“Ragnarok” Contours of a Software Project Development Environment

Progress Report
by
Henrik Bærbak Christensen

Summer, 1996

Abstract

This report describes the current state of my research in software development environments. I argue in favour of strong support for *project management, comprehension and navigation,* and *collaboration* primarily based on experiences from developing large-scale industrial-strength applications.

An underlying model of such an environment, named “Ragnarok”, is outlined. A design and first prototype of important parts of Ragnarok is described as well as some results from initial experiments.

Contents

1 Introduction

This report outlines one year of research in the field of "Software Development Environments" with special interest on the problems facing large-scale software projects.

Crafting large industrial-strength software applications requires numerous tasks to be performed covering a large spectrum of activities. In one end of the spectrum we must sell, plan, staff, and manage a project and at the other we have to produce, debug, and test code in some programming language(s).

The focus of the current work is at the managerial end of this spectrum. This is *not* because the problems at the language-near end are uninteresting; on the contrary strong compilers, editors, debuggers, etc. are important, and weak support can be costly in terms of wasted (human) resources. However as the size of a project increases more and more emphasis is put on the ability to collaborate and maintain overview and control; and failure on these aspects generally have more devastating consequences. Also research and tool support for the language-near activities has received much attention whereas support for the managerial aspects is more uncovered ground.

The focus is also on software *projects* as is evident from the title of the report. Projects are seen as having a well-defined goal which is achieved through a series of activities; activities which must be monitored and controlled while keeping overview of their dependencies and contributions in fulfilling the goal. Thus the term "software project development environment" is meant to stress the focus on project support in a software development environment and not as defining a new category of environments. Ragnarok could be classified as a programming environment [Nørmark89] or an integrated project support environment/software engineering environment [Sommerville89].

This progress report is divided in the following manner:

Section 2 describes my interest in the subject as well as outlines some experiences gained in my previous job as system designer and implementor.

These experiences are summarised in some contours of a system in section 3.

The next three chapters are devoted to what I see as basis for strong support in developing software projects: A model of software structure in section 4 combined with an approach to software configuration management (SCM), section 5. The SCM model is extended in section 6 to focus on collaboration issues. While the ideas presented in section 4 and 5 have been tested in an implementation, the ideas in section 6 are still somewhat in their infancy.

The ideas underlying the user interface are described in section 7 followed by a short description of the design of the Ragnarok prototype, section 8.

In section 9 I describe some of the experiences with the prototypes from two teams as well as my own experiences.

Section 10 is devoted to some of the ideas I have for continuing the present work; and I will conclude in section 11.

The ideas outlined in this report are grown out of the object oriented tradition and experience with building industrial-strength application using object oriented technology. Though I do not see any immediate problems in using Ragnarok with other approaches to system development, this aspect has not been studied in great detail.

2 Motivation

In the Nordic mythology "Ragnarok" is the big chaotic struggle between gods and giants in which the old world is destroyed.

I think that many developers who have participated in large software projects will readily agree that a sensation of "Ragnarok" is not completely unfamiliar.

Here "Ragnarok" is the name of a software development environment designed to address some of the issues associated with constructing and especially managing large-scale software projects[1].

Many of the ideas for Ragnarok grew out of my experiences as chief architect as well as implementor of a family of industrial applications for nearly three years. One of the main aspects of industrial development is in my opinion stated by Bertrand Meyer: "... *once everything has been said, software is defined by code.*" [Meyer88, p. 30]. It is a high quality application that pays the bills; not nice OMT diagrams, detailed milestone reports, nor reusable class libraries. It is therefore all too common that

[1] Bearing the name "Ragnarok" is ironically meant: Hopefully using it should be less chaotic than the name suggests.

software projects focus too much on producing code and loose control of other import managerial tasks as the deadline rapidly approaches. Ragnarok is an attempt to help avoiding this unfortunate situation.

I have therefore deliberately chosen a *broad* approach to the field of software development environments. Having a master degree in astrophysics has taught me that theories are just that – theories – until they stand the test of confronting nature itself. I strongly believe in testing ideas in a real situation; and only a successful adoption there provides evidence for the usefulness of an approach.

Providing *good* solutions for a *large number* of often-occuring problems (as opposed to a *"perfect"* solution for a *single* one) is in my opinion vital in order to convince other people to try new tools and approaches—including of course "Ragnarok". I will try and provide that and hope to suggest at the soundness of my ideas by people adopting and using them in their own software development projects.

2.1 A retrospective case study

For nearly three years I was engaged in developing a family of products for semi-automatic weather observation in airports. The system consisted of hardware and four special purpose software applications running on IBM compatible PC's in a local area network. I was chief architect, in the sense of Herbsleb [Herbsleb et al. 95], as well as implemented large parts of two of these applications (both running MicroSoft Windows).

The total software side of the product family accounted for about 11.500 staff-hours with project team sizes ranging from 3-7 members. Ten systems were delivered and maintained when I left the company. The first system accounted for almost half of the total staff-hours and was greatly underestimated in the budgets; however because of the many systems delivered and a high degree of reuse a balance was found.

Though the systems were a success both in terms of stability and from the view point of the daily users, we encountered many problems in the development process itself. It should be noted that because the company had been in the field for almost 10 years it had good domain knowledge and a sound understanding of the problems of the users. Thus the problems were not tied to the often reported discrepancy between user expectations and developers understanding of the domain.

Below a short list of problems is given. A more detailed description can be found in [Christensen95].

Problems concerning management

Though a work break down had been made, based on initial design, which defined basic milestones the plans were never the less quickly abandoned. This was caused by a number of reasons:

- The project manager was involved in software development because of lack of resources. Hence he had a strong inclination for producing the product rather than managing the process.
- The initial design was often revised as new insight was gained. However our project management tool was pretty bad at handling these kind of changes – and all changes had to be transferred to the management tool manually as there was no link to our programming environment.

Process metrics, in the form of logging and categorising staff hours, were collected for all projects. However as Jacobson notes [Jacobson et al. 92]: "Actually the real problem with metrics is that they are not used." It created a vicious circle: As the metrics were not used people were pretty sloppy about getting the data correct, thus the data could not really be used for planning the next project and so on.

Problems concerning collaboration

We generally lacked a *design language* for communicating and documenting design ideas and decisions [2]. This made it hard to share ideas, hard to document them in a compact way, meaning lesser reuse, and it was more difficult to introduce new developers to the projects. What we really wanted and lacked was a "road map" giving the rough outlines of the system.

Design was not *reviewed* properly. Therefore some problems about protocols between the different units were detected rather late in the development phase.

We had no real *version- and configuration management* tool support. Sometimes already fixed bugs suddenly reappeared because code was overwritten by an accidental old copy. Much time was spent diff'ing and merging when two developers had modified the same set of source files at home.

[2]The design notations Booch [Booch91] and OMT [Rumbaugh et al. 91] were still in their infancy when the basic design was initiated.

We had no support for *collaborative work and awareness* on source files. If two developers had to make changes to the same source file we adopted to ask each other to save the file before editing it ourself – of course this is error-prone and cumbersome.

Problems concerning sense of locality

A single system consisted of more than 1000 files of source code, scripts, documentation etc. Finding one's bearing in this jungle was guided by a (rather deep) directory structure; the final structure emerged after a major re-organisation after completing the first two systems [3].

The directory structure mimicked dependencies between modules and source files which aided in navigating in the large number of files. However it could of course not indicate other important relations like e.g. associations between classes so in this respect we had to completely rely on our own understanding and memory of the application structure (lacking a "road map" as described earlier). This was of course a major problem to developers introduced late in the implementation and/or maintenance phase but even the designers themselves got confused from time to time.

Our programming environment had a browser facility which could display inheritance graphs. However the browsing information was not available before the application could compile successfully. In other words you often did not have the browser facility when you wanted it most desperately.

Problems concerning maintenance

More than ten systems are in operation in Denmark. Though they all share a large common core of code, they are still tailored for individual needs and requirements of the specific airport. Though ideally *all* commonality should be factored out for ease of maintenance this solution was neither cost-effective nor practical for all parts of the application. Tailoring was therefore partly handled by *reuse through source copy*, for example there was a "main.cpp" file for every application though perhaps 80% of the code was identical[4]. But this approach of course had the drawback that when a bug was detected in copied code it had to be fixed in ten different but identical looking source files, and you had to run the same test procedures in order to verify that the changes behaved identical in all contexts. As both tasks were manual and tedious, they were rather error-prone.

Release control was obtained by making a complete raw backup of the project, and a release note stating the version number and bugs/changes performed was written. This scheme worked but restoring an old release for inspection was cumbersome because a complete backup had to be reinstalled and great care exercised not to accidentally overwrite newly created code.

3 Contours

In the following section I will try to outline the contours of an environment geared towards large-scale software development.

3.1 Hypothesis

My working hypothesis can be summarised in the following statement:

Hypothesis:
Much would be gained if the environment supported and encouraged a wider spectrum of the project activities required by large-scale software development, giving immediate benefits for any cost introduced.

It is my general impression that software developers like crafting software much more than managing the crafting process itself. This is especially a problem in teams where a manager is responsible for creating parts of the software.

If we can support more processes to the extent that they become "part-of-the-production" instead of "stealing-time-from-production" then it is more likely that they will be performed even in a tight scheduled project.

[3] Though the re-organisation was necessary it was rather costly because it took weeks before we again were proficient in locating things in the new structure.

[4] Though it was not necessarily the *same* 80% that was shared.

Nørmark argues in favour of raising the "tool abstraction level" i.e. let environment tools take over more routine work [Nørmark89]. This is certainly true and desirable but not all routine work can be taken over by tools (for instance logging and categorising staff-hours) and it is also important to support and encourage activities in the high end of the activity abstraction level (for instance to provide overview in project management).

3.2 Aspects

Based on the experiences shortly outlined in the previous section, I have identified three major aspects that I feel a development environment could fruitfully support. These are *project management, comprehension and navigation*, and *collaboration*. These aspects are not orthogonal but for the ease of presentation they are dealt with separately.

3.2.1 Project management

Our work-break-down structure was always out of date because it had to be updated manually when changes were made to design.

Hypothesis:

If we can associate project management attributes and tasks directly with software components, we need only maintain a single structure.

I propose to view management aspects as intimately and directly related to the design structure of a software project. This way there is never a useless and out-dated work-break-down structure—because any change in the application structure *is* a change in the work-break-down structure as well.

Goldberg gives an example list of task related management attributes which could be used as a template [Goldberg et al. 95, Chap. 7]. There are other obvious candidates to associate with components: Modification requests [Tichy88]/"bug reports", hyper-links (in)to other material like schedules (PERT/Gantt charts [Mikkelsen et al. 89]), requirements specifications (for tracking requirements through the life-cycle), documentation, etc. Quality assurance is an important aspect of management as well: Test suits and, in cases where test programs are impossible like e.g. testing user interface specifications, *checklists* are also obvious candidates for being directly associated with components.

Collectively I will denote any kind of attributes associated to components for *annotations*.

Having a single structure may also diminish the "reporting delay" of e.g. logging staff-hours. Staff-hours may be logged directly onto the components you are working on thus managers can track progress much closer and approximately in "real-time". The system itself can aid the developer in telling which components she has been working on and when, thus encouraging higher quality process metrics. Thus there is a better chance of having realistic historic data as resource when budgeting new project.

In this approach viewing *code* (files, modules and their relations) and *management data* (budgets, milestones, deviations from estimated staff-hours) are just different *views* on the same basic application structure.

3.2.2 Comprehension & Navigation

I suggest two ideas that in my opinion will help in comprehension and navigation:

- Provide support for building and maintaining a "road map" i.e. the overall outline of the (logical) application structure, and to keep this consistent with the actual application source code. Ideally the "road map" should be able to be displayed at different levels of detail.
- Provide effective, transparent support for the mapping between the logical and physical software structure. This would allow the developer to view and navigate in the logical application structure (the "road map") and let the environment help locate actual files on the physical store.

The "road map" would typically use a common design notation like OMT [Rumbaugh et al. 91], Booch [Booch91], unified method [Booch et al. 95] or (company defined) variants of these.

Furthermore I advocate the use of a *spatial metaphor* in designing the "road map":

Hypothesis:

If we associate logical design components with salient, visual landmarks having a unique position and extent in a physical space, navigational skills as well as overview will improve.

Humans are usually much more proficient in navigating in a physical, concrete, world than an abstract design space. I see great potential in making design space more physical and concrete; not just in order to increase navigational abilities in code but just as much to get an overview over complicated relations; I will elaborate further on this point in section 7.

A final, pragmatic, but nevertheless import, point I want to make:

An act of navigation should not create lots of individual visual elements.

Many systems like for instance Mjølner ORM [Magnusson94], Self 4.0 [Maloney95, Smith et al. 95], many commercial CASE tools (Select OMT, Rational Rose, Cadre ObjectTeam, and others), and many applications using window based desktop systems like MicroSoft Windows, X11, etc., produce large amounts of new windows/object when searching for a specific item (intermediate steps in the navigation). Consequently (too) much time is spent constantly tidying up the workspace.

3.2.3 Collaboration

I find that these issues are the most pertinent:

- To share a common design language within the team by which the application architecture can be understood, discussed, documented, and reused.
- To allow the individual developer to coordinate work on common tasks and avoid overwriting, redoing or in other ways destroying the work of others.
- Collaborative awareness i.e. to be aware of actions taken by other individuals or groups that may affect one's own work situation.

A "road map" using a common notation would support the first issue.

I see a strong *software configuration management* (SCM) model as a good foundation to support coordination and collaborative awareness. Developers could share software components by having access to common versions while being able to work in isolation when needed. Awareness of new versions of components as well as ongoing work could be mediated through messages and/or a highlighting scheme for the components.

A strong underlying SCM model will also have great value with respect to project management as it provides *traceability* of development effort (meaning better control) as well as release control.

My emphasis on collaboration as an important issue in development environments is supported in [Herbsleb et al. 95] where it is stated that *"The lack of tools to support annotating, keeping abreast of changes, bringing novices up to speed—and collaborative use in general—is a serious drawback"*.

3.3 Discussion

Based upon the discussion above I see the following as cornerstones for a software project development environment:

- Ability to directly manipulate the (logical) parts/components of a software application
- Ability to document the relations between components using a graphical design language like e.g. unified method, providing a "road map" of the application design.
- Ability to enhance overview and navigation in the architecture of components using a spatial metaphor.
- Ability to attach, view, and modify annotations (project management and other kinds of attributes) to these components.
- Provide a strong underlying software configuration management model in order to:
  - Ease collaboration by providing awareness, easy sharing, as well as ability to work undisturbed.
  - Provide traceability of the process as well as of released material.

As I have a strong wish for testing my ideas in a real setting I have also been lead by pragmatic considerations. One important decision is *not* to provide language support; though useful indeed it would direct research towards implementation issues and I am more interested in the overall aspects of software projects.

4 Model of Software Structure

Small strokes fell great oaks
Benjamin Franklin

In all but the smallest projects there is a need to divide and structure tasks into manageable pieces. This is the well-proven maxim of “divide-and-conquer” known since ancient time for managing large systems[5].

The present section outlines the model used within Ragnarok and lays the foundation for the management model in section 5.

4.1 Software component

The basic building-blocks in a software system I denote *software components*. An informal definition of a software component is:

> *A part of a software system that is perceived as a logical whole by the team members.*

Software components are often conceived by a single or a few chief architects [Herbsleb et al. 95] and introduced to a broader team of developers and implementors after the initial analysis and design phases.

As examples of components you may think of a C-compiler, an operating system, a container-library, a module to interface certain hardware, a communication package, a set of classes encapsulating meteorological domain knowledge, or a single class in an application.

Software components are natural units for discussion, documentation, and reuse within a development team: “I think I've found a bug in the graphics component”, “No, I haven't quite finished the documentation on the database component yet”, “Here we can use the field editor component we developed in the last project”, etc.

Software components naturally form a hierarchy: A C-compiler consists of many smaller and simpler components like preprocessor, parser, checker, code-generator, etc. Each of these are again composed of even smaller and more basic components. This hierarchy of components represents different levels of *abstractions* in the application: At a high level of abstraction a C-compiler is considered a whole whereas at a detailed level it is considered as consisting of a lot of modules and files.

A small example is depicted in figure 1.

Software components as described here are inspired by *class categories* [Booch91, Chap. 5] and *clusters* [Mathiassen et al. 93, Chap. 4]—and of course experience from crafting industrial applications, where creating reusable software components is vital.

4.2 Components and project tasks

From a project point of view software components are also natural boundaries for defining *tasks* and delegating responsibilities. In large projects smaller teams are assigned to subsystems with a (sub)project manager; subsystems are again subdivided until responsibilities, tasks, schedules and staffing are associated with all parts of a system.

This structure is of course not static, new components may be created and others deleted during the life-cycle.

4.3 Source- and derived components

As Tichy [Tichy88] distinguishes between source- and derived objects as manually versus automatically generated objects, I distinguish between *source components* and *derived components* using the same criteria. Typically files associated with a source component are processed by a compiler, linker, etc. to produce a series of object files or perhaps a library. A single source component may have several derived components depending on compiler directives, setting of switches for optimisation, etc. The derived component knows the setup of the translator (for instance the switches of the compiler) that constructed it. This way unnecessary translations can be avoided by the system caching often used derived components for system-wide use.

The emphasis in this report will be on the source part of software components.

[5] Like the Roman Empire.

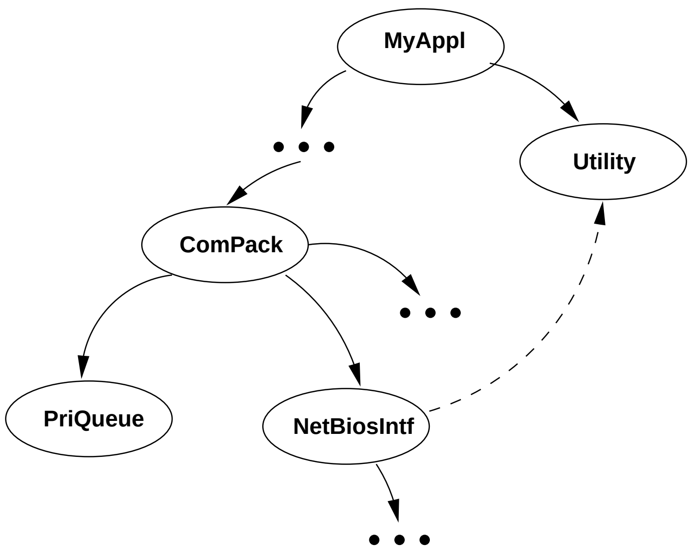

Figure 1: Example of a software component structure. Component MyAppl represents the root component with some unspecified child components (the dots) and the component "Utility" representing a class utility in the Booch sense. Further down in the hierarchy there is a component "ComPack" which represents a class for communication. ComPack uses a special purpose priority queue, PriQueue, and a net bios interface, NetBiosIntf, defined as two child components. The latter depends on some functionality defined in the Utility component. A components child list is shown by solid lines; depend-on lists by dashed lines (both explained in section 4.4).

4.4 Physical representation

The above intuitive definition of a software component is in terms of the *logical design architecture of a software system*. However a component must of course have a *physical manifestation*, traditionally in terms of a number of source files and possibly additional data like diagrams, documentation, bug reports, etc.

Physically a source component is defined in terms of:

1. A list of *children* components defining a *whole-part composition* — for example "PriQueue" and "NetBiosIntf" are children of "ComPack" in figure 1.
2. A list of components which this component *depends-on* defining a *reference composition*. For example "NetBiosIntf" depends on some utility functions in the general purpose "Utility" component.
3. A list of *files* associated with the component. Typically "ComPack" will have some files defining the interface- and implementation of the class like e.g. "compack.h" and "compack.cpp" in a C++ setting. As such files are *attributes* of the source component.
4. A list of *annotations*. Annotations are placeholders for associated attributes as outlined in the previous section.

Evidently this is an object-oriented definition of a source component: It has attributes and participates in whole-part- and reference compositions. It is therefore natural that source components may act on *messages* like for instance "compile", "check-in", "sum all logged staff-hours", etc. This notion is the basis of the direct manipulation interface of Ragnarok described in section 7.

The distinction between children components and components depended upon is of course a design decision. For instance the PriQueue component in the figure is a child of ComPack because it is a very specially optimised queue strictly created for use in ComPack. However if a more general priority queue is developed later which meets the demands of ComPack then this component may vanish and component ComPack could instead depend upon this new general component, which would probably be a child component of "Utility".

Note that there is no explicit reference to the parent component though everyone except the root of course has a parent. This is because the context-preserving approach to software configuration management outlined in section 5 assumes that a component does not rely on information in its parent or ancestors in general. I will deal with this aspect later.

4.5 Graph interpretation

The definition of the physical representation of a component suggests an interpretation in terms of a *directed graph*. The nodes in the graph are components while edges connect components. Edges comes in two flavours, namely the ones stemming from the child-list (whole-part composition), and the ones from the depend-list (reference composition). Throughout the figures in this report the parent to child edges will be drawn by solid lines with the arrow head pointing to the child, and dashed lines will be used for edges from a component to the one it depends upon[6]. Please refer to figure 1.

The graph for the total application I will denote the *component graph*.

4.6 Annotations

Annotations are associated with components. Some attributes, however, can be inferred indirectly from the hierarchical structure.

An obvious example is individual bugs reported on a component. To get the full picture of bugs pending and fixed for, say, ComPack we can simply merge the individual bug lists for ComPack, PriQueue, and NetBiosIntf, i.e. the components found by traversing the child edges from ComPack. The same scheme can be used to produce a complete checklist of quality assurance items to check before releasing a component.

Another example is summing staff-hours for tasks associated with a component. Here we could sum all logged hours in the graph defined by the child edges. This could then be used to calculate deviations from planning estimates.

4.7 Discussion

The outlined model assumes that a one-to-one mapping between the components of the logical design architecture and their physical represention is possible. However this is not always possible — for instance two interdependent classes cannot be defined in separate files in the Mjølner BETA fragment system [Madsen94]. This is of course unfortunate but can be handled in a pragmatic way, for instance by having a single interface file for the two classes in their common parent component.

One could simplify the above model by eliminating child components all together and consider child component just to be components depended upon. However the parent-child relation define the *hierarchical structure* of an application and is an important mechanism as it provides different *levels of abstraction*. This is important for project management and used intensively in the user interface to provide overview. It fits with what is considered “good programming practise”, the idea of “programming by contract” [Meyer88] and “incremental development” [Goldberg et al. 95].

The *contractual approach* used in IStar [Dowson87] resembles the outlined approach. However IStar formalises the interfaces between the components in their hierarchy which can only be changed by renegotiating the “contract” between the components.

5 Context-Preserving Software Configuration Management

This section outlines an approach to software configuration management denoted *context-preserving software configuration management*.

Though parallels between this approach and that of Bendix [Bendix95] and others can be made (see discussion in section 5.7) the motivation grew out of an analysis of the work situation of software developers and managers, more than from theoretical considerations.

Generally the terminology defined by Tichy [Tichy88] will be used unless otherwise stated. This includes using the Dewey notation for version identification throughout all examples and figures.

The approach is based on two things. The first follows quite naturally from the discussion in section 4, namely that *a source component is a natural unit for version control.* The component is informally the

[6] Pragmatically speaking if there is an edge from A to B then “Files in A imports/includes files in B”. Refer to figure 1.

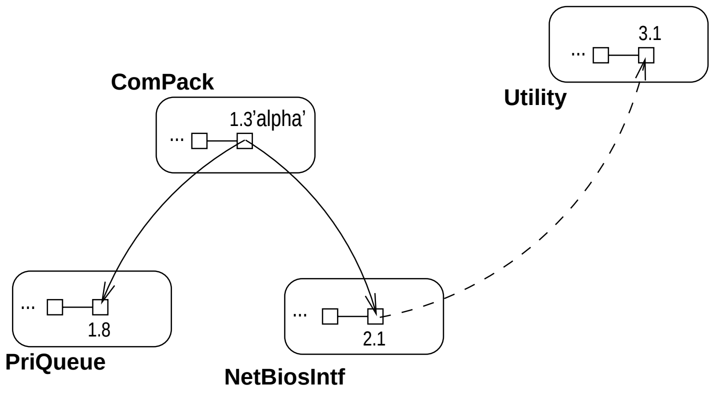

Figure 2: This example illustrates the context of version 1.3 (having a symbolic name "alpha") of ComPack. Version groups are depicted as rounded boxes with the version graph inside; boxes represent source-components-in-the-storage. The context of ComPack version 1.3 is PriQueue version 1.8, and NetBiosIntf version 2.1. NetBiosIntf version 2.1 in turn specify version 3.1 of the Utility component. To avoid too many arrows only the context of ComPack 1.3 is shown.

"unit" of discussion, documentation, reuse, etc. within a team and range from a single class to the whole application according to the level of abstraction. Therefore it is quite natural to consider *version groups of source components*. Version groups of source components are persistent objects stored in a database. This database is denoted the *project component storage*. As is common the space where modifications are made by developers is denoted the *workspace*.

To clarify terminology a distinction is made between *source components*, which are the entities developers change and work on in their workspace, and *source-components-in-the-storage* which reside in the storage and are immutable; they can only be read or deleted once created. A source component version group is a set of source-components-in-the-storage connected by the relations "revision-of" and "variant-of"[7] as defined in [Tichy88].

This will allow the team to speak in terms of "I have fixed the bugs you reported in version X of the graphics component" or "I've added extra functionality in version X of the meteorological domain knowledge component".

Secondly let us extend the common notion of a *version*:

Definition:

A version of a source component specifies the relevant context in which it was created.

As stated in the last section a source component contains lists of child components and components otherwise depended upon. In order to fully specify context these must now pinpoint the *exact version of each component at the time of creation*; not just specify the component version group. This is depicted in figure 2.

Pragmatically speaking any version of any component is able to completely recreate the (relevant) context that existed when the version of the component was created.

As an example consider the ComPack communication component shown in figure 2. A small team is responsible for creating ComPack. A "stub" version of the ComPack component is quickly made to allow the clients of ComPack to implement and test other aspects of the system. The ComPack team then implements a fully working component creating the two components PriQueue and NetBiosIntf in the process. Having passed internal test procedures the team wants to release the component for alpha-testing. Thus they create a new version of ComPack, 1.3, with symbolic name "alpha" (see figure), and publish the availability to the clients of ComPack. The clients, in turn, only need to "check out" the new "alpha" version of ComPack to retrieve the full context of the component including the added components PriQueue and NetBiosIntf as well as any changes to the Utility component.

The criteria of *relevance* is important and must be defined by the development team. In a project spanning several years it may be relevant to include the versions of the operating system, compiler, linker, etc. as the relevant context; in a smaller project these may be considered constant and handling of these be made by a backup scheme instead.

[7] The notion of *variants* is discussed in more detail in section 5.6.

5.1 Motivation

Why the demand for preserving the context?

One major achievement is, in my opinion, that the implementation of a version gets closer to an intuitive notion of the "version of a component".

In RCS [Tichy82], CVS [Berliner90], the three dimensional graph model of Bendix [Bendix95] and generally approaches based on selection rules for creating configurations the problem is, that the only way to preserve the context of a component is by *tagging* i.e. invent tags which must be associated with every file/object used in a configuration[8].

Say you want to make sure that you can recreate version 1.3 of ComPack. In RCS and CVS the only real mean you have is to tag all files associated with the ComPack component (and possibly also files in the company-wide software library) with a symbolic tag, here for instance "alpha".

Tagging poses three basic problems:

The first two are pragmatic. First of all you have to remember it! Forgetting to tag everything leaves you with some detective work in order to reconstruct the release. The second problem is that tags are usually global. Thus tagging files in the company library with the tag "alpha" would probably give a conflict because someone has already used that tag name. Also the library files will very rapidly contain a myriad of tags.

The third is conceptual — what you actually do is to tell all files that they belong to a certain version of a component, here "alpha" of ComPack. But this is *not* my intuition of version "alpha" of ComPack; there is no central notion of "ComPack in alpha release". In contrast "alpha release of ComPack" is an entity that designers and even more so its clients discuss and perceive as a *whole*: "Birger has reported a bug in version 1.3 of ComPack", "Your problem will be fixed in version 1.4 that I'll finish later today" etc.

The context-preserving SCM model avoids these problems. The version of the component can recreate the context it was created in. I.e. conceptually this unique version embodies "alpha release of ComPack" thus being very close to my own and my clients understanding. Also the symbolic name "alpha" is associated with this version; but it is local to ComPack avoiding polluting the name-space and name clashes.

Another important aspect is that of maintaining an overview of a large software application. As outlined in section 3 comprehending and navigating in a large software system can be problematic. But having a large software system under version control in essence adds yet another dimension to this complexity because not only the myriad of source files has to be understood in terms of their relevance to the overall picture; now also the myriad of *versions* of each file has to be taken into account. This approach significantly reduces complexity as there are no selection rules involved, and the component version itself knows and specifies which versions of other components it requires.

A more pragmatic aspect is *traceability*. Traceability of customer releases is extremely important, but also to provide historical data on process metrics, as Goldberg points out [Goldberg et al. 95, Chap. 6]. Context-preserving SCM ensures that the context of i.e. management data is not lost which could render the data useless.

5.2 Context

Let us return to the workspace of a developer. How can we define the context of a component A so that a check-in procedure can store the context as part of the new version of A in the project component storage?

Loosely speaking the context of a component A is everything that A directly or indirectly depends upon. Let us first define these relations:

> **Definition:** The relation *directly-depending-on* is a relation between two components. A component A is directly-depending-on another component B if there is an edge $[A, B]$ between A and B in the component graph. (That is: B is listed in either the child- or the depend-list of A).

> **Definition:** The relation *depending-on* is a relation between two components. A component A is depending-on another component B if there is a *path* between A and B in the component graph. (That is: There exists a set of components $\{C_1, C_2, \ldots, C_n\}$ such that A is

[8] It is difficult to imagine version attributes that distinguish 'release 1.0.0' from 'release 1.0.1' — except an attributes that explicitly states this fact.

directly-depending-on C_1 and C_1 is directly-depending-on C_2 and ... and C_n is directly-depending-on B).

The perhaps easiest way to describe the context of A is to express it in terms of a sub-graph of the component graph:

Definition: The *context graph* for a component A is a sub-graph of the component graph for which it holds that:

1. A is the root of the graph.
2. The graph contains all components C_i for which it holds that A is depending-on C_i.
3. The graph includes all edges $[C_i, C_j]$ for which it holds that both C_i and C_j is in the component set defined in point 2.

Please note that according to this definition the *parent* of a component A is *not* part of the context for A. If we had indeed included the parent of a component as part of the context then the context graph for any component would always equal the whole application component graph; which would have made the model a somewhat elaborate backup model instead of a SCM model.

5.3 Modification

In a development process we constantly modify bits and pieces of the components in our workspaces. We can distinguish between two kinds of modifications:

Definition: A component A is *directly-modified* if changes are made in any part of A i.e. changes made in any file associated with A, or modifications are made in the child- or depend-lists or the associated annotations.

Thus adding a new child-component or changing a dependency relation is just as much a direct modification as modifying one or several of the files associated with a component.

However a component may also be modified *indirectly* when for instance a child component or a component depended upon is changed:

Definition: A component A is *indirectly-modified* if a component C in the context graph for A has been directly-modified.

5.4 Preserving context

We want to preserve the exact context of a component A when creating a new version of A.

The check-out/check-in round trip is depicted in figure 3. The project component storage is asked to check-out version 1.3 of component ComPack. The reason is that we want to re-engineer component PriQueue into a general purpose priority queue. Thus in the workspace direct modifications are made to component PriQueue, ComPack (because PriQueue has been removed from the child-list), and Utility (PriQueue is now a child of Utility)[9]. On the figure components that are directly modified are shown with an asterisk, indirectly modified components with an asterisk in parenthesis.

Now checking-in ComPack again proceeds as follows:

1. Calculate the context graph G for ComPack. (This is simply the graph in the workspace in figure 3).
2. Prune the context graph G for all components that are not directly- nor indirectly-modified. (There are none in the example).
3. Enumerate all components C_i in the context graph in depth-first order and create a new version of each C_i with a new version identification in the project component storage.

Hence in the example a new version of PriQueue is created first with identification 1.9. Then a version of Utility is created referring to PriQueue 1.9. Next a NetBiosIntf version is created (because it is indirectly-modified) with version 2.2 stating that it depends on Utility 3.2 and finally a version 1.4 of ComPack is created.

Other developers can now be notified of the change and simply check-out ComPack 1.4 which would automatically create the new PriQueue component and setup the new dependencies.

[9]There is no need to set up a direct dependency between ComPack and PriQueue because it is implicit via NetBiosIntf; however in praxis it is probably better to state it anyway to secure against future restructuring.

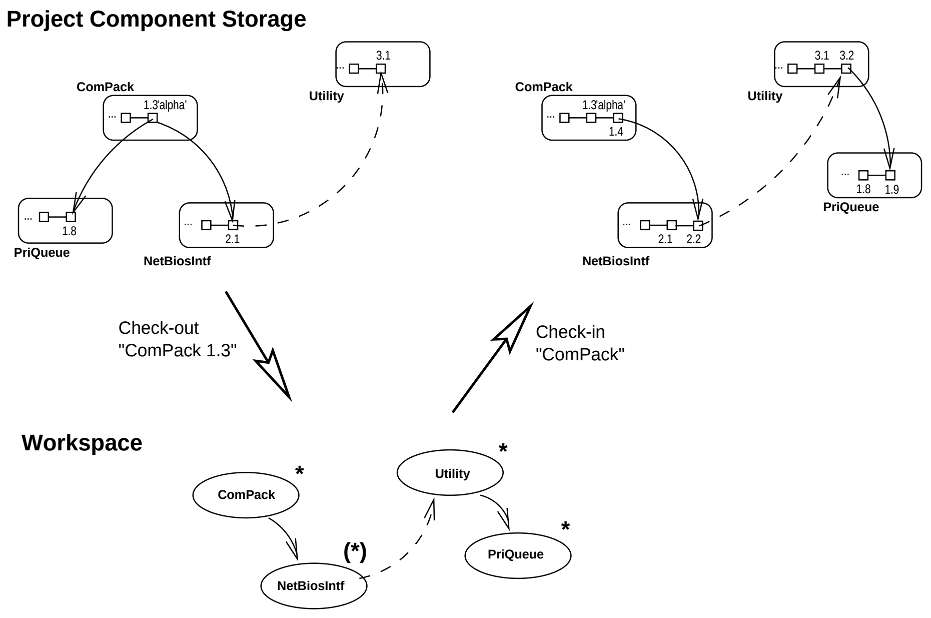

Figure 3: An example of a check-out/check-in round trip. ComPack 1.3 is checked-out and a major restructuring is made. Then ComPack is checked-in creating new versions of all components in the context graph. Directly modified components are marked with a *, while indirectly-modified are marked with (*).

5.5 Generic configurations

Another way of stating the context-preserving requirement is that generic configurations are not allowed in the project component storage, only baseline configurations.

However we of course have to be able to operate with generic configurations as well but it is only allowed in the workspace. Here developers can select versions of components in order to create new configurations of the system (or parts of it).

Presently I have not investigated the selection problem in much detail but the outlined model does of course not get in the way of incorporating strong selection engines to produce configurations in workspace based on selected attributes of the component. Furthermore my proposal of having *all* project management data as annotations of components may even provide new opportunities.

In the Ragnarok prototype one important selection rule is however available as it is necessary in order to merge independent development efforts. This selection rule (denoted “upgrade” in the RCM tool described in section 8.2.1) basically performs a check-out of a component but does *not* overwrite components in workspace if either:

A: The component in workspace is marked as *directly-modified*.

B: The component in workspace is *newer* than the one that would otherwise have overwritten it.

Rule A allows you to upgrade without first checking-in components that are being worked upon. Rule B allows merging independent work.

For example consider two developers working on different aspects of ComPack. Developer A is in the process of making the restructuring shown in figure 3, but has not performed the check-in. Meanwhile developer B has enhanced and optimised component NetBiosIntf and checked-in to create version 2.2. Now if A tries to check-in ComPack it will create a branch in the version group of NetBiosIntf because the check-in needs to create a new version of NetBiosIntf but there is already a version 2.2[10]. This is not the intention of developer A. The way out is to issue a command to “upgrade” NetBiosIntf in A's workspace to version 2.2. Rule B will ensure that the changes made to Utility (including the newly added PriQueue component) are *not* overwritten though version 2.2 of NetBiosIntf as created by B specifies version 3.1 of Utility. Now A has merged B's effort into his own and may create a new ComPack version without creating branches.

[10]The RCM tool described in section 8.2.1 will warn in such a situation and allows the user to cancel the operation.

5.6 Variants

Winkler introduced the terms *program-variants-in-the-small* and *program-variants-in-the-large* in [Winkler et al. 88]. Using branches in the version groups of components supports the notion of program-variants-in-the-large.

However it is often program-variants-in-the-small that is most useful in praxis because variations often occurs at the statement level, not at the component level. Examples are IO related classes having a stub variant for testing without hardware, code generators for different platforms, conditional assertions, debug, etc.

Program-variants-in-the-small are usually handled at the (near) language level by for instance the C preprocessor or the BETA fragment system.

Ragnarok supports program-variants-in-the-small indirectly by the component concept: For instance in BETA different variants are often represented by different fragment groups ("body files") which are naturally part of a single component, thus being treated as a whole. The concept of derived components (section 4.3) also supports managing the different translated components.

5.7 Related work

The model outlined here resembles the *three dimensional graph model* proposed by Bendix [Bendix95, Chap. 6]. Bendix proposes the model in order to integrate configurations and versions: In effect setting the dependency information under version control.

However in Bendix's model dependencies goes from a single version in a version group to a *whole* version group. Thus in figure 3 the edge from ComPack version 1.4 would reference the *whole* version group of NetBiosIntf, and not the *specific* version 2.2.

This does not permit us to state the restructuring made between 1.3 and 1.4 of ComPack. This means that preserving context must be handled by the selection mechanism. As already mentioned in section 5.1 I think this leads back into the unfortunate use of "tags".

Explicit configuration objects are used for instance in DSEE [Leblang et al. 87] and Mjølner ORM [Gustavsson90]. Tichy emphasise that a software object can be a configuration as well and uses it in his AND/OR graph [Tichy88]. A baseline configuration object is a better solution than tags but still has some of the deficiencies: We must remember to create one, put it under version control, and it is still just a schema to reproduce a context; it is not a self-contained object that embodies for instance "ComPack version 1.3".

5.8 Discussion

The check-in algorithm sketched above ensures that the full context of a new version of a component A is stored. This is done by cascading the storage operation through-out the modified parts of the context graph of A. Thus new versions of a component somewhere in the context graph may actually be stored even though it has not been directly modified itself—the new version of NetBiosIntf in figure 3 is such an example.

At first this may seem like an unfortunate and even undesirable effect. However I find it quite natural because it allows you to uniquely identify *parts* of a major release. In the example we can thus discuss the version of NetBiosIntf that is part of component ComPack version 1.4; and extract and manipulate it separately.

The outlined context-preserving SCM model has chosen the link to parent as the "cut-off" point in defining context. This came quite natural from common design principles of combining separate classes/modules into aggregate structures where the individual component should know nothing of the context they are used in.

However for instance block structured languages like BETA pose a problem. Conceptually block structure fits nicely within the parent/child (whole-part) relation between components. However changes in the parent (outer block) of course may seriously affect behaviour in a child (inner block); but according to the definition of context creating a new version of the child does *not* preserve the parent context. The problem can be avoided by pragmatic measures so that the context can be preserved but it is undeniable that this constraint affects the (physical) structuring of the application. This is however not uncommon in many CASE tools.

6 Software Configuration Management and Collaboration

So far the context-preserving SCM model has been introduced without considering what happens in a team. As outlined in section 3 I find collaborative issues extremely important.

I see software configuration management as a potential candidate for mediating collaboration and collaborative awareness.

6.1 Basic problems

There are many ways to delegate responsibilities in a software project, some of them are described in [Goldberg et al. 95, Chap. 12]. In most of the team models mentioned there are tasks that are not the sole responsibility of a single person. This may of course also be dictated by the education and background of the team members and the concrete project.

I see a basic conflict between the perspective of the *project* and of the individual *developer*.

From the project point of view *overview, control, traceability, and quality* are crucial. That is, making sure developers have a common, well-defined, goal, and ensure this is reflected in the SCM structure: The global SCM structure should not be polluted by all sorts of irrelevant, intermediate, versions.

From the developers point of view *flexibility* is important. A developer needs to experiment and in this process create intermediate versions as “safe ground”; also if a “showstopper” is found in the code of another developer, it is annoying and inefficient to have to wait for a slow bureaucratic management to grant you permission to fix it yourself.

A development environment should try to provide flexible solutions that makes it a decision within the project team how to balance the needs for control versus flexibility, as opposed to imposed by the environment.

6.2 Basic change mechanisms

Traditionally there are two mechanisms for handling the case where two or more developers need to change the same object:

- *Lock-modify-unlock*: This mechanism is basically a *binary semaphore* on the object. In order to change an object you must *lock* the object first, and only one person at the time is allowed to do so. As long as a person holds the lock no one else can change the object. This mechanism serialises all access to an object. A well known tool using this scheme is RCS [Tichy82].
- *Copy-modify-merge*: In this approach several persons are allowed to modify an object concurrently but they all operate on a copy. When done each must merge their changes into a new version (This step can not be done concurrently). CVS uses this approach [Berliner90].

Both have benefits and drawbacks. The benefit of the lock-modify-unlock mechanism it that conflicting changes are avoided, but serialised access may provide a bottle-neck in some situations. The copy-modify-merge mechanism has the problem that syntactic conflicts are discovered automatically but not semantic conflicts: If for instance one developer changes the interpretation of a variable this may still not give any syntactic conflicts though the code logic is changed.

6.3 Access control

Regardless of team-structure a project of reasonable size needs to delegate responsibilities and with responsibilities comes the question of *controlling access* to components.

In my opinion, people sometimes confuse the issues of *managing creation of new versions* and *controlling access to components*. The “copy-modify-merge” mechanism may falsely give the impression that “we can allow everybody to access everything because it will be merged nicely”. Of course this is not true; developer A would not want some outsider B to introduce a “fix” in his components just before a milestone without his prior acceptance[11].

[11] At least I wouldn't!

6.4 A flexible proposal

I propose the following four mechanisms should be supported:

- *Hierarchical access control:* Access to components of the system should be controllable.
- *Provide a "copy-modify-merge" mechanism:* To allow concurrent work on a single component.
- *Version control in workspace:* This idea proposed by Bendix [Bendix95] is very valuable in order to create the flexibility a developer or sub-team needs to experiment without polluting the project global SCM structure with irrelevant versions.
- *Powerful constructs to support collaboration around a single component:* This range from little support when developer A just uses something in B's component to the intensive support needed when A and B are working concurrently on the same component.

6.4.1 Hierarchical access control

Each component should contain a list of access rights for individuals and groups within the project team. A hierarchical system akin to UNIX with rights for global, group, and individual access to read, modify, create variants, and so on could be envisaged.

Access control also provides some security against deliberate malicious acts [Vessey et al. 95].

6.4.2 Copy-modify-merge mechanism

Concurrent work on the same component should be allowed[12]. Before a new version can be entered into a version group all conflicting modifications must be (manually) solved.

Besides active use of access control to prevent the "everybody is changing everything" situation, I suggest *two* kinds of check-out mechanisms namely a default *check-out for read* and an explicit *check-out for modification*. Having this distinction serves several purposes:

- I believe that by taking an explicit action in order to modify components, developers are more inclined to think carefully about what and why they want to change it.
- Version groups are not polluted by irrelevant versions stemming from an accidental extra space inserted while viewing some information.
- When issuing a request for checking-out a component for modification the environment gets valuable information to provide collaborative awareness: The requester can be told if other developers already are modifying the component, and these other developers may be notified that a new "modifier" has entered the stage.
- Explicit check-outs for modification facilitate a modification request driven approach as suggested by Tichy [Tichy88] and used intensively in for instance Aegis [Miller95].

6.4.3 Version control in workspace

Bendix proposes *fully supported recursive workspaces* in order to provide version control in the workspace [Bendix95, Chap. 5]. The proposal is based on identifying the current lack of support for the workspace concept, and argues that this approach allows experiments without polluting the repository with intermediate versions. Clearly this is a way to give developers flexibility and freedom while maintaining the overview and control aspect important seen from the management perspective.

Cagan terms the versions in workspace *micro-version* in contrast to the global *macro-versions* in [Cagan95]. I will generally denote them *project* and *local versions* to reflect the usual visibility: local version are "safe ground" during day-to-day implementation, project versions are deliberate steps and milestones towards the project goal.

6.4.4 Collaboration on a single component

I identify three typical collaboration scenarios centred around a single component:

- *Using a component:* This is the situation where one developer A relies on a component that another developer B has the responsibility of. A does not know the inner workings but wants to be aware when B has made improvements, added functionality, etc.

[12] This may especially be necessary as components may contain several files.

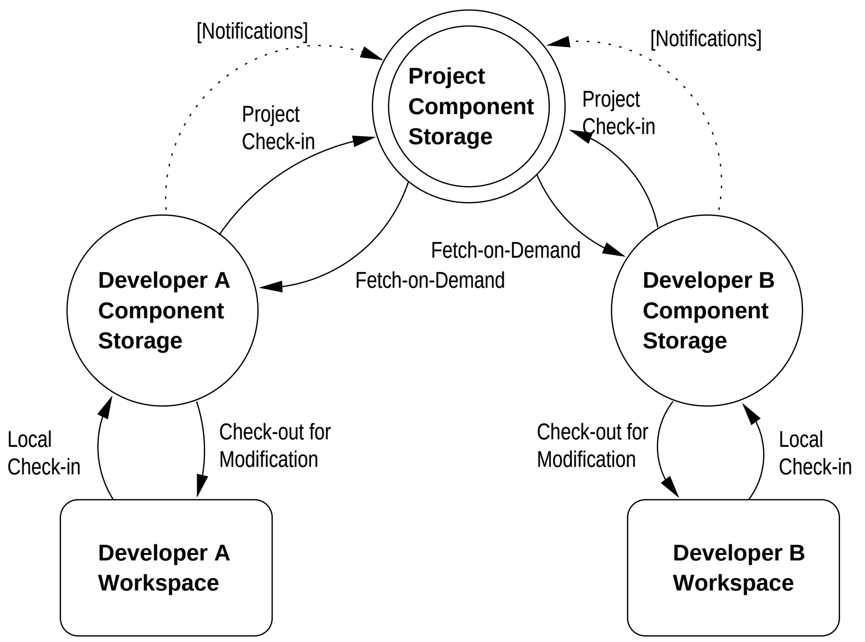

Figure 4: Collaboration and storage model.

- *Parallel work on a component:* Here A and B work in parallel on different aspects of the same component but can largely work independently with a need to get updated on each others work every now and then.
- *Concurrent work on a component:* Here A and B work on the very same aspects and need to work and share ideas concurrently.

The first case seems trivial to support. Basically I think this can be handled by some notification mechanism. When B submits a new version to the project component storage all developers depending on this is notified. Then they may include the new version in their local work.

The third case is an active area of research with many interesting contributions, see for example [Olsson94, Prasuan et al. 93, Smith et al. 95]. The general approach is to work on a *shared representation* so you can see all modifications on-line often combined with some tele conferencing abilities allowing you to talk together. Presently no support for this case is envisaged in Ragnarok.

The second case is an in-between scenario. Here I propose a *synchronisation mechanism* that allow A and B to merge their local versions from time to time in order to form a new common basis to continue work from. The idea is described in more detail in the section 6.5.4.

6.5 Model

I suggest a model in which there is a single *project component storage* and many *developer/local component storages*. These are on-line connected in some way.

Local versions are created and stored in the local component storage; however a notification about their creation is propagated to the project component storage. The bodies of the versions are not automatically transferred. Still another developer can see that a new local version has been made and may fetch a copy, given he/she has the rights to access local versions of the creator.

Components just for viewing/translation are (usually) not kept locally; for instance common libraries will typically be stored centrally and without general permission for modification hence there is little sense in creating local copies in the local storage nor local workspace[13].

Components that are going to be modified will be made available as copies in a local *workspace* for editing, tool manipulations, etc.

The idea is depicted in figure 4.

[13]Of course one may argue that a local copy *does* exist in a viewers memory.

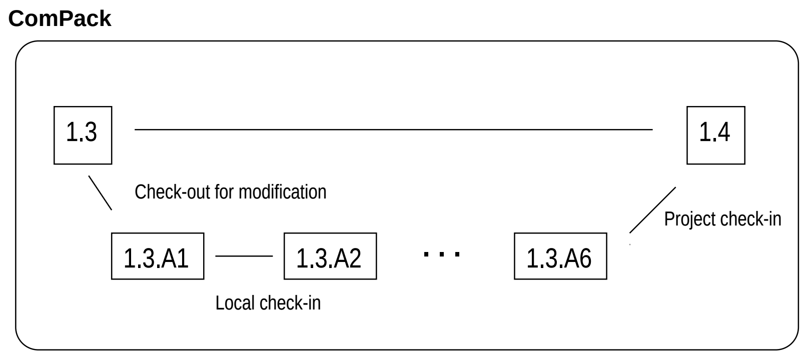

Figure 5: Example of local- and project check-in. A developer "A" check-out ComPack version 1.3 for modification, goes through a number of iterations using local check-in's until finally the component is checked-in into the project storage.

The use of the storages should be *transparent.* Ideally the developer should not care about a "local workspace" where local copies reside; instead files and annotations associated with components should be made available from anywhere in the system—from ones own local component storage, the project component storage, or perhaps the local storage of another developer—in a "fetch-on-demand" manner. This of course extends to the derived components allowing faster compilations because a certain component needs only be compiled on a single site -after which it may be used project wide.

This way we avoid the conceptual and practical problems when having multiple copies of stable, common, libraries in local workspaces.

6.5.1 Check-in

As shown in figure 4 having several storages requires different check-in operations.

Local versions are made by a *local check-in*. As noted this action produces a notification about the existence to the project component storage as well as performing the actual check-in action in the local storage.

A line of local versions usually culminates in a *project check-in* thereby providing an "official" new version seen from the project point of view. Typically such versions must meet project requirements in terms of fulfilling a list of goals, adhere to demands on quality, etc. A project check-in may automatically trigger events such as regression tests, etc.

Figure 5 outlines the idea.

A situation can arise where another developer has already project checked-in a newer version when a local version is about to be promoted to a project version (for instance if a developer "B" has already created a version 1.4 of ComPack in figure 5). In this case a merge is performed and conflicts must be resolved. Because of the context-preserving nature this involves merging everything in the context graph and resolving conflicting changes here as well.

The right to perform a project check-in is of course also controlled by permissions.

6.5.2 Check-out

A *check-out for read* issued by a developer merely informs the local storage that a specific version (and its context) is the one to use; thus if the developer wants to view a specific source file or some annotation the relevant data stored as part of the version is provided. No actual "copy" operation from the storage to a local file is made.

A *check-out for modification* on the other hand must create an editable instance for the developer to work on in a private workspace of the developer. I envisage a version identification scheme where local versions indicate person (or group) creator along with a serial number as exemplified in figure 5 and 6.

6.5.3 Awareness

Because the common project component storage is always notified of changes and components checked-out for modification developers could be notified of potential new versions on a subscription basis. Clas-

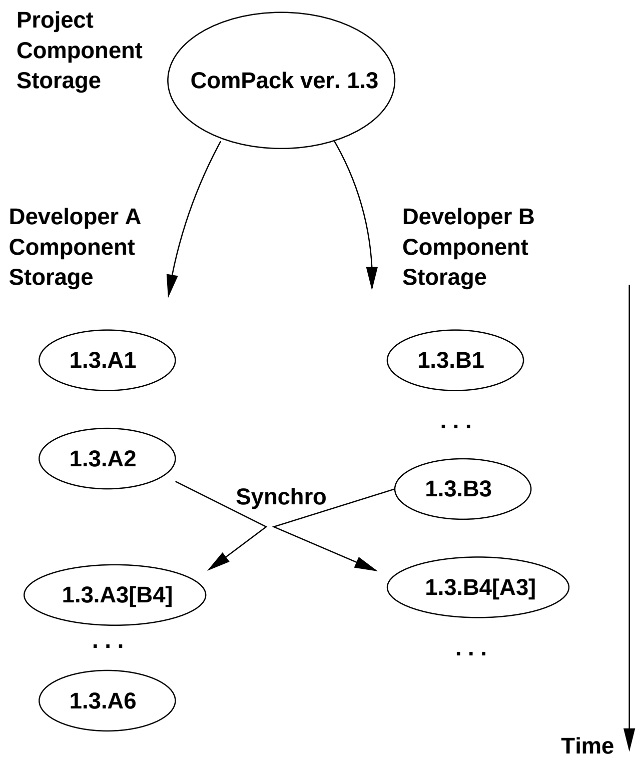

Figure 6: Synchronisation scenario.

sic notification mechanisms like e.g. e-mail could be considered in a command-line implementation; other, less disturbing, mechanisms will be discussed in section 7.

6.5.4 Synchronisation

The idea is depicted in figure 6.

Two developers A and B work independently from a common version of a component in the project component storage. They create local versions in their local developer storage, work independently and when the need to exchange modifications arises, they perform a *synchronisation*. This operation merges the two versions forming two new but identical versions in the local component storages. Work can then be resumed independently until a new synchronisation is made or project check-in performed.

The synchronisation needs not be two-sided. One developer, A, may wish to merge the changes of the other, B, into his own without affecting B's local storage.

6.5.5 Distributed collaboration

The suggested model has another important facet namely that there is not a great demand for high data transmission bandwidth between project- and local component storages. Only the information about a new local version (its identification and location) is generally communicated, not the full body of the version.

6.5.6 Off-line connections

The connection between project- and developer component storage needs not be on-line all the time. Then of course notifications of new local versions can not be broadcasted. However facilities to cope with this is important because the inevitable “Eerh, well—I'll fix it at home” syndrome in real world projects. However this poses no great problem; when the local storage gets on-line again the additions and modifications made are simply transmitted to the project storage. A project check-in requires an on-line connection though.

Off-line operation requires some operation that transfers everything needed to work independently on a given context from the project to the local storage. Here we have another benefit from the context-preserving SCM model namely that this is already defined for any component one may choose.

6.5.7 Recursive workspaces

The only difference between a local and a project component storage is that information about new versions is propagated from the local to the project component storage whereas the project storage does not propagate anything. However a local component storage could just as well specify *another* local storage as the component storage to notify (Notifications will still be propagated all the way to the project storage). This way recursive workspaces could be supported.

The synchronisation scenario in figure 6 could alternatively be realised by developer A and B sharing a common, but local, component storage where the synchronisation could be made.

6.6 Discussion

Magnusson et al. propose fine-grained version control to mediate collaborative awareness using a technique called *active diffs* [Minör et al. 93, Olsson94]. The underlying idea is to use visual clues that display the concurrent work of others on the same document.

The model outlined here can also provide awareness through version control when combined with the visual mechanisms described in the next section.

Prasuan et al. outlines a collaborative software engineering environment, Flecse, with special focus on (geographical) distributed development [Prasuan et al. 93]. All shared material is stored at a single site, and a distributed RCSTool (an RCS front-end) provide concurrency control. Though it provides awareness (changes are reflected on local sites concurrently) there is no provision for version control in workspace and the lock-modify-unlock mechanism serialise access. Partial results may be shared (akin to synchronisation) only by starting a concurrent editing session. Generally Flecse is geared towards the case of concurrent work on a single component; however it is my experience that the scenarios of synchronisation and simple use of components are more common.

7 Spatial Metaphor Interface

The user interface of Ragnarok is based on three cornerstones: A *spatial metaphor* combined with the visual formalism of *maps*, and *direct manipulation* of components.

7.1 Spatial Metaphor

Spatial metaphors have been investigated especially in the context of *hypertext*. Spatial metaphors are seen as one way of avoiding for the often reported sensation of “getting-lost-in-hyperspace” people experience after following a few hyperlinks in a hypertext document: Not knowing where they are, how they got there, how the displayed material relates to the rest of the text, etc.

Finding our way in a new large software system often leave us with the same sensation; and as the systems grow very large, even the systems that we have partly designed ourselves becomes difficult to overview.

7.1.1 Psychological foundation

Psychological theories emerge and disappear just as in other scientific fields, e.g. physics. It is therefore difficult and dangerous to speak about one *true* model of how humans navigate. However today it is generally accepted that humans build up a *cognitive map in the mind, which is the analogue to the physical layout of the environment*. This was first postulated by Tolman in 1948.

When developing such a cognitive map (for instance when we find ourselves in an unknown town) it is generally accepted that the acquisition of navigational knowledge passes through several stages.

According to this model we first represent knowledge in terms of *highly salient visual landmarks* in the environment such as remarkable buildings, statues etc. Thus we recognise our position relative to such landmarks and build up knowledge about their relative positions.

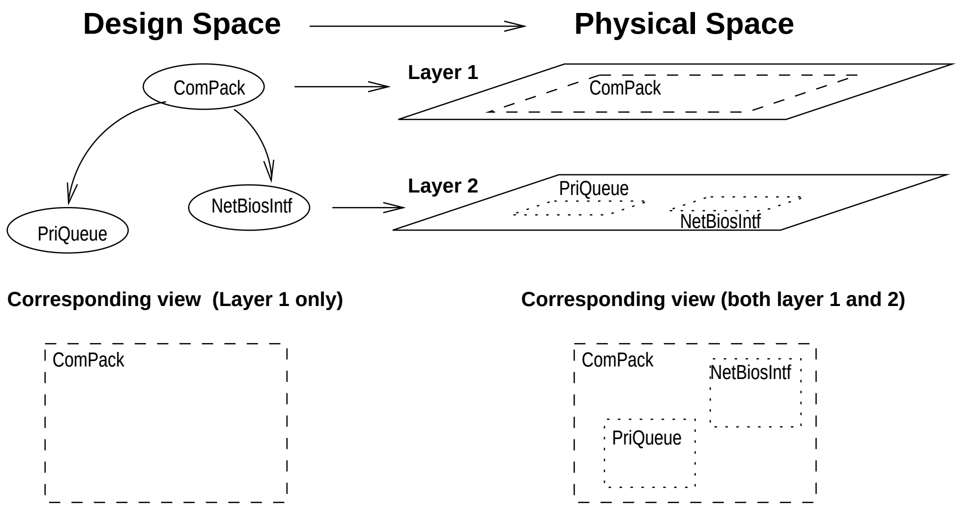

Figure 7: Abstraction layers. Different levels of abstraction are represented by planes where landmarks are located. Below the corresponding view the user sees when choosing to see layer 1 only and both layer 1 and 2.

The next stage is *route* knowledge where we can navigate from one point A to another B using our knowledge of the landmarks we pass by as we move: I.e. something like “Turn right at the church and continue until you see the railway station, then turn left...”. Still the route chosen may be non-optimal.

The third stage is the acquisition of *survey* knowledge which is the fully developed cognitive map Tolman speaks about. Here we can plan journeys precisely and describe the position of locations within the environment.

The overview presented here is due to [McKnight et al. 91].

Whereas the second and third stages are less relevant in this context as they are mental processes, their application relies on two important aspects of the first stage: Salient visual landmarks and the, for a physical world, obvious fact that the landmarks can be assumed not to move.

7.1.2 Landmarks represent components

The underlying idea in the user interface of Ragnarok is:

> *Software components are represented by* visual landmarks *having a unique position and extent in a physical space*.

This way focus is shifted from remembering the exact *name* of a component to remembering an approximate *position*.

Presently Ragnarok uses a simple rectangle to represent a visual landmark and the physical space is the two-dimensional plane.

All manipulations of components are mediated through the visual landmarks as outlined in section 7.4

7.1.3 Component hierarchies as visual nesting

At described in section 4 the hierarchy defined by the parent-child relation in an application can be viewed as different levels of abstractions.

We can visualise this by *visual nesting*. For instance component ComPack has child components PriQueue and NetBiosIntf. This is represented by the landmark ComPack having the landmarks PriQueue and NetBiosIntf nested within its boundaries.

As depicted in figure 7 we can visualise the different levels of abstractions (defined as the distance from the root in the component graph) as defining *planes* in which the corresponding landmarks are located.

These planes I denote *abstraction layers* as they represent a layer at a certain level of abstraction.

This interpretation allows the user to choose how much detail is wanted. For instance a user may wish to see little detail and decides to view layer 1 only. Then only the ComPack landmark is visible. If she chooses to see more detail she can specify to view both layer 1 and layer 2 which would display landmark ComPack with landmarks PriQueue and NetBiosIntf nested inside.

7.1.4 Aiding comprehension

Nested landmarks provide some indication of software structure: The nesting shows the parent-child relationship of components.

However additional graphics can be used to show many other kinds of relations.

One of the items listed in section 3 was the wish for a "road map" showing the relations between components in the application. The Ragnarok prototype provides a (very) limited set of drawing primitives to allow unified method class diagram [Booch et al. 95] notation to be drawn in the abstraction layers.

This way the abstraction layers not only can depict the parent-child relation, but also associations, inheritance, roles, multiplicity, comments, etc., between components.

The frame used for landmarks can also be changed to denote classes, class categories, or class utilities according to the visual syntax of unified method.

Please refer to section 8 where the design of the Ragnarok prototype is described using the visual capabilities of Ragnarok itself.

7.1.5 Displaying annotations

The graphical rectangles for landmarks can be used to display annotations in or, in case of lengthy annotations like e.g. bug-lists or project management attributes, annotation summaries.

The size of the landmark of course determines how much information it is possible to display. However zooming in and enlarging a single landmark until it is large enough to accommodate all information may be too cumbersome. Therefore I think the best way is to provide summery information in the landmarks and then provide a way to spawn an independent *annotation viewer* in the form of a separate (text)window or dialog box.

7.1.6 Supporting highly salient landmarks

The present prototype of Ragnarok does not support the "highly salient" part of landmarks: Presently they are presented with identical looking rectangles.

This seems like a severe problem because real-world navigation is so dependent on this property: Imagine going through your home town where all buildings were reduced to identical looking gray boxes.

One idea is to associate distinct *icons* to landmarks. However this reduces the space available for other information and requires a large icon database or developers with good drawing skills.

Another interesting approach that I will try instead is to use the *background*. The idea is to superimpose landmarks on a easy identifiable background; for instance an image of earth. This would convey knowledge along the lines of "This landmark lies within Norway, so it must be part of the graphics component as I know it covers Scandinavia", etc. This approach has the advantage that components that are very alike but belong to different subsystems can still be distinguished easily[14]. The Terra-Vision project at Art+Com in Berlin [Joachim96] which uses detailed, zoom-able, satellite images of earth to convey a physical navigation sensation when browsing the World-Wide-Web, can be used as inspiration.

The background must of course use very pale colours in order not to disturb other more important information.

Initial use of the prototype suggests that the additional graphics (class diagram notation) provides adequate visual clues for small systems.

7.2 Map visual formalism

Harel advocates the use of *visual formalisms* by which (mathematical) problems can be understood and solved by visual means taking advantage of humans highly developed visual system [Harel88]. His main example is *graphs* which is a visual representation of a set of elements with some binary relation between

[14] As explained in section 2 we had ten systems where some components were pretty much the same and had the same names; and sometimes we got mixed up which of course was very unfortunate

them; this basic visual formalism can be used in many different contexts, for instance state charts, dance movements, and, as I have done, software project structure, and so on.

Nardi et al. argues in favour of using visual formalisms, like plots, panels, maps, outlines, and tables [Nardi et al. 93]. Referring to the work of Reisenberg [Reisberg87] on "perceptual knowledge", that is, knowledge that can be accessed only through interaction with external representations, they conclude that: *"In short, we have access to certain kinds of knowledge only when we* see *it."*[15]. Visual formalisms *"provides manipulable external representations with well-defined semantics"* allowing us to do just that.

Having an underlying spatial metaphor makes a *map* an obvious choice for representing data. Maps has a number of important, commonly understood, features which we can exploit:

- *Respects spatial relations*. The basic purpose of a map is to show spatial relations between objects.
- *Scale determines level of details*. Maps come in different scales; by varying the scale we can get overview or details depending on our problem.
- *Different aspects possible*. Maps can focus on different aspects of the objects they presents. Well-known examples are maps with emphasis on roads, political maps, terrain maps, etc.

7.2.1 Ragnarok maps

In Ragnarok a map is:

- Associated with a specific abstraction layer. A map shows landmarks in this layer and optionally in one or more of the layers *below* (denoted the *depth* of view). You can request a map to instead associate with the layer below (in essence a *zoom in* operation) or above (*zoom out*).
- Displaying a certain rectangular region of the abstraction layer. The map can display the region in different *enlargements*, like e.g. double or half size.
- Showing a specific *aspect* of the landmarks. Aspects are for instance *version control* aspect (show current version number and list of files of components—or maybe a graphical representation of the version graph like in Orm [Gustavsson90]), *comprehension* aspect (show class diagram notation), *management* aspect (project status summaries), *quality assurance* aspect (checklists with items marked as unchecked/checked), etc.

7.2.2 Overview map

Lots of maps displaying different areas of the underlying abstraction layers do not in itself create overview (more likely the contrary). One way to retain overview is to create a special map whose main purpose is to display the location of other maps. This is an approach often used in computer strategy games like for instance "Civilization" [Meier91].

Ragnarok has an overview map denoted the *world map* as it defines the "world" that the developer is working within; other maps are denoted *detail maps*. The areas that are displayed by detail maps are shown by *outlines* in the world map.

The world map is not a static, project defined map, but can be zoomed and scaled by the individual developer to display the part of the project that is relevant to him or her.

7.3 Ragnarok user interface

A snapshot of the Ragnarok user interface can be seen in figure 8.

The Ragnarok window is divided into four panes:

- The *world map* in the upper left corner. This is a map that shows *outlines* of detail maps superimposed on landmarks. One may zoom in and out, change the scale used, as well as what aspect to display. However the size and position of the map within the Ragnarok window is fixed so that one always know where to look for overview information.
- A *playground* right next to it in which may reside multiple *detail maps*. Detail maps display regions of the abstraction layers to a certain depth, and can be moved, zoomed, scaled, and resized. In addition their aspect can be changed. Detail maps are the primary medium for interaction with landmarks and hence components.

[15] A well known example is that of remembering how a word is spelled: writing out a couple of possible spellings is usually enough to pick the one that "looks right".

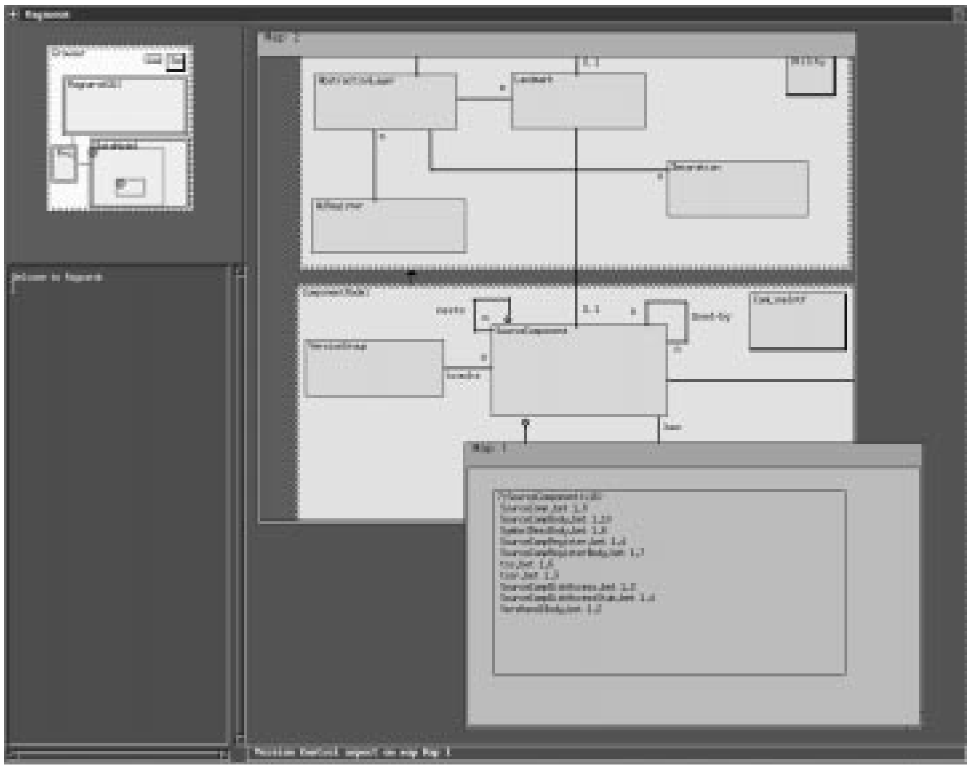

Figure 8: The Ragnarok prototype window. Detail map 1 shows the version control aspect of component "SourceComponent" (files with associated RCS revision numbers) while map 2 shows a comprehension aspect using unified method class diagram notation to show relations between "SourceComponent" and other class components. Please refer to appendix for a colour version of this figure.

- A *information window* in the lower left corner which is essentially a running *log* of important operations, typically the progress of check-in or check-out operations.
- A *status bar* below the playground. This is primarily intended to displaying warnings without disturbing the user with dialogues.

On the figure you can see two detail maps, map 1 showing version control properties and map 2 showing relations using unified method class diagram notation. In the world map the outlines of the two maps are shown (please refer to the colour figure in appendix). This way one can easily see where the maps display, relative to the overall context.

Detail maps and their corresponding outlines in the world map are of course fully synchronised so that any move or resize of one of them is reflected in the other.

New detail maps are created simply by dragging out a new outline in the world map using the mouse.

7.3.1 Moving maps

Ragnarok has no scroll bars on the detail maps. Instead the outline of them on the world map is moved (by grabbing the circle in the upper left corner). Though one may argue this is indirect in the sense that an outline is moved, not the detail map itself, it is definitely a much better way of moving your viewport in a two-dimensional plane than using horizontal- and vertical scrollbars. In the scroll bar case you need *two* actions to scroll to an arbitrary point which is both counter-intuitive and difficult to coordinate precisely. Dragging the map outline in the world map instead is intuitive, fast, and accurate.

When multiple overlapping maps are present you need a mechanism to bring one on top of the others. Most systems, like X11, MicroSoft Windows, etc. do this when clicking the window frame; which is impossible when the window is fully covered. Alternatively you can often leaf through windows, bringing each one to the top, one by one, until you get to the right one. In Ragnarok you simply click the aforementioned circle on the map outline, which is faster and a more direct method.

The radar-view of the Self 4.0 programming environment [Smith et al. 95, Maloney95] achieves some of the same goals of relating the windows position with respect to the whole and allows movement in the plane.

7.3.2 Avoiding cluttering

As mentioned in section 3 a design goal has been to avoid intermediate steps in navigations in the form of spawning new windows/maps requiring effort to tidy up the screen all the time.

In Ragnarok all navigational operations just changes the view in the detail map, i.e. zooming in and out, resizing, moving, etc., and all operations have fast short-key interfaces. This way the need for intermediate maps is reduced. For instance if you have a map showing some details and need to see the immediate context, you simply give one or two zoom-out operations (simply typing “o” once or twice), view the context, and then zoom-in again (typing “i” once or twice).

In case of annotation viewers spawned from landmarks they should have the same visibility as the landmarks: I.e. if a map overlays a landmark associated with a large annotation viewer, the viewer should disappear—to reappear when the landmark again is visible.

7.3.3 Semantics of position

Arriving in an unknown city can be confusing. Still there are some implicit rules about city layout that we can utilise in locating things. Often public services like railway stations, tourist informations, town halls, etc. are located in the centre; habitant quarters are in the outskirts; and airports usually quite far from the centre.

Having a spatial metaphor allows us to set analogue guidelines for the layout of software architecture. One can imagine that domain knowledge central for the company is always centrally located; user interface relevant parts “north” of the centre; and operating system specifics in the “south”, etc.

This way team members will know where to look even in an completely unfamiliar project.

7.3.4 Visual tools

The combination of a spatial metaphor and maps can become the backbone for numerous tools and views:

- *Collaborative awareness*: Landmarks could display information about who is currently modifying local versions of a component; by names, colour coding, or some other scheme. This way awareness could be achieved in a much less disturbing way than for instance conventional notifications using e-mail.
- *Project overview*: Colour code landmarks according to release state of components (green = released, yellow = alpha-test, red = development, and so forth), deviations from estimates, critical path components, etc. etc.
- *Visual directory grep*: Mark approximate location of a match in the landmarks by for instance a bright red spot. Imagine a directory grep looking for some identifier giving a list of 2000 matches in 300 files in 90 directories: Such information is virtually useless. However 2000 red spots (most of them will probably merge) dispersed over a map of an application give immediate overview and structural information by indication of clustering and possible misuse in form of lonely points — and the ability to locate and zoom into areas of interest quickly.
- *Structural diff*: Colour code components according to no. of changes between two versions (green = no changes – to – red = many changes). Or mark with red spots on the landmarks approximate positions changed in the files.
- *Code metrics* or *test coverage* can be visualised to give structural information.

I have no doubt that many other useful ideas will come into mind as people start using this approach.

The appeal of these tools all relies on the visual formalism of maps combined with a spatial metaphor for components. This empowers us to get an overview of relations—in essence giving a *topography* of the application—otherwise difficult to infer from textual output.

One should also mention the idea of *fish-eye views* [Furnas86] where e.g. the mouse pointer defines the part of the application structure that is interesting (the focus) and therefore shown in great detail while remote regions are shown in successively less detail. It could be interesting to investigate this technique as well.

7.3.5 Spatial interpretation of hyper-links

Powerful editors and browsers of today often also contain cross-referencing abilities like e.g. the Mjølner BETA structure editor Sif [Sif94]. Sif can follow semantic links from the application of a name to the declaration. Though useful it has a backside, namely the already mentioned "getting-lost-in-hyperspace" phenomenon.

I envisage a close integration between the Ragnarok user interface and powerful editors/browsers: Clicking a file name in a landmark in a map brings up an editor on the file, and if hyper-links are followed in the editor, the map will smoothly move to display the landmark containing the file of the hyper-link endpoint. This way a spatial interpretation of hyperlinks is provided hopefully aiding in maintaining the sense of direction.

7.4 Direct manipulation

Shneiderman introduced the term *direct manipulation* in his often cited paper [Shneiderman83]. The basic idea is to provide visual artifacts that react on user manipulations through devices like keyboard, mouse, joystick, etc., in a sensible way.

The Self-4.0 programming environment [Smith et al. 95, Maloney95] for the Self prototypical object oriented language [Ungar et al. 95] is based on concreteness and has taken the idea of direct manipulation far. In this environment prototypical objects are moved and manipulated in a very direct way. Especially construction of user interfaces is direct as one simply drags and drops components on top of each other to form composite objects. More abstract manipulations on objects are performed by context-sensitive menus that appear when a mouse button is pressed above an object.

The Mjølner Orm system has an *object-oriented* approach to window systems resembling the Self approach [Hedin et al. 94], using context-sensitive menus.

I have adopted this idea and commands are issued directly to landmarks: Either using a context-sensitive pop-up menu when the mouse is above any item in Ragnarok (landmarks, maps, etc.) or simply by short-keys from the keyboard.

7.5 Discussion

The spatial metaphor can be used as basis for many views/aspects on the software project. However there are obvious exceptions. Devising a project schedule to determine the sequence of tasks, critical paths, etc. requires a view with strong focus on *time*. Clearly a layout based on spatial relations is not suitable.

One problem is that of utilising the *space* available: In order to create the "road map" we need free space around the landmarks to draw class diagram notation. This limits the space available for displaying information within the landmarks themselves. The key to the solution lies in providing summary information only in the landmarks themselves and have the ability to get more detailed information on request.

Another problem is that the success of a spatial metaphor relies on the "world" created being relatively stable. We have all tried the problems when our favourite supermarket decides to put everything in new places; it takes quite a while to regain proficiency in locating things. And we all know that restructuring and redesign occur in software projects. However it depends very much on *scale*. It doesn't matter much that a couple of classes are shuffled in a class category as long as the category is not moved. And my own experience is that the large scale structure of an application is more stable than at the class level — and especially in larger projects there is a substantial inertia because restructuring is costly.

The hierarchical window system of Mjølner Orm [Magnusson94, Hedin et al. 94] bears some resemblance to Ragnarok especially in the use of visual nesting. However whereas Orm insists on displaying the context leaving little room for deeply nested views, Ragnarok provides infinite zooming in.

An interesting article by Jones et al. questions the usability of a spatial metaphor [Jones et al. 86]. In three experiments with persons classifying newspaper articles using names or locations they get strong indications that as the number of items is increased names are more easily remembered than locations. However their results can not be transferred directly to Ragnarok:

- In the experiments the test persons were all the time trying to structure *new* data as opposed to *well-known* data.

- The test persons were not allowed to preview the material before classifying it.
- They used a *flat* location space.

In contrast the spatial metaphor as used in Ragnarok is meant to structure data we are well acquainted with in a hierarchical space. We can use our knowledge of the data to group related things together and as software engineering usually means locating the same things again and again we get to know our "world" well.

I see an analogy between screen-oriented editors and the spatial approach in Ragnarok: The benefit of screen editors compared to line-oriented editors is that the context of an item of interest is always visible and that items to a large extent are manipulated directly. Using line editors you are required to *remember* the context of a line and use special commands to manipulate individual characters or words. Ragnarok always shows the context and allows the developer to directly interact with the item of interest.

8 Implementation of the Prototypes

"Man må kravle før man kan gå"
Danish saying.

In order to test some of the fundamental ideas a rough prototype of Ragnarok was made. The prototype is a *vertical prototype* [Floyd84] with main emphasis on two basic properties:

- The basic *context-preserving software configuration management* model (section 4 and 5).
- The use of a *spatial metaphor* combined with *maps* for navigation in a project structure (section 7).

Other interesting properties that there was unfortunately not time to implement were annotations and annotation viewers, the collaborative extensions to the context-preserving SCM model[16], and "highly salient, visual landmarks".

As described in section 9 I got the idea during the initial design phase to actually create two prototypes: A stand-alone command-line tool containing the context-preserving SCM model only, and a graphical prototype with more emphasis on the user interface.

The prototypes are implemented in the BETA language [Madsen et al. 93], using the Mjølner BETA system [Andersen et al. 94], and uses some of the design patterns described in [Gamma et al. 94]. The implementation consists of about 13.000 lines of BETA code and about 385 "staff"-hours went into it. During the design phase I used paper based mock-ups of the envisaged Ragnarok to get an initial impression of the environment.

8.1 Layered design

Though it was a prototype a great deal of attention was paid to the design in the hope that it could be carried on into the next generations of Ragnarok.

The design is based on a *layered model* having three layers. Each layer knows only the layers below itself and extends the functionality. Below they are described from the simplest layer and up.

8.2 Component Model

Component Model is a class category which provides the context-preserving SCM model. Basically it is a front end to RCS [Tichy82]; it maintains information about components and their dependencies in flat text files, one for each component, and controls actual source files by calling the RCS "ci" and "co" tools and parses their output.

The outline of the Component Model class category is depicted in the lower part of figure 9.

The full context-preserving SCM model described in section 5 is implemented in Component Model but presently no support is made for local versions and local component storages as described in section 6. The project component storage is supported in the form of a hierarchical repository containing the RCS files and the Component Model controlled files. The directory hierarchy typically (but not necessarily) mimics the hierarchy of the components.

[16]These ideas also partly grew out of response from the user groups working with the prototype.

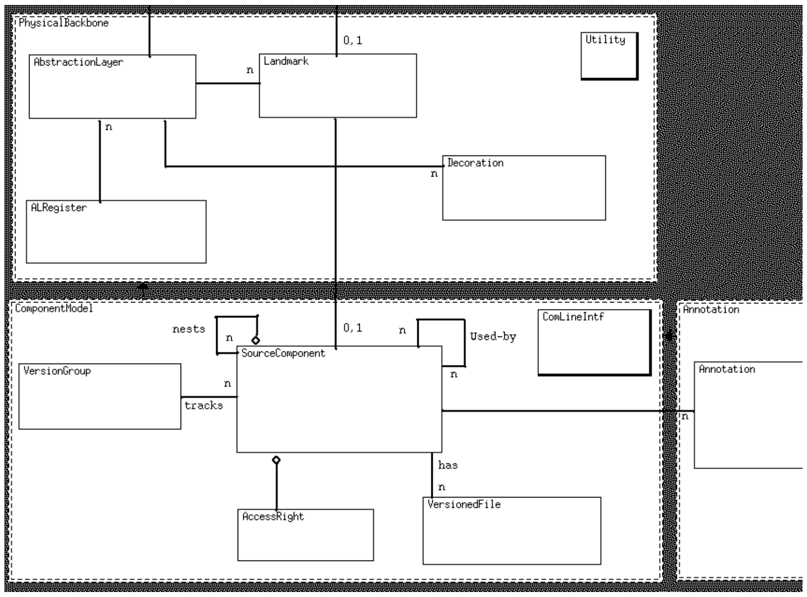

Figure 9: Component model, Annotation, and Physical Backbone. The class diagram graphics was implemented very fast, therefore they are a bit “shaky”.

Each developer defines a private workspace on a per-project basis in the form of a directory: A check-out operation recreates the directory structure found in the repository using this directory as root.

Check-outs default to read-only. To modify a component you obtain a *lock* on it adhering to the “lock-modify-unlock” mechanism of the underlying RCS tool. The lock is *only* on the component i.e. child components and components depended upon are *not* automatically locked.

SourceComponent is the central class representing the concept of a source component from section 4. It has AccessRight's and Annotation's associated (presently only stubs for both), and manages a list of VersionedFile's (the source files controlled by RCS), as well as the child- and depends lists (denoted “nests” and “used-by” in the diagram).

A VersionGroup class handles versioning of SourceComponent instances, and is responsible for maintaining data structures that allow recreation of all versions of the source components, and maintaining the flat text files for persistence.

Annotation is a class category on itself because it is a “placeholder” for future development where it is envisaged that different kinds of annotations will be provided by subclasses of class Annotation.

The ComLineIntf class utility contains the command-line tool source and manual described next.

8.2.1 RCM: The command-line tool

RCM is a command-line interface to the Component Model class category, unimaginatively named “RCM” which stands for “Ragnarok Component Model”.

Originally RCM was meant purely as a mean of bootstrapping the development process of Ragnarok, but as outlined in the next section, it is presently being used by other teams.

RCM provides operations for creating the component structure, generating different lists of the component and dependency structure as well as lists of associated files with RCS version numbers, and of course the basic context-preserving version control operations of check-in, check-out, and some facilities for merging individual development efforts.

8.3 Physical Backbone

The physical backbone is a class category that creates a mapping from the logical design space into a physical, concrete, space, by associating landmarks to components, and managing these landmarks in abstraction layers.

The physical backbone class category is depicted in the upper part of figure 9.

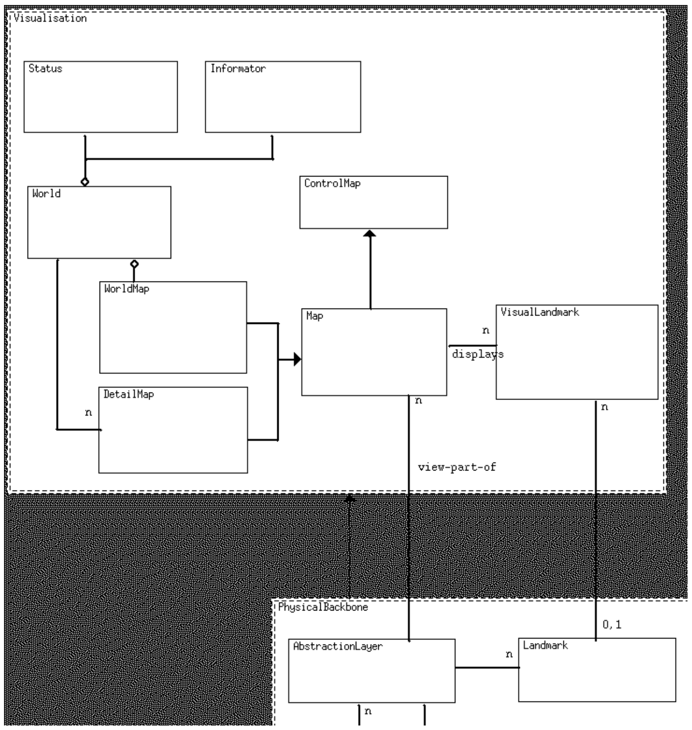

Figure 10: Visualisation, and relations to Physical Backbone.

Class Landmark is associated with SourceComponent on a one-to-one basis (the 0 cardinality is presently not used), and an abstraction layer may contain several landmarks. The abstraction layers as a whole are controlled from ALRegister, primarily for persistence between Ragnarok sessions.

The Decoration class defines instances of class diagram notation graphics like e.g. lines, inheritance arrowheads, aggregation symbol, text, etc.

The main task of the physical backbone is that of managing coordinate transformations: Integers are used for coordinates and in order not to run out of resolution in very deeply nested landmarks the scale between layers differs by a factor two (i.e. point (4,7) in layer 1 projected down on layer 2 is (8,14)). Typically a detail map requests an abstraction layer to "give a list of everything visible in a region (x_0, y_0, x_1, y_1) and d layers down". The abstraction layer calculates this list converting the coordinates into a coordinate system suitable for displaying on the map.

The reason that class Landmark not is a subclass of SourceComponent is that it may be beneficial to be able to create landmarks but defer the creation of the actual source component, typically in a design situation.

8.4 Visualisation

The visualisation part of Ragnarok is build on top of *Lidskjalv* [Lidskjalv95], a platform independent object-oriented user interface construction framework for the Mjølner BETA System.

Most classes should be identifiable from the discussion in section 7. The ControlMap is an abstraction of a Map that is mostly introduced as a way to break dependency between Map and VisualLandmark: Map knows VisualLandmark but VisualLandmark only knows ControlMap.

The *observer pattern* of Gamma et al. [Gamma et al. 94] is used to keep maps synchronised when several are displaying the same underlying landmark. The classes are however not shown because they reside in a project class utility (Soup) which is a general purpose library; and adhering to the spatial metaphor a landmark can only be in one place. This is clearly unfortunate seen from a documentation point of view, and I am presently considering to allow "referencing landmarks" to overcome this limitation.

9 First Light - Case Stories

In astronomy the event of the first proper image coming from a new telescope after assembling and up-lining is called “first light”.

The layered design of the prototype gave me an opportunity to have “first light” quite early and test the underlying ideas in the context-preserving SCM model. When the first layer, Component Model, was implemented, it was equipped with a command-line interface and used as a stand-alone SCM tool to control the development process of the rest of the Ragnarok prototype as well as provide release control for the command-line tool itself. I have always found it a benchmark test of any development tool that it can indeed aid in developing itself. The “bootstrapping” of the first Algol compiler is an admirable example.

However the “baby” soon outgrew its cradle. Within a week of the first version of RCM a development team at Institute for Storage ring facilities in Aarhus (ISA) was using it which put much more emphasis on the collaborative issues, as two of the developers would use the tool actively.

Later another team of two developers at DEVISE started using the tool in a project developing a prototype hospital system.

Both groups are involved with real development projects. The descriptions below are based on one hour open-ended interviews using a mixture of the standardised open-ended interview and the general interview guide approach [Patton80].

9.1 ISA team

The ISA team project, named LabSys, is a system for controlling accelerators, storage rings, and other large distributed equipment in experimental physics.

The team consists of five people, three in Aarhus and two in Stockholm. Presently only the three developers in Aarhus are actively using the RCM tool, but the Stockholm group will start using it some time this summer. The delegation of responsibility is rather strict where each member has “the final word” for a well defined part of the system.

The development language is C++ combined with SQL, and the tools are MicroSoft Visual C++, MicroSoft SQL Server, and MicroSoft Access via ODBC, on the MicroSoft NT platform. Presently about 970 kB of data in 175 files in a rather shallow hierarchy of about 25 components is under RCM control. The basic design consists of four main layers, each represented by a (large) component giving few inter-component dependencies. The component structure mimics the design architecture.

The full LabSys project is estimated to about five man-years with about two man-years worth of work made so far.

Before RCM was introduced they had version control using RCS [Tichy82] combined with scripts. Now RCM has been adopted as their SCM solution.

As the main benefits of using RCM they emphasise:

- *Version control of configurations:* Using scripts to control RCS were cumbersome because adding a new file to a module required changing the script making it unsuitable to check out previous releases of the module.
- *Overview:* The RCM recursive list commands make it possible to get a good overview of files and components in the project.
- *Context-preserving aspect:* Though not in the maintenance phase yet they see the benefits of traceability and consistency of releases in the underlying model.

When questioned about the drawbacks they mentioned:

- *Version numbering:* RCM uses the simplest possible version numbering scheme (simply consecutive numbers) for components which means that e.g. the branch structure can not be seen from the version identification. Thus getting an overview of versions in a version group is difficult
- *Version pollution:* They work with fewer intermediate versions than when they used RCS. They list themselves two main reasons: Because of the version numbering problem they fear loosing the overview of versions if too many “non-milestone versions”are introduced, especially in the root component; secondly they are presently in a phase of pretty straight forward implementation with little experimentation meaning less need for intermediate versions.

- *Context overview:* The backside of the context-preserving model is that checking-out a single component may trigger a lot of actions on other components; and it is sometimes difficult to get an overview of the consequences of for instance a check-out.

The present version numbering scheme is obviously too limited and has of course never been meant as final but merely a prototype solution. The introduction of local storages would help on the pollution problem (and a crucial extension when the Stockholm team joins in) while they suggested a "consequence function" for the last item, where one could see the consequence of e.g. a check-out before actually doing it. A similar technique is available in CVS [Berliner90].

Concerning the underlying concepts in RCM they were very positive. The component idea to group a set of source files suited them fine because changes are often made to many files but usually within a single component (their components are generally large with some 10-20 files, partly because of the "wizards"/code generation abilities of Visual C++).

The hierarchical approach fitted well with both the structure of their source code as well as their delegation of responsibilities.

On the question of using a "lock-modify-unlock" versus "copy-modify-merge" mechanism for controlling version creation they argued strongly in favour of locks. This is partially because the Visual C++ environment contains strong tools for user interface construction but relies on the ability of generating unique ID's for interface elements in resource files. Thus merging resource files created in parallel is impossible and they cannot allow this to happen.

This suggests that Ragnarok should support explicit locks as well as a "copy-modify-merge" scheme, if a team decides so; perhaps on a per component basis.

9.2 DEVISE team

The DEVISE team is working on a prototype of a system handling resource allocations in a hospital. The main purpose of the project is to test programming environment tools within the DEVISE BETA system.

The team consists of two developers. Before RCM was introduced there was no explicit delegation of responsibilities.

The system is developed in BETA using the development tools in the Mjølner BETA programming environment: The structure editor Sif, CASE tool Freja, the interface builder Frigg, persistent store browser, etc., on the UNIX platform. About 270 kB of source code in 50 files in 28 components is presently controlled. Being an evolving prototype the component hierarchy is a bit ad hoc with many inter-component dependencies. The resulting component structure is thus more akin to a directory structure than mimicking the logical application architecture.

None of the team members have any practical experience with other software configuration management tools.

One of the main reasons that the team wanted to use RCM was due to problems concerning collaboration; before the introduction of RCM they had individually modified copies of many files which was tedious and error-prone to merge.

As main benefits they list:

- *Easier merge of individual effort:* RCM forced them into a more explicit division of responsibilities, and serialised access meaning much less effort is spent on merging work made in parallel.
- *Context-preserving aspect:* On one occasion they had to provide a working prototype whilst in the middle of changing the underlying persistent storage layer. An earlier known-to-work version was checked out in a single operation.
- *Security against overwriting work:* RCM protects against loosing effort during a cumbersome manual merging process.

Of problems they mention:

- *Version pollution:* Often intermediate, even non-working, versions have to be checked-in because their working hours often do not overlap and in order to enable the other developer to change things, components must in effect always be unlocked after a work session.
- *Global locks too coarse:* Often they resort to change the write permission of a single file instead of obtaining a lock on a component. The global locks of RCM are too coarse and heavy-weight if only a single debug print statement is wanted in some code which is actually the responsibility of the other developer, and because the change is only temporary there is no need nor wish for a check-in.

These drawbacks could also be handled by the introduction of a local storage layer between the developer and the project component storage.

The DEVISE team had no objections concerning the underlying principles. They even felt that RCM had influenced the structure of their application in a positive way. When asked about the idea of context-preserving configuration management one developer countered the question with "What is the alternative?"—a statement I find particularly amusing as few other SCM tools provides strong support in this area. They too reported that it was difficult to overview the consequence of a check-out but did not feel it as a serious problem.

9.3 RCM on Ragnarok

I used RCM intensively myself in the development of the physical backbone and visualisation layers as well as to control the releases of RCM itself to the two user groups[17]. The Ragnarok prototype consists of about 600 kB of BETA code, LaTex documents, and text files in 100 files in 30 components.

I generally agree with the viewpoints of the user groups: It is ideal for release control because of the context-preserving property, and the often felt problem of getting the configurations right is almost gone. Still getting a local component storage is very important.

One annoying problem I have encountered myself is that the depend-lists of components has to be maintained manually[18]. It is all to easy to forget updating the component dependency structure if an additional 'include' is added to the source code. This means that the context of an old version cannot be checked out properly.

9.4 Ragnarok

The graphical user interface prototype of Ragnarok has so far not been used intensively and not by anyone but myself. The main reason is that the direct manipulation interface to the components is only implemented as short-cut keys, not by pop-up menus; and no manual exists. Secondly only few of the full set of version control facilities are available. Thirdly there are still some inconveniences and bugs around.

It is however my impression that the spatial metaphor provides good and fast navigational support; related components/files are spatially close, so when wanting to open a specific file you can actually create a detail map where the component with the wanted file is present in one or two simple mouse operations.

The ability to change the contents of a detail map by dragging around the outline in the world map is also a strong facility.

Another important difference between using RCM and the graphical Ragnarok is that when using RCM you quite easily revert into thinking about components as "directories"[19], and not as logical design blocks. However when you in Ragnarok create a new landmark you think much more in terms of classes and class categories because the visual representation focusses your mind in the direction of creating object-oriented design diagrams.

10 Future Work

Lots, lots, lots...

Below I will give a short list of activities:

- Get the user interface prototype of Ragnarok into a stage where the user groups can begin using it instead of the command-line tool. As they are already used to the context-preserving SCM model they need only "climb the learning curve" for the spatial metaphor and use of maps. Such tests would provide valuable information concerning the validity of the underlying assumptions.
- Implementing and testing the collaboration SCM model in a real setting. The coordination of development effort between the Aarhus and Stockholm teams in the ISA LabSys project provides an ideal test situation. Also handling of the BETA environment in DEVISE could be used as "laboratory".

[17] Presently there has been 13 releases

[18] Bendix refers to this as the "shadow" problem [Bendix95].

[19] If a team is only interested in the SCM aspects and only use RCM then this is of course fine.

- Implementing annotations. Traceable, strong, support for handling bug reports and checklists for quality assurance could be first step, and then augment the model with task- and project management aspects in the next.
- Delete operations for versions as well as components have not been treated. They are of course a bit complicated due to the context-preserving SCM model. The semantics of delete operations must be defined, implemented and tried out.
- Handling of derived components to facilitate software manufacture.
- Trying out the ideas on large-scale projects. The idea of unifying the work-break-down structure with the design structure using project management annotations to components is a daring one and needs to be tried out in a real setting.
- It is infeasible that I can envisage all uses of Ragnarok; an endless string of ideas for new aspects, visual tools, and annotations can be foreseen. Thus providing *extensibility* is important. Inspiration can be found in the work on Emacs [Stallman84] and of Malhotra [Malhotra94].
- The landmarks in Ragnarok must cope with a potentially very large set of operations depending on aspects, annotations, etc., which may end in serious problems concerning overview and consistency. It should be studied if e.g. a *tools and materials* metaphor [Bürkle et al. 95] could be used to structure the user interaction in a comprehensible and intuitive way.
- Integration with tools in the language-near end of the activity spectrum, compilers, debuggers, editors, etc., is vital for acceptance by developers.
- Presently Ragnarok relies on two-dimensional planes as physical space and simple rectangles as landmarks, but a full three-dimensional model would be very interesting to investigate.

An interesting line of thought is looking at an *event model* that suits the human mind better. Computer systems generally provide a simple linear event model in which user interaction events are processed linearly (where events may be "unprocessed" like emacs "undo"). Humans however seldom work linearly. Watching an artist creating a portrait you' ll notice how attention in a split second is moved from a large line in the face to a tiny shadow somewhere else, and then back again. In system development and implementation we often experience the same: In the middle of writing a code statement you get a brilliant idea for something in a completely different part of the system. I term these events "quantum leaps of the mind". *Mind maps* is a technique that allows you to write down ideas during a brain-storm where you have many such quantum leaps without loosing the overview. I would like to research more on the cognitive foundation for this aspect, and see if it is possible to provide some support for structuring and maintaining overview of different lines of thoughts while allowing such quantum leaps.

One could also envisage language support. The ability to draw class diagrams suggests code generation abilities, dependencies could be inferred from the source code itself, etc. However the DEVISE project already has state-of-the-art tools for these aspect, [Freja95] and [Sif94], and my interests are more towards the overall aspects of management, project support, and user interface issues. However a symbiosis would clearly provide a strong environment.

11 Conclusion

The main contributions of my work so far is the *context-preserving approach to software configuration management* and the use of a *spatial metaphor and maps* in visualising the structure of a software project.

The continued use of the RCM tool and the good response it has received from both the ISA and the DEVISE teams leads me to believe that the ideas are sound in the context of "small to medium sized" projects. Hopefully they scale to larger projects especially when combined with the ideas outlined in section 6. Both teams readily accept the underlying concepts of a component, hierarchical structure, version control of dependencies as well as preserving context as a natural and intuitive fundament. However it is very important to ground the ideas in further experiments in larger projects with more people.

Even with the limited use of the full user interface prototype I also think there is great potential in the ideas, especially when they are combined with annotations. In my opinion they have the potential of providing overview of the many aspects of large-scale systems in a natural and intuitive way.

So—with high hopes and some successes in the bag, I look forward to the next two years of work.

12 Acknowledgements

"To live in fear is a life half-lived"
From the film "Strictly ballroom"

There are two people who I need to thank more than anyone else.

The first one is my wife who (reluctantly) agreed to cope with a substantial cut in our income in order for her husband to pursue some of his crazy ideas.

The second one is Erik Meineche Schmidt for giving me the chance of getting back into academia, and providing guidance during the first period.

I am of course also in debt to Ole Lehrmann Madsen for being an enthusiastic supervisor and the DEVISE group in general for providing feedback on my ideas and also guiding me through some of the dark corners of BETA. Also thanks to Torben Worm, Karsten Telling Nielsen, Henrik Røn, and Stephan Erbs Korsholm for enduring the early versions of RCM and providing valuable comments and many good new ideas. Klaus Marius Hansen suggested the name "Ragnarok" for the environment.

References

[Andersen et al. 94] Peter Andersen, Lars Bak, Søren Brandt, Jørgen L. Knudsen, Ole L. Madsen, Kim J. Møller, Claus Nørgaard, Elmer Sandvad, *The Mjølner BETA System*, in [Knudsen et al. 93]

[Bendix95] Lars Bendix, *Configuration Management and Version Control Revisited*, PhD dissertation, Institute of Electronic Systems, Aalborg University, Denmark, Dec. 1995

[Berliner90] Brian Berliner, *CVS II: Parallelizing Software Development*, in Proceedings of USENIX Winter 1990, Washington D.C.

[Booch91] Grady Booch, *Object Oriented Design*, The Benjamin/Cummings Publishing Company, Inc., 1991

[Booch et al. 95] Grady Booch, James Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set Version 0.8, Rational Software Corporation, Santa Clara/CA, 1995

[Bürkle et al. 95] Ute Bürkle, Guido Gryczan, Heinz Züllighoven, *Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions*, Human-Computer Interaction, Vol. 10, pp. 293-336, 1995

[Cagan95] Martin Cagan, *Untangling Configuration Management*, in [Estublier95]

[Christensen95] Henrik B. Christensen, *A Retrospective Case Study: SAVOS*, Internal work note, Obtainable from the author.

[Dowson87] Mark Dowson, *Integrated Project Support with IStar*, IEEE Software Nov 1987, p. 6-15

[Estublier95] J. Estublier (Ed.), *Software Configuration Management, ICSE SCM-4 and SCM5 Workshops, Selected Papers*, Lecture Notes in Computer Science 1005, Springer Verlag 1995

[Floyd84] C. Floyd, *A Systematic Look of Prototyping*, in R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Zullighoven, (Eds.), *Approaches to Prototyping*, Berlin: Springer-Verlag, 1984

[Freja95] *Freja, An Object-Oriented CASE Tool*, MIA 93-24(1.0), Mjølner Informatics Report, 1995

[Furnas86] George W. Furnas, *Generalized Fisheye Views*, in [Mantei et al. 86]

[Gamma et al. 94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reuseable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1994

[Goldberg et al. 95] Adele Goldberg, Kenneth S. Rubin, *Succeeding with Objects, Decision Frameworks for Project Management*, Addison-Wesley 1995

[Gustavsson90] Anders Gustavsson, *Software Configuration Management in an Integrated Environment*, Dept. of Comp. Science, Lund University, Thesis, 1990

[Harel88] David Harel, *On Visual Formalisms*, Communications of the ACM, Vol. 31, May 1988

[Hedin et al. 94] Görel Hedin, Boris Magnusson, *Direct interaction in the Orm programming environment*, in [Knudsen et al. 93]

[Herbsleb et al. 95] James D. Herbsleb, Helen Klein, Gary M. Olson, Hans Brunner, Judith S. Olson, Joe Harding, *Object-Oriented Analysis and Design in Software Project Teams*, Human-Computer Interaction, Vol. 10, pp. 249-292, 1995

[Jacobson et al. 92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard, *Object-Oriented Software Engineering - a Use Case Driven Approach*, Addison-Wesley 1992

[Joachim96] Joachim Sauter, *Terra-Vision*, privat comm., project description on WWW: http://www.artcom.de:80/Welcome

[Jones et al. 86] William P. Jones, Susan T. Dumais, *The Spatial Metaphor for User Interfaces: Experimental Tests of References by Location versus Name*, ACM Transactions on Office Information Systems, Vol 4, No. 1, Jan 1986

[Knudsen et al. 93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson, *Object-Oriented environments - The Mjølner Approach*, Prentice-Hall 1993

[Leblang et al. 87] David B. Leblang, Robert P. Chase Jr., *Parallel Software Configuration Management in a Network Environment*, IEEE Software, Nov. 1987, p. 28-35

[Lidskjalv95] *Lidskjalv: User Interface Framework*, MIA 94-27(1.1), Mjølner Informatics Report, 1995

[Nardi et al. 93] Bonnie A. Nardi, Craig L. Zarmer, *Beyond Models and Metaphors: Visual Formalisms in User Interface Design*, Journal of Visual Languages and Computing, Vol. 4, No. 1, March 1993

[Nørmark89] Kurt Nørmark, *Programming Environments – Concepts, Architectures and Tools*, R-89-5, IES, Aalborg University, April 1989

[Madsen et al. 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, *Object-Oriented Programming in the Beta Programming Language*, Addison Wesley, 1993

[Madsen94] Ole Lehrmann Madsen, *The Mjølner BETA fragment system*, in [Knudsen et al. 93].

[Magnusson94] Boris Magnusson, *The Mjølner Orm System*, in [Knudsen et al. 93]

[Malhotra94] Jawahar Malhotra, *Tailorable Systems: Design, Support, Techniques, and Applications*, Ph.D. Thesis, DAIMI PB-466, Aarhus University, 1994

[Mantei et al. 86] Marilyn Mantei, Peter Orbeton, *CHI'86 - Human Factors in Computing Systems: Proceedings from a Conference*, ACM, New York, 1986

[Maloney95] John Maloney, *Morphic: The Self User Interface Framework*, Sun Microsystem Inc, 1995

[Mathiassen et al. 93] Lars Mathiassen, A. Munk-Madsen, P. A. Nielsen, J. Stage, *Objektorienteret Analyse*, Forlaget Marko, 1993

[McKnight et al. 91] Cliff McKnight, Andrew Dillon, John Richardson, *Hypertext in Context*, Campbridge University Press, 1991

[Meier91] Sid Meier, *Civilization User Manual*, MicroProse, 1991

[Meyer88] Bertrand Meyer, *Object-oriented Sofware Construction*, Prentice Hall International Series in Computer Science, 1988

[Miller95] Peter Miller, *Aegis, A Project Change Supervisor, User Guide*, version 2.3, 1995, part of the Aegis distrubution, obtainable through various ftp-sites.

[Patton80] Michael Quinn Patton, *Qualitative Evaluation Methods*, Sage Publications, Beverly Hills, Calif., 1980

[Mikkelsen et al. 89] Hans Mikkelsen, Jens O. Riis, *Grundbog i Projektledelse*, 3. udgave, 2. oplag, Forlaget PROMET Aps, 1989

[Minör et al. 93] Sten Minör, Boris Magnusson, *A mode for Semi-(a)Synchronous Collaborative Editing*, Proceedings of Third European Conference on Computer Supported Cooperative Work - ECSCW'93, Kluwer Academic Publishers, 1993

[Prasuan et al. 93] Prasuan Dewan, John Riedl, *Towards Computer-Supported Concurrent Software Engineering*, IEEE Computer, Jan. 1993

[Olsson94] Torsten Olsson, *Group Awareness using Fine-Grained Revision Control*, Proceedings of NWPER, 1994

[Reisberg87] D. Reisberg, *External representations and the advantages of externalizing one's thought*, in *Proceedings of the Cognitive Science Society*, Seattle, Washington, pp. 281-293, 1987

[Rumbaugh et al. 91] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall International Editions, 1991

[SCM] J. F. H. Winkler (ed.), *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, West Germany, B. G. Teubner, Stuttgart, Jan 1988

[Shneiderman83] Ben Shneiderman, *Direct Manipulation: A Step Beyond Programming Languages*, IEEE Computer, August 1993

[Sif94] *Sif / A Hyper Structure Editor / Tutorial and Reference Manual*, MIA 90-11(1.2), Technical Report, Mjølner Informatics, 1994

[Smith et al. 95] Randall B. Smith, John Maloney, David Ungar, *The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility*, Conference Proceedings, OOPSLA'95, ACM SIGPLAN Notices, Vol. 30, No. 10

[Sommerville89] Ian Sommerville, *Software Engineering*, (3rd edition), Addison Wesley, 1989

[Stallman84] Richard M. Stallman, *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, in “Interactive Programming Environments” by D. R. Barstow, H. E. Shrobe & E. Sandewall, 1984

[Tichy82] Walter F. Tichy, *Design, Implementation, and Evaluation of a Revision Control System*, 6th Conference on Software Engineering, Tokyo, Japan, 1982

[Tichy88] Walter F. Tichy, *Tools for Software Configuration Management*, in [SCM]

[Ungar et al. 95] David Ungar, Randy B. Smith, *SELF: The Power of Simplicity*, Lisp and Symbolic Computation, 4, 3, 1991

[Vessey et al. 95] Iris Vessey, Ajay P. Sravanapudi, *CASE Tools as Collaborative Support Technologies*, Communications of the ACM, January 1995/Vol. 38, No. 1, p. 83

[Winkler et al. 88] Jürgen F. H. Winkler, Clemens Stoffel, *Program-Variants-in-the-Small* in [SCM]

A Colour images

The next two pages are colour image screen shots of Ragnarok; LaTeXcould not be persuaded to print them properly in landscape mode, so the captions are given here instead:

1. Colour version of figure 8.
2. A somewhat more elaborate Ragnarok screen shot where the user has checked-out component Landmark for modification ("obtained a lock"). The colour coding of map 4 showing version control aspect indicates components directly-modified (red), indirectly modified (yellow) and not modified (green) after Landmark has been locked.