Support for Collaborative Component-Based Software Engineering

Cornelia Boldyreff
University of Lincoln
cboldyreff@lincoln.ac.uk
Phone +44 (0) 1522 83 7115
Fax +44 (0) 1522 88 6974

David Nutter
University of Lincoln
dnutter@lincoln.ac.uk
Phone +44 (0) 1522 88 6653
Fax +44 (0) 1522 88 6974

Stephen Rank
University of Lincoln
srank@lincoln.ac.uk
Phone +44 (0) 1522 83 7306
Fax +44 (0) 1522 88 6974

Phyo Kyaw
University of Durham
phyo.kyaw@durham.ac.uk
Phone +44 (0) 191 33 41741
Fax +44 (0) 191 33 41701

Janet Lavery
University of Durham
janet.lavery@durham.ac.uk
Phone +44 (0) 191 33 41740
Fax +44 (0) 191 33 41701

iABSTRACT

Collaborative system composition during design has been poorly supported by traditional CASE tools (which have usually concentrated on supporting individual projects) and almost exclusively focused on static composition. Little support for maintaining large distributed collections of heterogeneous software components across a number of projects has been developed. The CoDEEDS project addresses the collaborative determination, elaboration, and evolution of design spaces that describe both static and dynamic compositions of software components from sources such as component libraries, software service directories, and reuse repositories. The GENESIS project has focussed, in the development of OSCAR, on the creation and maintenance of large software artefact repositories. The most recent extensions are explicitly addressing the provision of cross-project global views of large software collections and historical views of individual artefacts within a collection. The long-term benefits of such support can only be realised if OSCAR and CoDEEDS are widely adopted and steps to facilitate this are described.

Keywords: Distributed Systems, Internet Technologies, Web Applications, Metadata, CASE tools, Groupware, XML, Software Management, Data Models, File Management Systems.

INTRODUCTION

The systemic representation and organisation of software descriptions (e.g. specifications, designs, interfaces, and implementations) of large distributed applications using heterogeneous software components have been addressed by research in the Practitioner and AMES projects (Boldyreff, Elzer, Hall, Kaaber, Keilmann, and Witt, 1990; Boldyreff, 1992; Boldyreff, Burd, Hather, Mortimer, Munro, and Younger, 1995; Boldyreff, Burd, Hather, Munro, and Younger, 1996). The Practitioner project explicitly addressed the reuse of software concepts, and developed a standard form to handle representations of software concepts from their specification to their associated implementations as components. The AMES project, while focused on maintenance support, organised the associated software

components at various levels of abstract representations using hypertext and the web. In both projects, it was assumed that the underlying collections of software components would support software reuse and the subsequent evolutions of systems composed from components. However, without appropriate representations and organisations, large collections of existing software are not amenable to the activities of software reuse and software maintenance; these activities are likely to be severely hindered by the difficulties of understanding the software applications and their associated components. In both of these projects, static analysis of source code and other development artefacts, where available, and subsequent application of reverse engineering techniques were successfully used to develop a more comprehensive understanding of the software applications under study (Zhang & Boldyreff, 1990; Fyson & Boldyreff, 1998). Later research addressed the maintenance of a web-based component library in the context of component-based software product line development and maintenance (Kwon, Boldyreff, and Munro, 1997). The classic horizontal and vertical software decompositions proposed by Goguen (1986) have influenced all of this research. While they are adequate for static composition, they fail to address the dynamic aspects of composing large distributed software applications from components especially where these include software services that may be dynamically bound at run-time.

Recent research within the CoDEEDS project has made some progress towards the determination of design spaces to support both the static and dynamic system composition as well as the determination of the physical deployment and long-term operation of large distributed system composed

from heterogeneous components (Boldyreff, Kyaw, Nutter, and Rank, 2003). The current prototype implementation of collaborative support for the determination, elaboration, and evolution of design spaces, based on the CoDEEDS framework (Boldyreff & Kyaw, 2003), employs at its base another development of our recent research within the GENESIS project, the Open Source Component Artefact Repository, OSCAR (Boldyreff, Nutter, and Rank, 2002a; Boldyreff, Nutter, and Rank, 2002b; Boldyreff, Nutter, and Rank, 2002c; Nutter, Boldyreff, and Rank, 2003)

The GENESIS project developed a generalised environment for process management in collaborative software engineering (Gaeta & Ritrovato, 2002). A key component of this environment is an underlying distributed repository, OSCAR, to hold the software artefacts (both the artefact data and its associated metadata). A software artefact is any component of a work product resulting from the software engineering process. Thus the support provided covers not only the engineering of software systems from reusable software components, but also more generic reuse based on any work product components, such as project plans, requirements specifications, designs, test cases, and so on.

The research areas addressed in this chapter are:

- Process-aware support for collaborative software engineering;
- Management of software (and other) artefacts within and across software engineering projects; and
- Use XML-based artefact representations and interchange formats.

The remainder of this chapter is organised as follows: Firstly, the background related to web-based collaborative software development and software evolution are examined, then the overall design of OSCAR and the support for co-operative software development that it currently offers combined with CoDEEDS are described, along with extensions to OSCAR to provide historical awareness of artefact development across projects (Nutter & Boldyreff, 2003) and a global view of a number of distributed artefact repositories are elaborated. Finally, planned deployment and future research activities are discussed.

WEB-BASED COLLABORATIVE SOFTWARE DEVELOPMENT

The web and its associated technologies facilitate communication and cooperation amongst software developers enabling large collaborative software development projects to be undertaken. The open source community provides many examples of such projects. Multinational software projects are also commonplace within industry today. Various solutions are available to address the immediate support of these collaborative development projects throughout the lifecycle of the project. These solutions, open source and commercial, vary considerably in the elements of collaborative development and project management they address. SourceForge, in the open source domain, provides basic support for managing cooperative development of software artefacts such as handling mailing lists, forums, repository services, and bug tracking. However, it does not support workflow, resource management, or collaborative work by

many users on a single artefact (apart from the use of a CVS (Concurrent Versions System) repository to handle configuration management). Microsoft Project Professional supports enterprise project management over single or distributed sites in the commercial domain. It concentrates on the workflow and planning elements of cooperative development but has no specific focus on software engineering projects, unlike Rational's range of products, which support industrial software development across a global enterprise in the commercial domain. There are also general, not software development specific, web-based solutions that have been used to support cooperative working of distributed software development teams such as SiteScape, which handles a central repository, with forum-like facilities for interaction and BSCW (Basic Support for Cooperative Work) which formed the basis of the SEGWorld development (Drummond & Boldyreff, 1999). The GENESIS project has employed SiteScape to manage the deliverables associated with its various work packages and to coordinate document reviewing associated with the project's research and software developments. All of these current solutions support web-based access to project related data and artefacts under production by the software team.

In contrast, the OPHELIA (Dewar, Mackinnon, Pooley, Smith, Smith, and Wilcox, 2002; Wilcox, Smith, Smith, Pooley, MacKinnon, and Dewar, 2002) project offers support for collaborative work using software tools and employs a CORBA-based tool integration approach to do this. Various tools including project planning (MS Project) and development tools (such as ArgoUML) have been integrated using the ORPHEUS implementation of the OPHELIA framework. These applications can interchange data with

other modules of the ORPHEUS system, which perform tasks such as metrics calculation, recording traceability information, and archiving data. Theoretically, any tool may be integrated within ORPHEUS but providing truly universal data interchange is of course difficult and the effort required to integrate a tool significant.

The use of standard representations such as UML and XML-based notations has a beneficial effect on the cost and efficiency of software engineering projects. The GENESIS and CoDEEDS projects represent artefacts as XML documents. This has allowed the rapid development of sophisticated tools to handle artefacts; using currently available tools for XML handling has avoided the requirement to build entirely new artefact handling software. The use of UML as a common communication language amongst software engineers is supported by both projects. UML improves communication at the human level, while use of an XML exchange format can facilitiate the exchange of software artefacts.

Current solutions lack any means of obtaining a global view of project data and software artefacts across a number of projects irrespective of the initial methods and tools employed during the project's lifetime. Here the underlying artefact management system, OSCAR, being developed within the GENESIS project coupled with the CoDEEDS framework offers the basis for delivering such support in the future. One benefit of this is that a collection of artefacts can be treated as a repository of reusable components. The navigation and search facilities provided by GENISOM support the discovery of reuse candidates.

The GENESIS platform offers a process-aware environment which supports distributed software engineering, allowing flexible modelling and control of a collaborative software engineering project. While the GENESIS platform is based around process modelling and control, CoDEEDS specifically supports software engineering, providing support for specific software-related tasks such as architectural design. Both projects address supporting the evolution of software artefacts during their development and subsequent deployments within a variety of systems.

SOFTWARE EVOLUTION

A. Boldyreff was one of the first to recognise the role of evolution within the process of engineering computer systems (Boldyreff, A. W., 1954). He distinguished between mathematical models of systems and their corresponding physical realisation; and noted the necessity to evolve these models in step. In the 1970s, Lientz and Swanson (1980) studied a large number of software projects in many data-processing organisations. The study showed that software maintenance consumed approximately half the time of software professionals in the organisations which responded to their questionnaire. Generally, larger organisations spent a larger proportion of their time on maintenance, though results varied across industries. Their study showed that in organisations where maintenance was considered as a separate activity, it consumes a smaller proportion of effort. Lientz and

Swanson's study was carried out in the late 1970s, and the level of technology that was used by the organisations reflected this. For example, change logs were handled manually, and implementation languages such as COBOL and FORTRAN were common. Lientz and Swanson concluded, unsurprsingly, that larger and older systems have greater maintenance problems than smaller and newer systems, and that personnel issues such as the skill level and turnover of staff are of importance in determining the quality and effort of system maintenance.

Lehman and Belady (1985a) made a detailed study of the development of a single software system. In contrast to the method used by Lientz and Swanson, Lehman and Belady studied the software product (IBM’s OS/360) rather than the organisation. They examined the system's size at each release point, and showed that the size (in terms of lines of code and number of modules) and complexity of a system grows with each successive release, unless specific effort is made to reduce these factors. During this work, Lehman and Belady developed the idea of software system types, using the terms S-type, P-type, and E-type to describe the three types (Lehman & Belady, 1985b).

S-type programs are the simplest kind, being those programs which are formally defined as a function between input and output, with no reliance on or interaction with their environment, such as simple UNIX software tools; grep and awk, for example. P-type programs are those which solve real-world problems, and must use heuristics to arrive at approximate solutions. Examples include weather forecasting and chess playing, where the input to

the software is well-defined and well-formed, but in order to arrive at a useful solution in a reasonable amount of time, approximations must be used. E-type software is the most complex and most interesting kind of software.

An E-type program is situated in and interacts with its environment, leading to feedback between the software and the 'real world'. Total correctness of an E-type system cannot be shown in the abstract. Such software interacts with its environment and thus it can be only be shown to be effective in a particular, given, situation.

The results of these studies motivated Lehman to develop his laws of software evolution (Lehman, 1979; Lehman, Ramil, Wernick, Perry, and Turski, 1997; Lehman, 1996).

These laws describe the behaviour of software systems over time (Lehman, 1996). They are:

- **Continuing Change:** An E-type program must either adapt or become obsolescent.
- **Increasing Complexity:** Unless an evolving program has work done specifically to reduce its complexity, it will become more complex as a result of the evolution.
- **Self-Regulation:** The evolution process is self-regulating, with statistically determinable trends and invariants.
- **Invariant Work-Rate:** The average effective global activity rate is constant over the life-time of the system.

- **Conservation of Familiarity:** The content of successive releases is statistically invariant.
- **Continuing Growth:** Functional content of a system must increase with each release in order to satisfy user demands.
- **Declining Quality:** Unless an E-type program is rigorously maintained and updated to its changing environment, it will be perceived as declining in quality.
- **Feedback System:** The evolution process for E-type programs is multi-loop and multi-level. Successful management of the process depends on recognising and accounting for this fact.

Two of the key problems of maintenance are understanding the software in order to determine where to make changes, and validating the changed version of a system - determining that the correct changes and no others have been made (Baxter & Pidgeon, 1997). One important cause of the difficulty of maintenance is the complexity of software systems (Jackson, 1998); understanding a system in its entirety is often necessary before even a simple change can be made and validated, thus the need for support environments to capture and preserve the developer's understanding of programs.

As described above, there have been several studies of the evolution of software systems. These and other studies have led to models of the process and products of software evolution which have been used to manage and control software evolution. Process models identify the mechanism by

which the evolution is carried out, and product models identify the characteristics of the software which are important with respect to evolution.

There are two complementary research approaches to software evolution. The first approach, related to reverse engineering, aims to devise methods of working with legacy systems, while the second approach, related to forward engineering, attempts to design software that is easy to change. Whether a software system has been designed for ease of modification or not, there are common tasks which must be performed. In order to change a software system, the software engineer performing the task must understand both the system and the changes to be made (Takang & Grub, 1996). The software engineer must be able to verify that exactly the required changes have been made to the software.

Various techniques for handling software evolution have been described in the literature, including those by Takang and Grub (1996) and Pigoski (1996). Takang and Grub describe several software life-cycle processes, and put each in the context of evolving software systems, while Pigoski takes a more evolution-centred approach, concentrating more on the processes which occur after the development of a software system. Pigoski describes software evolution processes, metrics, and management issues.

While developing software which is easy to change is not entirely removed from changing so-called ‘legacy’ software, it is sufficiently different to merit separate treatment. Various techniques for creating software have been described. These range from product-oriented guidelines for developing

understandable source code (McConnell, 1993; Kernighan & Pike, 1999) to processes with attempts at psychological grounding in program comprehension (Smith, 1999).

There have been several attempts to categorise methods for dynamically changing software at run-time. These include simple techniques based on plugins (i.e., dynamically loadable modules) and parameter alteration (Rubini, 1997), and more sophisticated approaches based on component replacement or adaption (Bihari & Schwan, 1991; Segal & Frieder, 1989).

The lack of explicit representation of communication in a software system causes problems with the evolution of the system; communication is a key part of a software system, and should be explicitly represented rather than implicitly inferred. Maintaining the existence of connectors through to the run-time instantiation of the code allows connectors to encapsulate more information about the communication that occurs between components, to contribute to the mobility, distribution and extensibility of systems, and to act as domain translators (providing mappings from messages in one format to messages in another) (Oreizy, Rosenblum, and Taylor, 1998).

The initial design of a modern system usually aims to have low inter-component coupling. This coupling between modules increases as a system is maintained (Lehman, 1998). Whatever the initial architecture of a software system, maintenance of the system without regard to the effects on the architecture will cause degradation of architecture (Lehman, 1996). There are several ways to tackle the problems here:

- Use a process of maintenance that pays explicit and careful attention to the architecture of the system.
- Design the architecture of the system in such a way that maintenance can be carried out in a way that preserves the structure and 'cleanliness' of the system.

When building a software system of significant size, reuse of existing pieces of software is desirable. Usually, unless the components have been specifically designed to work together and do not violate each others' assumptions, simple composition of components is not possible. Each component will make different assumptions about the environment and the behaviour of other components in the system, leading to so-called architectural mismatch (Garlan, Allen, and Ockerbloom, 1995). The most common approach to tackling this mismatch is to 'wrap' components (commonly by inserting 'glue' code between them) to insulate them from each other and to transform the input and output (Shaw, 1995).

One approach to architectural reuse is the concept of product-line architectures. These provide the opportunity to reuse parts of previously existing systems in later software, though this requires a significant amount of work to achieve, and is hard to perform after-the-fact (Bosch, 1999).

Use of the C2 architectural style (Oreizy et al, 1998), which is based on a layered system of components and connectors, has been claimed to ease run-time software evolution; evolution without re-compilation of the

system, in such a way that the system retains its integrity without becoming successively brittle over modifications (Oreizy & Medvidovic, 1998). Two types of system change are identified: changes to the system requirements, and changes to the implementation that do not affect the requirements.

Work on run-time architectural evolution has, in general, concentrated on providing the ability to dynamically replace components. This typically requires provision to be made at design-time (Amdor, de Vicente, and Alons, 1991; Oreizy, 1998).

Distributed systems offer further challenges and opportunities. Large distributed (and other) systems may need to remain functional for long periods of time without interruption. In order to tackle this, Kramer and Magee (1985) propose replacing traditional (build-time) static configuration with incremental dynamic reconfiguration. This requires a greater separation between programming (implementation of behaviour) and configuration (implementation of composition), and requires a configuration language distinct from the programming language(s) used in the system. The more recent C2 architectural style advocates explicit representation of connectors, which provides the ability to abstract away from distribution and to insulate components from changes occurring in other parts of the system (Oreizy & Taylor, 1998).

There are several approaches to handling the evolution of software system. These fall into the two categories of process-oriented solutions and product-oriented solutions. The GENESIS platform supports process-oriented

software evolution, while the CoDEEDS project's aims are to assist with the maintenance of knowledge about software engineering products that have been developed collaboratively.

SUPPORT FOR COLLABORATIVE DEVELOPMENT

In order to realise the above approaches and models in practice, software engineering support environments with explicit provision for evolutionary design of component-based systems are required. Below, two complementary projects are described in greater detail.

The CoDEEDS Project

The CoDEEDS project is concerned with the Collaborative Determination Elaboration and Evolution of Design Spaces (Boldyreff, Kyaw, Nutter, and Rank, 2003b; Boldyreff & Kyaw, 2003). It provides support to design teams enabling them to record their determination of the solution space in the development of large complex distributed systems composed of heterogeneous software components. The result is a potentially N-dimensional design space layered by static and dynamic views of the component sub-systems and models of their deployed instances within the system being designed and deployed in practice. The design environment being developed as part of the CoDEEDS project supports collaborative design throughout the system lifecycle with an agent-based architecture to support design team in their various activities.

Different members of the design team may employ their own preferred design methods and tools when carrying out the detailed design work. The CoDEEDS environment provides a global view of the overall design of the system and the various design decisions that have been made in its composition from a number of potentially heterogeneous components. Figure 1 indicates the primary areas (use cases) supported by the GENESIS and CoDEEDS systems: it shows both the overlapping and discrete primary areas addressed by each system.

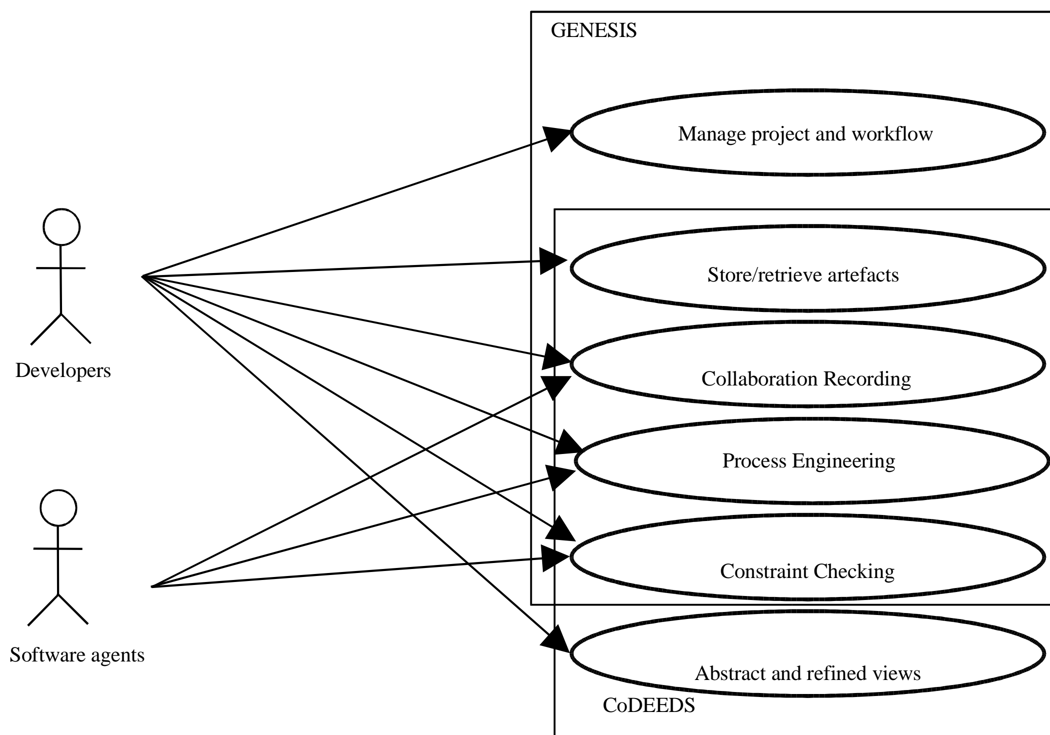
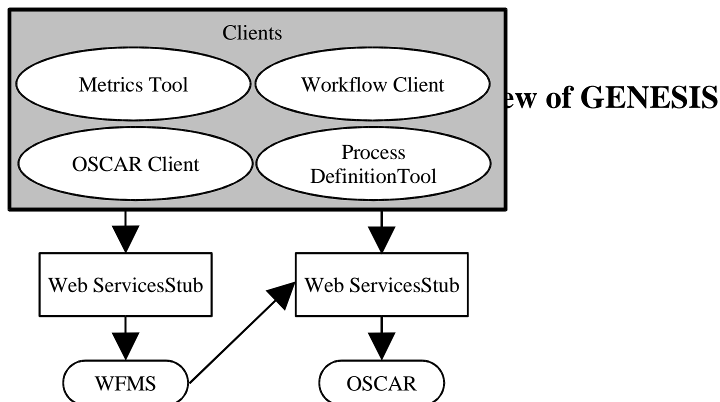


Figure 1 GENESIS and CoDEEDS Overlap

The GENESIS Project and OSCAR

The GENESIS project is focused on the development of a Generalised Environment for Process Management in Co-operative Software Engineering. In the context of figure 1 it addresses the needs for process and work product management. It is employed at both the project management and process workflow level. It complements the design rationale capture of the CoDEEDS system through its support of process engineering and collaborative activity recording. The GENESIS project has developed a low-overhead platform to support collaborative software engineering. The system has been designed to be process *aware,* but *non-intrusive*; like CoDEEDS, it does not mandate methods and tools to be employed by the development team. GENESIS is now an open Source project that was seeded by initial closed-source developments by the project partners.



ew of GENESIS

GENESIS, outlined in figure 2, provides a solution for modelling and enacting workflow processes, and for managing both planned and unplanned work products. The process enactment is distributed over multiple physical sites coordinated by a global process at one site. Both local and global processes are managed via the GENESIS workflow management system.

Underlying both the GENESIS platform and the CoDEEDS system is an artefact management system, OSCAR, which acts as a repository for all artefacts resulting from development. OSCAR supports the creation, storage, retrieval, and presentation of artefact data elements and their associated meta-data. “Everything is an artefact” is the view of the repository’s data; this results in a simplified data model throughout OSCAR. By using Castor, an open source data-binding framework, in the implementation of OSCAR, the ability to treat artefacts as objects and documents simultaneously has been achieved allowing for flexible processing and extension of artefacts and their associated types. The actual storage of the artefact content is achieved through plug-ins to external storage mechanisms such as CVS. An abstraction over software configuration management (SCM) is currently mapped to a CVS plug-in and a plug-in for the Perforce SCM system is under development. Similarly plug-ins for searching are possible, such as the GENISOM extension described in the following section. Instrumentation to collect data about the users and system activities provides the basis for awareness extensions also described in the following section, and potentially for studies of collaborative working in the future.

Currently OSCAR is shipped with the following set of basic artefact types:

- **Software –** specifications, designs, code, etc.
- **Annotation –** any additional information such as email messages and other discussion that may help users of the original artefact
- **Human Resource** – description of the relevant software engineering personnel
- **Project** – workflow models and enactment descriptions
- **Default** – all artefacts are extensions of this

The user may extend this default set of types, at present new types may only be added to the system when the server is started. In particular, the CoDEEDs and GENISOM projects expanded the set of artefact types for their own purposes.

OSCAR's restrictive RMI interface is being complemented with a more accessible Web Services interface to ease deployment of the system in user environments where access through a firewall is necessary. This alternative interface will be useful to industrial users of OSCAR and to users in Open Source projects (Boldyreff, Lavery, Nutter, and Rank, 2003a).

Extending Artefact types

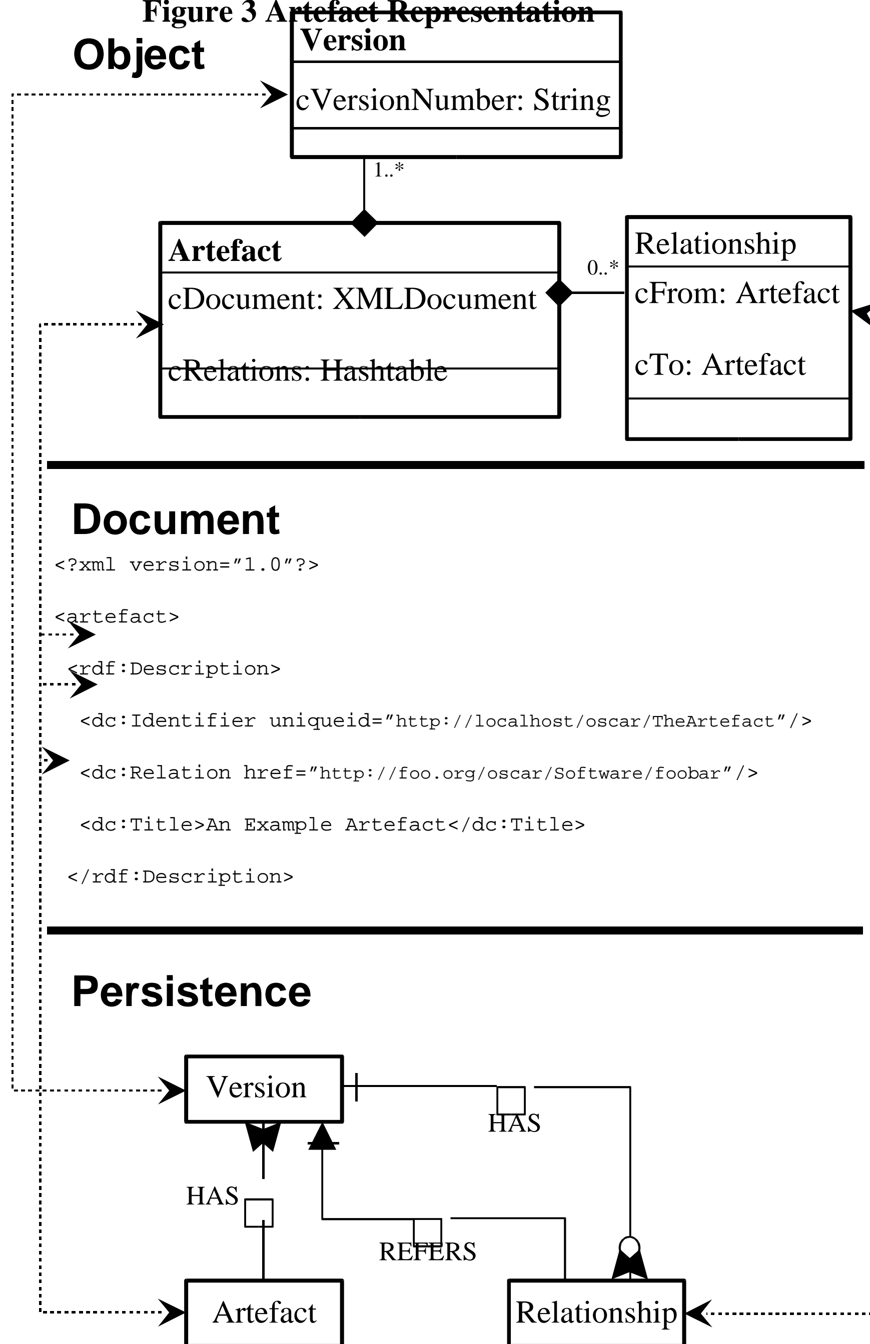
To extend the set of types, two things are required: a set of Java classes derived from the base artefact type containing the functionality provided by

the new type and a Castor mapping file to translate between instances of the class and an XML document. Once the new type has been written and tested the OSCAR server must be reconfigured and restarted to recognise the new type. Users may then create and modify instances of the type like any of the basic types.

There is a faster way to extend the set of “types”: that is to use the default classes and mapping file but under a new name. Obviously the user gains no new features by doing this but can differentiate a set of otherwise similar artefacts by changing the type name without needing to spend time developing new classes. For example, the CoDEEDS project first used the default artefact type under the name “CoDeedsArtefact” before writing classes and mappings for an artefact that provided features necessary for the project, which then replaced the default type.
Figure 3 illustrates how the XML-based representation of artefacts forms a link between the higher-level human-understandable representation, which is rendered as a Java object describable in UML, and the lower-level database (entity-relationship) representation, which is used to provide persistence

1.1.1

Figure 3 Artefact Representation



.

BasicArtefact Operations

Currently high level artefact operations exist for automatic indexing to support search and retrieval, and for various transformations to allow for

flexible presentation of artefacts to users, usually as an XML document, sometimes as an object. Also, basic facilities common to all artefacts exist, including the ability to query and modify the basic metadata, store data within an artefact, store and retrieve versions of an artefact or collection of artefacts and make relationships between artefacts.

EXTENDING OSCAR WITH ADDITIONAL REPOSITORY SERVICES

We describe two additional services that are part of OSCAR alongside the basic management facilities described previously.

Historical Awareness

The possibility of extending OSCAR with historical awareness arises along with the cross project historical data that is captured as OSCAR is used to support a number of projects and as data sharing between distributed OSCARs is realised.

Historical awareness deals with a collection of heterogeneous artefacts allowing the user to view the complete context of an artefact's creation and history of changes into its present form across a number of projects rather than a contextless view of changes to a single project artefact (Nutter & Boldyreff, 2003). Historical awareness is superficially similar to change logs and history views provided by SCM systems but, unlike these systems, provides information that has not been explicitly requested by the user. One way of displaying historical data is via a timeline relating the changes made

to an artefact by various users over time. Such a display can be driven by events as they occur providing immediate feedback to developers sharing an artefact across projects or within a single project. In effect, through historical awareness, users gain a view of the software artefact's evolution over time and across a number of uses within various projects.

The implications of supporting component reuse via this feature are that historical awareness may be able to provide potential users of the component with the big picture of the component's development over time necessary for program comprehension, which must precede effective reuse and evolution. It also gives them immediate feedback from other developers reusing the component and possibly adapting or evolving its functionality, thus preventing conflict (Nutter & Boldyreff, 2003).

GENISOM

A prominent problem within the field of Component-Based Software Engineering concerns finding suitable components to reuse. Reusable assets are in abundance over the web and in libraries, but it is extremely difficult to locate reusable software components that are relevant to a particular application. The necessary organisation is often lacking and difficult to achieve given the dynamic nature of such software collections. This problem can also be found where a large evolving software system consists of an ever growing number of components and the management and hence the comprehension of the associated software components tends to become increasingly difficult. In the GENISOM project, we have applied Self-

Organising Maps (SOMs) to a large population of software components and developed various visualisations of the SOMs. Their effectiveness in relation to the organisation of a large software collection and their usage by software engineers wishing to search the collection has been investigated (Brittle, 2003; Brittle and Boldyreff, 2003).

Self-organising maps are an adaptive technique used to hierarchically (in our case) organise a large search space into a two-dimensional array of smaller spaces. The organisation is performed using an unsupervised neural network (Kohonen et al, 2000).

OSCAR's initial large-scale population for demonstration purposes is derived from the packages of the Debian open source project and consists of just over 1500 software artefacts. This population with its extracted metadata has been employed in some experimental studies to gauge the effectiveness of using SOMs to classify large collections of software artefacts in the GENISOM project (Brittle, 2003). In GENISOM, we have replicated Kohonen's original WebSOM (Kohonen et al, 2000) and extended it to the domain of web-based software artefact collections. SOMs are used as a data visualisation technique to support users browsing and searching large collections of data by representing the collection's population as an interactive map, thereby exploiting computer technology and peoples' abilities to comprehend visual representations. Even though reusable assets are in abundance, a growing problem is the ability to actually locate assets that are relevant for reuse. Organisation of a collection is therefore a necessity and the GENISOM project and other research (Merkl,

1998) have come to the conclusion that SOMs are a viable organisational tool that could be used instead of hierarchical or faceted classification. SOMs also provide a virtually automatic organisation process that could save on the costs associated with employing reuse librarians and reduce the amount of time needed to train engineers in the use of the library. More recently, GENISOM has been redeveloped to provide a front-end to OSCAR and the test population has been expanded to include the Java software components that comprise the current implementation of OSCAR (Brittle & Boldyreff, 2003).

The GENISOM maps provide potential component reusers with various views of the software collection. Figure 4 illustrates one view of such a map.

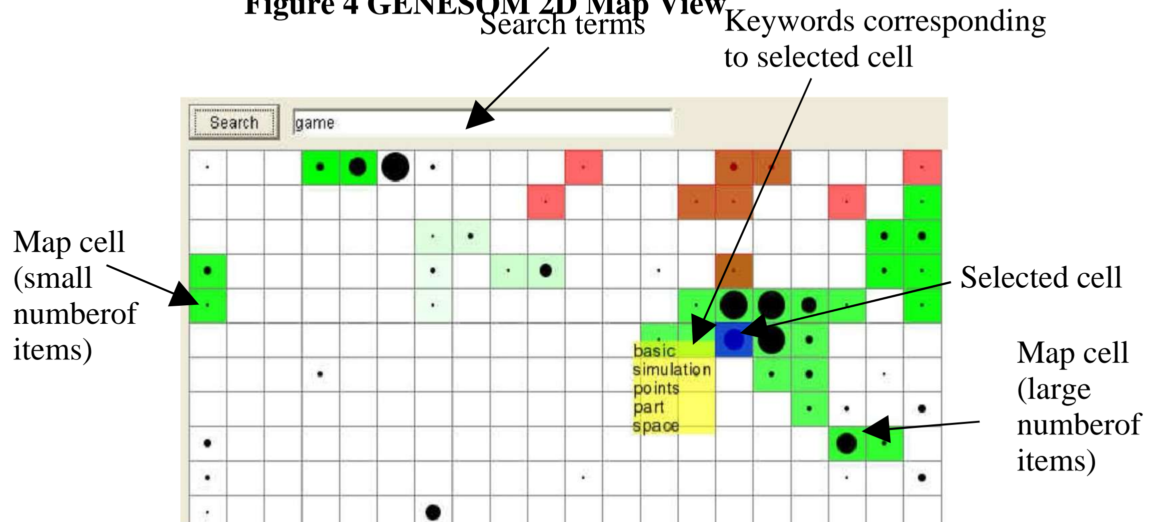


Figure 4 GENESOM 2D Map View

Our preliminary results applying a prototype implementation GENISOM to the Debian and OSCAR components show promise and support our belief that SOMs are an ideal solution to organising the incrementally expanding content of the large distributed repositories that we anticipate will result from OSCAR's usage by a growing number of software development projects.

Extending OSCAR for GENISOM and Awareness.

Extending OSCAR to support both these projects will require modification of both the client and server parts of OSCAR. Though the goals were different, some of the architectural modifications are similar.

GENISOM at first required client-side modifications to generate useful maps from a user's own collection of artefacts in a workspace. These initial modifications required the addition of a new artefact type to describe a particular Self Organising Map configuration and a special artefact type describing Debian package metadata used to represent the test artefact population.

The modifications to the client entailed adding a new user view in addition to the existing hierarchical view of the workspace contents and allowing the user to switch between the views at will. Dialogues to guide the user through the process of creating a self-organising map of the contents of their workspace (and a descriptive artefact) were prepared and added to the client. A tool to extract test artefacts from the Debian packages file was prepared.

Adding awareness support requires modification of the client, though in contrast to the view added for GENISOM this will not allow navigation of the complete workspace; just the parts of the workspace which are affected by the activities of other software engineers. Several server-side modifications are necessary to deliver awareness information to the client: an event handler is required to convert change and dependent change events generated by artefacts into a form suitable for display in the awareness view. This handler will feed the information it creates to the distribution

mechanism, which communicates with the peers in a distributed awareness network.

The awareness network is built by closely linking clients (few hops) working with similar artefact collections; potential algorithms for doing this are described in previous work (Nutter & Boldyreff, 2003). Awareness information messages are then given a time to live (TTL) and sent to the originating client's immediate peers and from there propagated further until the TTL has expired. Since nearby peers will all be using similar artefacts, this approach will ensure that information expires once it becomes irrelevant, keeping the network clear of spurious traffic and removing the need to filter information for relevance on the client.

This method necessarily means that some information will be lost when the network is imperfectly arranged as it will not reach all the clients interested in it. However, the display method outlined for historical awareness can cope with lost information.

DEPLOYMENT

Within the framework of the GENESIS project, the consortium's industrial partners have deployed the GENESIS platform including OSCAR in a number of user trials. Members of the GENESIS project team have used it

to support their own internal development. A stable version of the GENESIS platform is available on SourceForge (at http://sourceforge.net/projects/genesis-ist). The CoDEEDS system is currently a research prototype which is being prepared for release as an open-source system.

The GENESIS platform has been evaluated in the industrial partners' organisations (LogicDIS and Schlumberger) using a comprehensive test bed. In each partner's organisation, the platform was used to model an already-completed project. The project was re-run with the assistance of the GENESIS platform.

Consideration has been given to the adoption of the GENESIS platform by organisations. For large organisations with highly distributed cooperating teams the adoption of a new technology is a complex process that requires an organisation to consider the technology in context of the organisation's business goals (Lavery, Boldyreff, Nutter, and Rank, 2003). Prior to the adoption of GENESIS a large organisation must determine the answers to two difficult questions:Do the existing software processes require additional or improved technical support supplied by GENESIS?

- Does the organisation need to improve their software processes and will GENESIS support that improvement effort?

It is essential to any organisation that the adoption of any new technology is based on the determined needs of the organisation. In the GENESIS project we advocate the use of the Carnegie Mellon Software Engineering Institute's Capability Maturity Model (SW-CMM) (Dewar et al, 2002) to determine

those organisational needs and to support an incremental technology adoption strategy (Lavery et al, 2003).

As GENESIS and CoDEEDS are a collection of distinct systems that work together to provide effective support for the management of both software product evolution and software processes enactment it is possible to introduce the individual systems incrementally based on the determined needs of the organisation.

To ease adoption of the platform, a stand-alone version of OSCAR has been developed and made available. As well as the tools described earlier to up-load the Debian project software; a simple import tool for Java software and other miscellaneous files has been developed. This has enabled the GENESIS project software to be easily transferred into OSCAR as part of the project's own use of its developments.

As with the local and global work processes, the work products managed by OSCAR will soon be visible in a similarly global name-space composed of multiple local OSCAR repositories. Also in progress for OSCAR is user-transparent meta-data extraction and indexing functionality.

It is only with the wide-spread adoption of OSCAR and the development of much larger collections of software artefacts stored in OSCAR that advantages, such as being able to obtain global views of such collections held in distributed repositories, will become apparent.

Instrumenting the tools provided by both GENESIS and CoDEEDS will allow evolution studies of both software engineering processes and products to be performed. Monitoring the real behaviour of projects managed by the GENESIS workflow engine will allow studies of software development processes, indicating how closely real software engineering projects adhere to idealised models. Studying the evolution of products across a number of projects allows a full picture of the development effort to be obtained and may be the basis for predicting future changes.

The architecture of the GENESIS platform currently relies on the relatively tight binding of RMI. This is being transformed to a new architecture based on web services. Once this has been done, the distribution model of the platform will be more flexible. It will no longer be necessary to maintain a strict one-to-one relationship between GENESIS and OSCAR installations; an instance of OSCAR could be shared by more than one GENESIS platform, or a single GENESIS project could use more than one repository.

The industrial partners have evaluated the GENESIS project in real projects. The feedback on the prototype platform that was evaluated has provided motivation for future development in terms of functionality, usability, and interaction mechanisms. The CoDEEDS prototype is also being released as an Open Source project. Feedback from its users will guide its further development.

CONCLUSIONS AND FUTURE WORK

Our initial experimental developments show that GENISOM provides an effective way to organise of a large collection of artefacts. Research is in progress to evaluate visualisation techniques applied to the associated SOMs in terms of their utility to supporting the software reuse by software engineering teams.

The applicability of collaborative technologies and theory to software engineering in the open source environment has not yet been studied. The CALIBRE Co-ordinated Action will provide an opportunity for collaboration experts and Open Source stakeholders to employ tools and techniques for collaboration in highly distributed projects.

We have also proposed a track of research complimentary to the UK E-Science agenda (Boldyreff & Nutter, 2003). The objective of this research programme is to study the needs of collaborators on the scientific grid who will be performing the following activities:

- Designing experiments, much like collaborative design of software
- Replicating or studying previous experiments: data provenance is therefore important
- Collaborating on data analysis, requiring descriptions of scientists working on the system, data sources and full traceability between them.

The eScience agenda itself is very technology focussed, concentrating on the development of technologies for distributed computing and data exchange. However, we believe that collaboration is at the heart and critical

to the success of scientific endeavour and must be considered in any large-scale scientific system for that system to be successful.

This chapter has described two open-source projects which support collaboration using UML and XML. Use of standard representation formats such as these plays a critical role in facilitating software reuse and the evolution of software artefacts. Support is needed for both the process of software engineering as well as the products of these processes. GENESIS provides support for the processes, OSCAR and CoDEEDS provide support for the products. As software engineering matures as a discipline, software reuse has become a more viable option and is becoming a more important part of the software engineer's toolkit. The systems described here support collaborative development per se, and also collaboration across projects at different times, by supporting reuse, aided by common standard representations.

ACKNOWLEDGEMENTS

We wish to acknowledge and thank both James Brittle and Christopher Korhonen for their work with the GENESIS project team. GENESIS was funded by the EU under their IST programme, and CoDEEDS was funded by the UK EPSRC.

REFERENCES

Amdor, J., de Vicente, B., and Alons, A. (1991). Dynamically Replaceable Software: A Design Method. Proceedings of the 3rd European Software Engineering Conference, (ESEC), pages 210-228.

Baxter, I. and Pidgeon, C. W. (1997). Software change through design maintenance. Proceedings of the 1997 International Conference on Software Maintenance (ICSM 97), pages 250-259.

Bihari, T. E. and Schwan, K. (1991). Dynamic adaptation of real-time software. ACM Transactions on Computer Systems, 9(2):143-174.

Bosch, J. (1999). Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. Proceedings of the First Working IFIP Conference on Software Architecture, pages 321-340

.

Boldyreff, A. W. (1954). Systems Engineering. Technical Report P-537, Mathematics Division, The RAND Corporation. 16 June 1954. Available at http://www.dur.ac.uk/cornelia.boldyreff/boldyreff-se.pdf

Boldyreff, C. (1992). A Design Framework for Software Concepts in the Domain of Steel Production. Proceedings of the Third International Conference on Information System Developers Workbench, Gdansk, Poland, 22-24 September 1992.

Boldyreff, C., Burd, E.L., Hather, R.M., Mortimer, R.E., Munro, M., and Younger, E.J. (1995). The AMES Approach to Application Understanding: A Case Study. Proceedings of the International Conference on Software Maintenance, IEEE Computer Press.

Boldyreff, C., Burd, E.L., Hather, R.M., Munro, M., and Younger, E.J. (1996).Greater Understanding Through Maintainer Driven Traceability. Proceedings of the 4th Workshop on Program Comprehension, April 1996, pages 100-106, IEEE Computer Press.

Boldyreff, C., Elzer, P., Hall, P., Kaaber, U., Keilmann, J., and Witt, J. (1990). PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software. Proceedings of Software Engineering 1990 (SE90), Brighton, UK, Cambridge, UK: Cambridge University Press, 1990.

Boldyreff, C. and Kyaw, P. (2003). A Framework for Developing a Design Evolution Environment. Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC).

Boldyreff, C., Kyaw, P., Nutter, D., and Rank, S. (2003), Architectural Framework For a Collaborative Design Environment. Proceedings of Second ASERC Workshop on Software Architecture, Banff, Canada.

Boldyreff, C., Lavery, J., Nutter, D., and Rank, S. (2003), Open-Source Development Processes and Tools. Proceedings of Taking Stock of the Bazaar: 3rd Workshop on Open Source Software Engineering, Portland, Oregon.

Boldyreff, C. and Nutter, D. (2003). Supporting Collaborative Grid Application Development within the E-Science Community. 1st International Workshop on Web Based Collaboratories, collocated with IADIS WWW/Internet, Carvoiero, Algarve, 8th September.

Boldyreff, C., Nutter, D., and Rank, S. (2002a). Open-Source Artefact Management for Distributed Software Engineering. Proceedings of the 2nd Workshop on Open-Source Software Engineering at The 24th International Conference on Software Engineering.

Boldyreff, C., Nutter, D., and Rank, S. (2002b). Active Artefact Management for Distributed Software Engineering. Proceedings of the Workshop on Cooperative Supports for Distributed Software Engineering Processes, in the Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC).

Boldyreff, C., Nutter, D., and Rank, S. (2002c).Architectural Requirements for an Open Source Component and Artefact Repository system within GENESIS. Proceedings of the Open Source Software Development Workshop, Newcastle upon Tyne, U.K. 25-26th February 2002, pp 176-196.

Brittle, J. (2003) Self Organizing Maps Applied to Web Content. Final Year Project Report, Department of Computer Science, University of Durham.

Brittle, J. and Boldyreff, C. (2003). Self-Organising Maps Applied in Visualising Large Software Collections, Proceedings of IEEE VISSOFT.

Dewar, R. G., Mackinnon, L. M., Pooley, R. J., Smith, A. D., Smith, M. J., and Wilcox, P. A. (2002). The OPHELIA Project: Supporting software development in a distributed environment. IADIS WWW/Internet 2002, 13-15 September.

Drummond, S. and Boldyreff, C. (1999). SEGWorld: A WWW-based Infrastructure to Support the Development of Shared Software Engineering Artifacts. Proceedings of the Workshop on Web-Based Infrastructures and Coordination Architectures for Collaborative Enterprises, IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE Press, pp. 120 -125.

Fyson, M. J. and Boldyreff, C. (1998). Using Application Understanding to support Impact Analysis. Journal of Software Maintenance: Research and Practice, 10:93-110.

Gaeta, M. and Ritrovato, P. (2002). Generalised Environment for Process Management in Cooperative Software Engineering. 26th Annual International Computer Software and Application Conference Proceedings, IEEE, pp. 1049-1053.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts. Proceedings of the Seventeenth Interntional Conference on Software Engineering, pages 179-158.

Goguen, J. A. (1986). Reusing and Interconnecting Software Components, IEEE Computer, pp 16-28, February 1986. Reprinted in Tutorial: Software Reusability, edited by Peter Freeman, The Computer Society Press of the IEEE, pp 251-263, 1987.

Jackson, M. (1998). Will there ever be software engineering? IEEE Software, 15(1):36 39.

Kernighan, B. W. and Pike, R. (1999). The Practice of Programming. Addison Wesley Longman.

Kohonen, T., Kaski, S., Lagus, K., Salojarvi, J., Honkela J., Paatero, V., and Saarela, A. (2000). Self Organization of a Massive Document Collection. IEEE Transactions on Neural Networks, 11(3):574-585.

Kramer, J. and Magee, J. (1985). Dynamic configuration for distributed systems. IEEE Transactions on Software Engineering, SE-11(4):424-436.

Kwon, O. C., Boldyreff, C. and Munro, M (1997). An Integrated Process Model of Software Configuration Management for Reusable Components. Proceedings of the Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE'97), June 18-20, Madrid, Spain.

Lavery, J., Boldyreff, C., Nutter, D., and Rank, S. (2003). Incremental Adoption Strategy for the GENESIS Platform. GENESIS Project Report, University of Durham. Available at http://www.dur.ac.uk/janet.lavery/documents/AdoptStratFinal.pdf

Lehman, M. M. (1979). On understanding law, evolution and conservation in the large program life cycle. Journal of Systems and Software, 1:213-221.

Lehman, M. M. (1996). Laws of software evolution revisited. In Proceedings of EWSPT96, number 1149 in Lecture Notes in Computer Science, pages 108-124. Springer-Verlag.

Lehman, M. M. and Belady, L. A. (1985a). Program Evolution: Processes of Software Change. Number 27 in APIC Studies in Data Processing. Academic Press.

Lehman, M. M. and Belady, L. A. (1985b). Programs, life cycles and laws of software evolution. In Program Evolution: Processes of Software Change, number 27 in APIC Studies in Data Processing, pages 393 449.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution The nineties view. In Eman, K. E. and Madhavji, N. H., editors, Elements of Software Process Assessment and Improvement, pages 20 32, Albuquerque, New Mexico. IEEE CS Press.

Lehman, M. M. Software's Future: Managing Evolution. IEEE Software 15 (3):40-44

Lientz, B. P. and Swanson, E. B. (1980). Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley.

McConnell, S. (1993). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.

Merkl, D. (1998). Self-Organizing Maps and Software Reuse. Computational Intelligence in Software Engineering. World Scientific.

Nutter, D. and Boldyreff, C. (2003). Historical Awareness Support and Its Evaluation in Collaborative Software Engineering. Proceedings of the Workshop on Evaluation of Collaborative Information Systems and Support for Virtual Enterprises at the 12th IEEE international Workshops on Enabling Technologies For Collaborative Enterprises (WETICE).

Nutter, D., Boldyreff, C., and Rank, S. (2003). An Artefact Repository to Support Distributed Software Engineering. Proceedings of 2nd Workshop on Cooperative Support for Distributed Software Engineering Processes, CSSE 2003, Benevento, Italy.

Oreizy, P. (1998). Issues in modeling and analyzing dynamic software architectures. In Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Sicily, Italy.

Oreizy, P, Rosenblum, D S, and Taylor, R N. On the Role of Connectors in Modelling and Implementing Software Architectures. Technical report UCI-ICS-98-04, Department of Information and Computer Science, University of California, Irvine,

Oreizy, P, and Medvidovic, M. (1998) Architecture-Based Runtime Software Evolution. In proceedings of the International Conference on Software Engineering, Kyoto, Japan. Pages 19-25.

Oreizy, P. and Taylor, R. N. (1998). On the role of software architectures in runtime system reconfiguration. I.E.E. Proceedings-Software, 145(5): 137-145.

Paulk, M. C., Curtis, B., Chrissis, M. B., and Weber, C. V. (1993). The Capability Maturity Model for Software, IEEE Software, 10(4):18-27.

Pigoski, T. M. (1996). Practical Software Maintenance. John Wiley and Sons.

Rubini, A. (1997). The sysctl interface. Linux Journal, 41. Available at http://www2.linuxjournal.com/lj-issues/issue41/2365.html.

Segal, M. E. and Frieder, O. (1989). Dynamic program updating: A software maintenance technique for minimizing software downtime. Journal of Software Maintenance: Research and Practice, 1(1):59 79.

Shaw, M. (1995). Architectural Issues in Software Reuse: It's Not Just The Functionality, It's the Packaging. In proceedings of the I.E.E.E. Symposium on Software Reusability

Smith, D. D. (1999). Designing Maintainable Software. Springer-Verlag.

Takang, A. A. and Grub, P. A. (1996). Software Maintenance: Concepts and Practice. International Thomson Computer Press.

Wilcox, P. A., Smith, M. J., Smith, A. D., Pooley, R. J., MacKinnon, L. M., and Dewar, R. G. (2002). OPHELIA: An architecture to facilitate software engineering in a distributed environment. 15th International Conference on Software and Systems Engineering and their Applications (ICSSEA), December 3-5, Paris, France.

Zhang, J and Boldyreff, C. (1990). Towards Knowledge-Based Reverse Engineering. Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, Syracuse, NY, 24-28 September 1990.