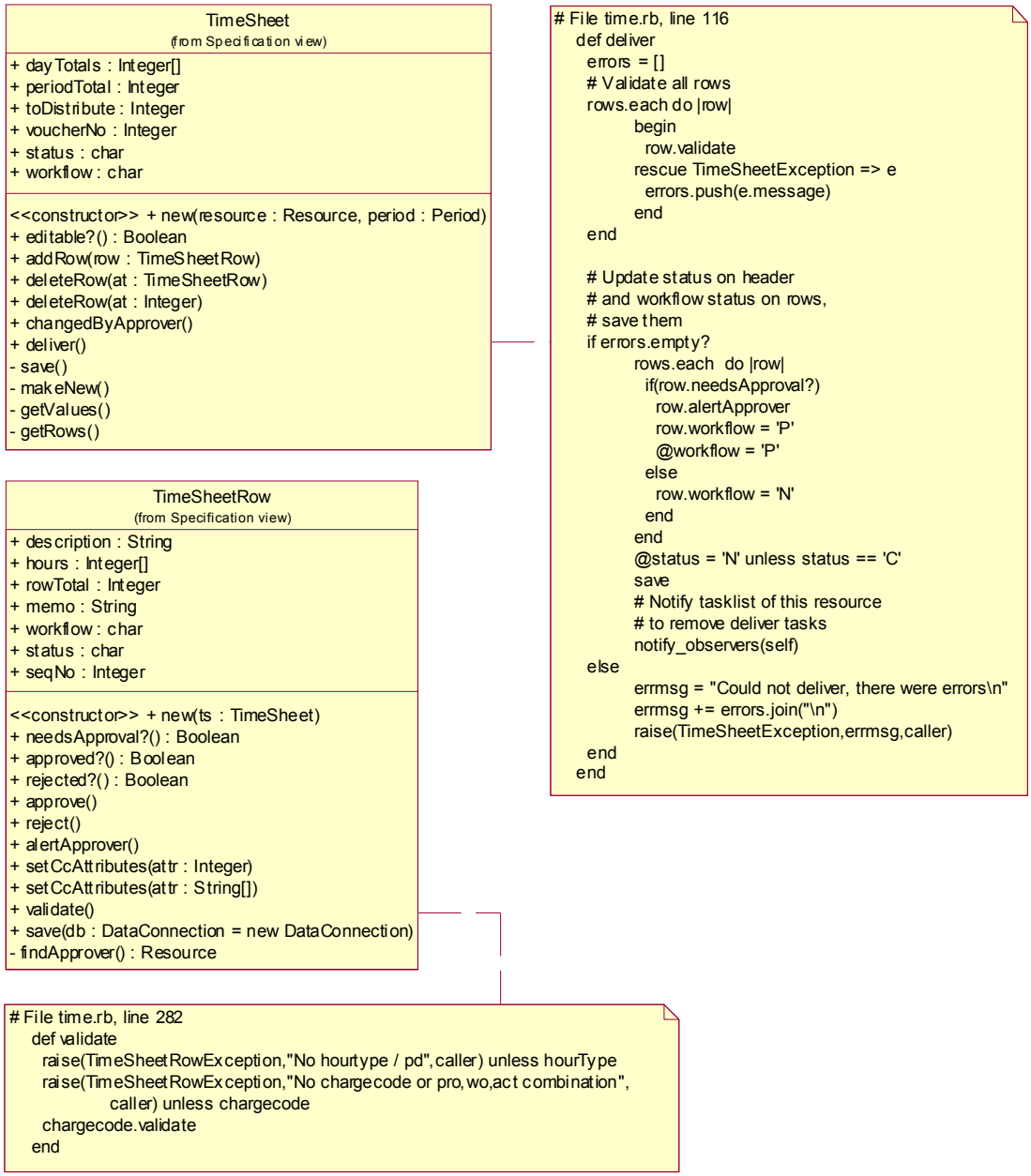


Diagram 11 Time details