IT Licentiate thesis
2000-007
MRTC Report 00/24

Applying Configuration Management Techniques to Component-Based Systems

MAGNUS LARSSON

UPPSALA UNIVERSITY
Department of Information Technology

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

UPPSALA UNIVERSITY

Applying Configuration Management Techniques to Component-Based Systems

BY
MAGNUS LARSSON

December 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2000

Applying Configuration Management Techniques to Component-Based Systems

Magnus Larsson
Magnus.Larsson@mdh.se

Department of Computer Engineering
Mälardalen University
Box 883
SE-721 23 Västerås
Sweden

http://www.idt.mdh.se/

ISSN 1404-5117
Printed by University Printers, Uppsala University, Sweden

To
Christina
Emmy, Ida and Jacob

This is a dissertation submitted for the degree of Licentiate of Philosophy in Computer Systems at Uppsala University, December 2000.

ABSTRACT

Building software from components, rather than writing the code from scratch has several advantages, including reduced time to market and more efficient resource usage. However, component based development without consideration of all the risks and limitations involved may give unpredictable results, such as the failure of a system when a component is used in an environment for which it was not originally designed.

One of the basic problems when developing component-based systems is that it is difficult to keep track of components and their interrelationships. This is particularly problematic when upgrading components. One way to maintain control over upgrades is to use component identification and dependency analysis. These are well known techniques for managing system configurations during development, but are rarely applied in managing run-time dependencies. The main contribution of this thesis is to show how Configuration Management (CM) principles and methods can be applied to component-based systems.

This thesis presents a method for analysing dependencies between components. The method predicts the influence of a component update by identifying the components in a system and constructing a graph describing their dependencies. Knowledge of the possible influences of an update is important, since it can be used to limit the scope of testing and be a basis for evaluating the potential damage of the update. The dependency graphs can also be used to facilitate maintenance by identifying differences between configurations, e.g., making it possible to recognise any deviations from a functioning reference configuration.

For evaluation of the method, a prototype tool which explores dependencies and stores them under version control has been developed. The prototype has been used for partial analysis of the Windows 2000 platform. Preliminary experiments indicate that most components have only a few dependencies. The method has thus given an indication that the analysis of the effects of component updates may not be as difficult as might be expected.

Distributed by Department of Computer Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden, and Department of Computer Systems, Information Technology, Uppsala University, Box 325, S-751 05 Uppsala, Sweden

ACKNOWLEDGEMENTS

The research presented in this thesis was carried out within the STINA (Standard Technologies in Industrial Applications) project, which is a cooperation project between Mälardalen University and ABB Corporation.

I sincerely thank my supervisor Prof. Hans Hansson for great support and feedback in response to my ideas.

This work could not have been done without the valuable and important encouragement I have received from my tutor and friend Dr. Ivica Crnkovic. I really appreciate Ivica's efforts in motivating me to prepare this thesis.

This work has been financed by ABB Automation Products through Erik Danielsson. I wish to thank specially Erik Danielsson for giving me this opportunity.

Also thanks to Frank Lüders, Rolf Sundberg and Christina Larsson for criticizing and reviewing this thesis in a constructive way. Without their valuable input this work may not have been completed.

I especially thank Victor Miller for helping me with the English language.

Special credit is due to Peter Ekman for assisting me with the layout of the document.

Thanks to Erik Gyllenswärd for being a good example and for his encouragement during many years as a colleague at ABB.

Magnus Larsson

Västerås November 2000

LIST OF PUBLISHED ARTICLES

The following articles have been published at international conferences and workshops.

These four articles are included in this thesis:

- **Development Experiences of a Component-based System**
  In *proceedings 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems* Edinburgh, Scotland, April 2000. IEEE Computer Society
  Authors: Magnus Larsson, Ivica Crnkovic

- **Component Configuration Management for Frameworks**
  In *proceeding, Asia-Pacific Software Engineering Conference, Workshop on Software Architecture and Components* Takamatsu, Japan, December 1999.
  Authors: Ivica Crnkovic, Magnus Larsson, and Kung-Kiu Lau

- **Component Configuration Management**
  In *proceedings ECOOP Conference, Workshop on Component Oriented Programming* Nice, France, June 2000.
  Authors: Magnus Larsson, Ivica Crnkovic

- **New Challenges for Configuration Management**
  In *proceeding, System Configuration Management, SCM-9, proceedings* Toulouse, France, September 1999. Lecture Notes in Computer Science 1675, Springer Verlag.
  Authors: Magnus Larsson, Ivica Crnkovic

Other articles published by Magnus Larsson:

- **Software Process Measurements using Software Configuration Management**
  In *proceedings The 11th European Software Control and Metrics Conference* Munich, Germany, May 2000.
  Authors: Ivica Crnkovic, Magnus Larsson, and Frank Lüders

- **The Different Aspects of Component Based Software Engineering**
  In *proceedings MIPRO (Microprocessor systems, Process control and Information Systems) Conference* Opatija, Croatia, May 2000.
  Authors: Ivica Crnkovic, Magnus Larsson, and Frank Lüders

- **Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues**
  In *proceedings 4th IEEE international Baltic workshop on Databases and Information Systems,* Vilnius, Lithuania, January 2000.
  Authors: Ivica Crnkovic, Juliana K. Küster Filipe, Magnus Larsson, and Kung-Kiu Lau

- **A Case Study: Demands on Component-based Development**
  In *Proceedings, 22nd International Conference of Software Engineering* Limerick, Ireland, January 2000. ACM, IEEE, SigSoft
  Authors: Ivica Crnkovic, Magnus Larsson

- **State of the Practice: Component-based Software Engineering Course**
  In *Proceedings, Workshop* ICSE 2000 conference, 3rd International Workshop on CBSE, January 2000.
  Authors: Ivica Crnkovic, Magnus Larsson, and Frank Lüders

- **Processing Requirements by Software Configuration Management**
  In *Euromicro 99, proceedings of the 25th EUROMICRO conference* Milan, Italy, September 1999. IEEE, Computer society
  Authors: Ivica Crnkovic, Peter Funk, and Magnus Larsson

- **Managing Standard Components in Large Software Systems**
  In *Proceedings on 2nd workshop on Component Based Software Engineering* Los Angeles, USA, May 1999.
  Authors: Ivica Crnkovic, Magnus Larsson

CONTENTS

THESIS

1 Introduction

Software systems are becoming increasingly complex and providing more functionality. To be able to produce such systems cost-effectively, suppliers often use component-based technologies instead of developing all the parts of the system from scratch. The motivation behind the use of components was initially to reduce the cost of development, but it later became more important to reduce the time to market, to meet rapidly emerging consumer demands. At present, the use of components is more often motivated by possible reductions in development costs. By using components it is possible to produce more functionality with the same investment of time and money [14]. When components are introduced in a system, new issues must be dealt with e.g. dynamic configurations, variant explosion and scalability. Some of these issues are addressed with the discipline Component-Based Software Engineering (CBSE). CBSE provides methods, models and guidelines for the developers of component-based systems. Component-based development (CBD) denotes the development of systems making considerable use of components.

Although very promising, CBSE is a new discipline and there are many associated problems which remain unsolved. Many solutions can be arrived at, by using principles and methods from other engineering disciplines, such as configuration management. This thesis describes some of these disciplines, presents proposals and analyses possibilities of applying different methods in CBSE.

The main topic of this thesis, component management, is discussed in three stages: configuration management, component-based software engineering and the application of configuration management to component based systems. This thesis also discusses experiences from software development collected in several of the included articles.

An overview of CBSE and its different aspects is presented in section 2. This section summarise the state of the art including certain parts from our work.

The management of components in a product is an important subject. Configuration management (CM) is used to manage the development of complex systems. CM covers version, change, build, release and workplace management. An overview of configuration management is given in section 3.

Systems which support on-line updating of components are difficult to manage due to the dynamic nature of the components as described in section 4. This section briefly summarises the problems which occur in dynamic configurations. More detailed

descriptions can be found in the articles included, and particularly in “New challenges for Configuration Management”.

When updating a system during run-time, it is very difficult to predict which parts of the system that will be affected. A system crash due to a component update is in general unacceptable. Section 5 shows how to manage components, by applying ideas from the configuration management area.

In section 6 future work is outlined and in section 7 some conclusions are drawn.

Four articles are included as complement to this thesis. The first, “Development Experiences from a Component-Based System” discusses the different levels of component reuse and certain experiences gained from analysing the lifecycle of a system for process control. The second article, “New Challenges for Configuration Management” analyses management of components and highlights problems related to configurations of components. The third article, “Component Configuration Management” presents and discusses a proposal for the treatment of dependencies between components. Finally “Component Configuration Management for Frameworks” explains the framework concept using a COM implementation as an example of how frameworks can be realized.

1.1 Method

The research has been conducted with a survey of different technologies and methods used in component-based software engineering. The articles presented in this thesis are based on practical experience from both successful and less successful projects.

From the survey of component technologies, the lack of management of components in both industrial and academic systems can be seen. This thesis combines the theory of and experience from the configuration management area with the management of components.

A prototype has been developed to manage the interaction between components on the Windows™ platform. This prototype has proven useful for developers at ABB, since it allows them to keep track of the system behaviour by tracing component dependencies.

1.2 Related work

There is much ongoing research in the areas of configuration management, components, distributed systems and dynamic reconfiguration. The work which has most influenced this thesis will be outlined here. Additional related work is presented in each section and in the enclosed papers where appropriate.

The component management prototype presented in this thesis is inspired by a model for predictable system updates presented by Cook and Dage [20]. The system chooses

the appropriate component to run in relation to the input domain. Cook and Dage's model does not consider dependencies between components. Another disadvantage of their model is that the arbitrator, the instance which decides which component will run in relation to the input, must be integrated in the execution environment. Cook and Dage's model relates to this thesis by means of the usage of multiple versions of components concurrently.

This thesis presents methods to analyse the dependencies between components. Traditional graph theory, as presented in [31,53], is used to organise the dependencies between components. The dependency graphs can be represented with matrices or adjacency lists. For calculating all direct and indirect dependencies in a system, different algorithms for transitive closure are used. Examples quoted are three algorithms from Warshall, Nuutila, and Eve and Kurki-suonio [28,43,59].

Voas [57] discusses the possibility of certifying components to assure the quality of the system. An important question to be answered before using a component in a system is: Does the component have a positive impact on the system? To be able to answer this question certain analyses of the system must be performed. It is important especially to know the dependencies between components in the system. This thesis presents a model for performing dependency analyses which can be utilised in implementing the Voas certification strategy.

1.3 Contribution

The main contribution of this thesis is a method which can be used for managing component dependencies when updating systems with new components. The problems of dynamic configurations and the importance of solving them to provide controlled updates of systems is presented. The prediction of the effects of updating the system is crucial for all types of computer-based systems and especially for safety-critical systems.

The method for managing interactions between components in a controlled way with software configuration management is discussed in all chapters, but principally in sections 4, 5, 6 and in the included articles.

A prototype which explores dependencies and stores them under version control has been developed to implement the configuration model. The prototype has been used to analyse the Windows 2000 platform and its components.

Experience from real projects is needed to develop component-based systems. A gathering of such experience from software development using components is presented in a number of articles, which point out the existing problems with component-based software engineering. Four of the published articles have been selected to be included in this thesis. A short introduction to these articles is presented below.

- **Development Experiences from a Component-based System**
  Presents experiences and background to the problems encountered in developing component-based systems. Published in *Proceedings 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems* Edinburgh, Scotland, IEEE Computer Society, April 2000.
- **New Challenges for Configuration Management**
  Presents the problems encountered in run-time configuration management and certain proposals for managing components. Published in *System Configuration Management, SCM-9, proceedings* Toulouse, France, Lecture notes in computer science 1675, Springer Verlag, September 1999.
- **Component Configuration Management**
  Introduces the idea of configurations placed under configuration control. Published in *Proceedings, Asia-Pacific Software Engineering Conference, Workshop on Software Architecture and Components* Takamatsu, Japan, December 1999.
- **Component Configuration Management for Frameworks**
  Shows how object oriented frameworks can be implemented with components using COM. Published in *ECOOP Conference, Workshop on Component Oriented Programming* Nice, France, Springer workshop reader, June 2000.

The different articles are presented below with a short summary of their respective contribution and presentation of the specific contribution of each author.

1.3.1 Development Experiences from a Component-based System

Building software systems with reusable components has many advantages. If the reuse concept is utilized on several levels of a system development, the development becomes more efficient, the reliability of the products is enhanced, and the maintenance requirement is significantly reduced. In this paper the different levels of component reuse, and certain aspects of component development are discussed. As an illustration of reuse issues, a successful implementation of a component-based system used for industrial process control is presented. The experience can be summarised in that it is better to invest more effort in creating an open and extendable architecture than to focus on the development of only current technologies.

Magnus Larsson contributes to this article with knowledge about the case study and the background, an analysis of different reuse aspects and the replacement of proprietary components with standard components. Ivica Crnkovic wrote the sections about requirements and development environment.

1.3.2 New Challenges for Configuration Management

When moving from monolithic to open and flexible systems a new issue relating to dynamic upgrades of components is introduced. It is important to have up-to-date

information about which components are installed in a system. To know this is a general problem since it is difficult to obtain configuration information from each component. Configuration management has been traditionally focused on the development phase; in particular, on managing source code, but now, when changes are introduced in component-based systems at run-time, new methods are needed. This paper identifies these kinds of problems and proposes configuration management routines for libraries and version interfaces for components.

Magnus Larsson contributes with background information about the problems of run-time configuration management and the sections about managing components and proposed CM for libraries and components. Ivica Crnkovic wrote the section about component compatibility.

1.3.3 Component Configuration Management

The problem of component identification is discussed in this paper. The components are usually binary units deployed in the system at run-time and the insight into their characteristics is not as clear as into those of the software units which we manage at development time. For external components, extensive tests can, to some extent, compensate for the lack of information. When the information about the components is gathered, it is possible to keep track of changes introduced in the system and their impact on the system. The change management process is similar to that for configuration management. This paper makes proposals for managing dependencies during run-time using software configuration management principles. The proposal is to identify component dependencies in a system and to have a tool able to browse these.

Magnus Larsson wrote the sections about component dependencies and the browser. Ivica Crnkovic presented the introduction and the section about component management and SCM was a joint production.

1.3.4 Component Configuration Management for Frameworks

Object-oriented design frameworks are increasingly recognized as better components than objects. Frameworks can be expressed formally and later verified formally. An object in a framework can have multiple roles which are mapped in component interfaces. Objects can have multiple roles in different frameworks. This paper shows how object-oriented design frameworks can be implemented in COM and presents a discussion about how frameworks can be composed in a controlled way using configuration management techniques.

Magnus Larsson provided the COM implementation of frameworks and wrote the section on configuration management issues in collaboration with Ivica Crnkovic. Kung-Kiu Lau presented the ideas of frameworks and how to formalise them.

2 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is the discipline of developing components and developing products with components. Products are no longer developed from scratch; they are instead an assembly of components developed independently of the products. This means that components are developed without complete knowledge of their execution environment. To assembly components, proprietary code, which glues the components, is usually needed. This code is often referred to as "glue code" [18] and in certain cases, the glue may take a longer time to develop than the components concerned.

Object-oriented programming (OOP) had the same approach; objects were reusable entities that could be assembled as programs. Component-based development can be seen as an extension of object-orientation, but takes one step further. OOP binds the implementation to a particular class library and language. Smalltalk is one example in which the programmer is bound to the Smalltalk language and the classes provided in the environment. Components, on the other hand, are generally not bound to a particular language and they communicate through independent interfaces.

One common characteristic of object technologies and component technologies is that there are as many definitions of objects as there are of components. These definitions are elaborated upon in this section. Component-based systems are usually implemented within a particular component model. Different component models such as COM, EJB and CORBA will be described in section 2.2.

Patterns, as described in section 2.3, can be used to facilitate the understanding of the architecture of a system built from components. Patterns can also be reused in subsequent projects since the same architectural solutions often reappear.

This section ends with a description of component interfaces, followed by a discussion of component-based development.

2.1 Component Definitions

"Components are for composition. *Nomen est omen*". [54] This is a quotation with which most people agree when discussing the nature of components. But to develop a precise and well-understood definition of a component upon which everybody agrees, is not an easy task. Many have tried, but the result is a flora of definitions which all differ slightly. Wallnau [17] presents four main definitions, representative of those emerging in the software industry.

1. A component is a non-trivial, nearly independent, and replaceable part of a system which fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

2. A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces which can be detected during run-time.
3. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party.
4. A business component represents the software implementation of an "autonomous" business concept or business process. It consists of all the software artefacts necessary to express, implement and deploy the concept as a reusable element of a larger business system.

Sametinger [50] defines a reusable component as a component which is a self-contained, clearly identifiable artefact which describes and/or performs specific functions and has clear interfaces, appropriate documentation and a defined reuse statement.

There are other ways of classifying components, for example, as internal and external components. Internal components are developed within an enterprise for use by internal projects. Usually these components encapsulate the core competence of the company. External components, on the other hand, are developed by a third party and usually for the general market. Examples of such components are generic user interface components and communication protocols. Product line components are a component category designed to fit into a flexible architecture which can be configured for different products within the product line [14]. There are also different levels of components as mentioned in [40], from small library components to large product components.[23]

These different definitions of components are presented to demonstrate that it is not easy to arrive at a generally acceptable definition. However, before a component-based system can be designed, a specific definition must be agreed upon to establish the context for the developers.

Common to all these definitions is that a component shall provide interfaces, conceal the implementation and be independently deployable.

Szyperski's definition [54] of a component is used in this thesis. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party.

2.2 Component Models

Component models are sometimes called component frameworks but because of the risk of confusion with other definitions of frameworks, the term model is used in this thesis. Models give significant support to the user by managing the infrastructure, making it possible for the user to concentrate on meeting the requirements without needing to develop the infrastructure. In a sense, the models themselves can be seen as components since they are used to minimize the development effort. However, they do not express external dependencies and are designated infrastructure components. Other examples are databases or operating systems.

Three major component models are used successfully today: COM [4], JavaBeans [7] and CORBA [6]. All have different levels of service for the application developer. Table. 1 shows the corresponding technologies for each level of service.

	COM	**Java**	**CORBA**
Basic components	COM components	JavaBeans	CORBA objects
Distribution	DCOM	RMI	CORBA IIOP
Enterprise services	COM+	EJB/J2EE	CORBAServices

Table. 1. The different technologies used at different levels of service

Distribution is by means of a communication protocol added to the basic component model. COM uses Distributed COM (DCOM), Java has Remote Method Invocation (RMI) and CORBA uses the Internet Inter-ORB Protocol (IIOP). Support for business components is available in COM+, EJB and CORBAServices.

There are differences between systems with components tightly coupled together and those with loose references between the components. Tightly coupled systems are tied together at build-time or with strong references, e.g. a shared library. With loose references, the components are connected to their fellow components as required and not during the build phase. For such systems, it is much more of a challenge to determine, when starting, the final appearance of the system. To be able to predict the behaviour we need to know which components will cooperate. All three models presented in this section are loosely coupled with support for dynamic invocation and lookup.

The use of component models can be an appropriate way of beginning component development. If components are developed independently, it is highly unlikely that they will be able to cooperate usefully [30], because of a probable mismatch in the

requirements. The primary goal of component technology, independent deployment and assembly of components will not be achieved.

A component model supports components by requiring them to conform to certain standards and permitting instances of these components to cooperate with other components conforming to the model.

The key contribution of component models is the partial enforcement of architectural principles. By forcing component instances to perform certain tasks, the component model can enforce policies. E.g. a component model might enforce some ordering on event multicasts and thus prevent entire classes of subtle errors caused by glitches or races which might otherwise occur.

2.2.1 Component Object Model (COM)

The Component Object Model [4], from the Microsoft Corporation, provides a model for designing components with multiple interfaces with dynamic binding to other components. COM is also a run-time environment, unlike CORBA, which is only a specification. COM is an open standard which has been implemented on many different platforms, but it has mainly been used on Microsoft Windows for which it was first developed. Components expose themselves through interfaces only. The interfaces are binary which makes it possible to implement the component in a variety of programming languages such as C++, Visual Basic and Java. A COM component can implement and expose multiple interfaces. Figure 1 shows how a client uses COM to locate the server components and then to request the required interfaces using the `QueryInterface` method.

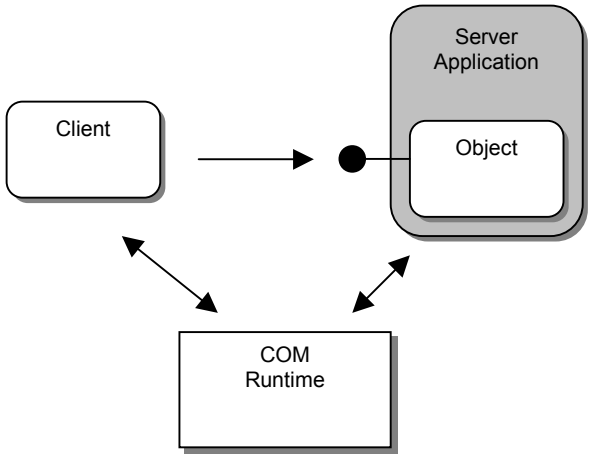


Figure 1. COM establishes the connection between client and server and the client communicates directly with the server.

DCOM is the protocol used to make COM location transparent. The client talks to a proxy, representing the server and manages the real communication with the server.

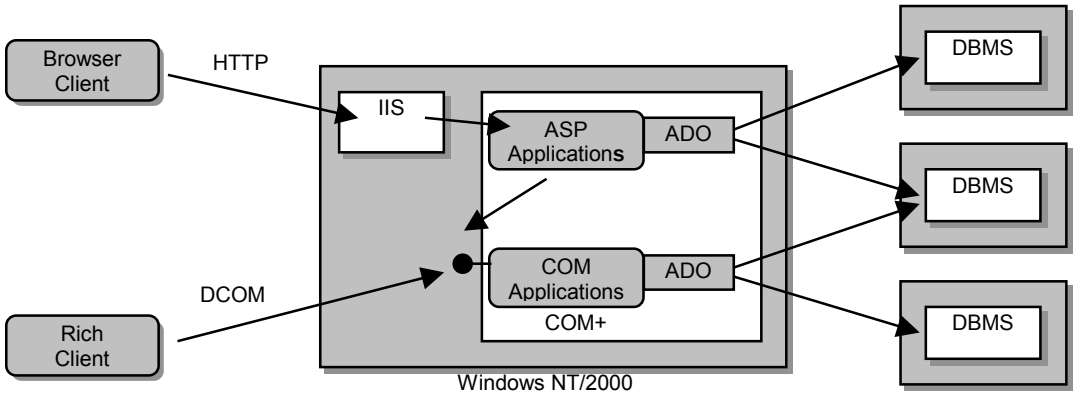


Figure 2. The principle of the use of COM+ in a three-tier architecture.

COM+ [4] is an extension of COM which includes support for transactions, directory service, load balancing and message queuing. Figure 2 shows how clients can connect, through an Internet Information Server (IIS) or DCOM, to the business logic, which is implemented with COM+. The business logic uses ActiveX Data Objects (ADO) to access the data in the databases. Compare this picture with the EJB technologies illustrated in Figure 3 to see the similarities.

2.2.2 Enterprise Java Beans (EJB)

Enterprise Java Beans [7], from Sun Microsystems, is a component-architecture for server-side components used to build distributed systems with multiple clients and servers. A Java Bean is a reusable component which supports persistency and can interact across all platforms supported by Java. EJB uses Java Beans but is much more than a component model. EJB provides support for transactions and security over a neutral object communication protocol, which gives the user the opportunity to implement the application on top of a protocol of choice. EJB is part of the Java 2 Platform Enterprise Edition (J2EE) [49] which includes many other technologies such as remote method invocation (RMI), naming and directory interface (JNDI), database connectivity (JDBC), server pages (JSP) and messaging services (JMS).

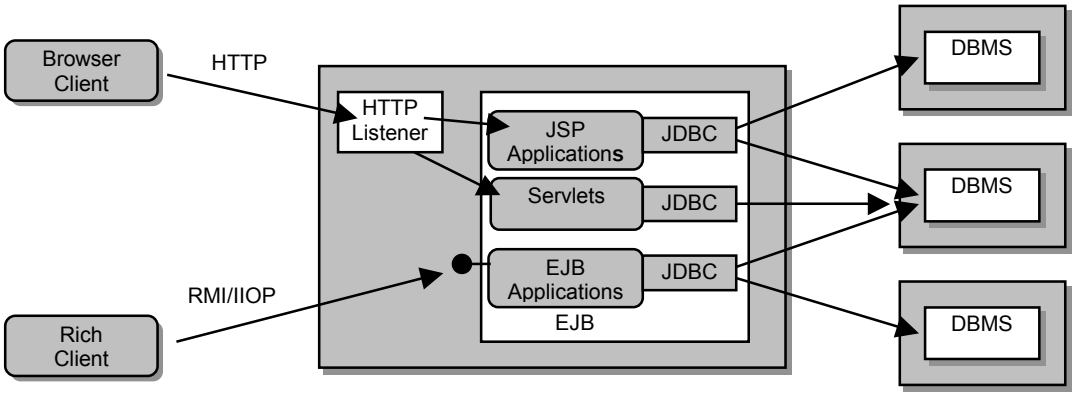


Figure 3. The principle of the use of EJB in a three-tier architecture.

Figure 3 shows the architecture of a three-tier application using EJB. The clients connect to the server components through either a web server or directly, using remote method invocation (RMI). The server components which implement the business logic reside within an EJB container with support for transactions and security. The data is stored in databases which are managed by a database management service (DBMS) and are accessed through the data base connectivity component (JDBC). Java server pages (JSP) or servlets are used when thin web clients access the system through the Internet. Compare Figure 3 with Figure 2 to see the similarity of the technologies used in the COM+ environment.

A JavaBean is a special case of an ordinary Java class. To make a JavaBean an Enterprise bean the JavaBean must conform to the specification of EJB by implementing and exposing a few required methods. These methods enable the EJB container to manage beans in a uniform way for creation, transactions etc. The clients of an enterprise bean can vary widely, for examples a servlet, an applet or another enterprise bean. Since enterprise beans may call each other, a complex bean task might then be divided into smaller tasks and handled by an hierarchy of beans. This “divide and conquer” procedure is very efficient.

There are two different kinds of enterprise beans: session and entity beans. Session beans live as long as the client code which calls it. They represent the business process and are used to implement business logic, business rules and workflow. Entity beans model data and are often used by session beans to represent the data they use.

EJB is designed to interact with CORBA implementations and access CORBA objects transparently. There is also a bridge between COM and EJB which can be used to make systems even more open. The *adapter* pattern [29] can normally be used to implement various bridges between object models.

2.2.3 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) [6] is a standard developed by the Object Management Group (OMG) at the beginning of the nineties. The OMG supplies industry guidelines and object management specifications to provide a common framework for integrating application development. Primary requirements for these specifications are reusability, portability and interoperability of object-based software components in a distributed environment. Figure 4 shows how CORBA is part of the Object Management Architecture (OMA), which covers object services, common facilities and definitions of terms.

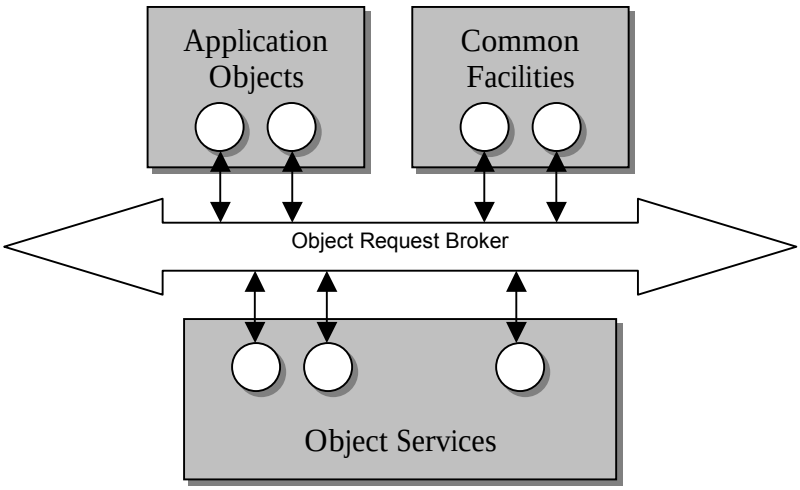


Figure 4. The parts of the Object Management Architecture.

Object services include naming, persistency, events, transactions and relationships. These can be used when implementing applications. Common facilities provide general-purpose services such as information, task and system management. All services and facilities are specified in IDL. An object request broker (ORB) provides the basic mechanism for transparently making requests and receiving responses from local or remote objects. Requests can be made through the ORB irrespective of the service location or implementation. Objects expose their interfaces using the Interface Definition Language (IDL) as defined in the CORBA specification.

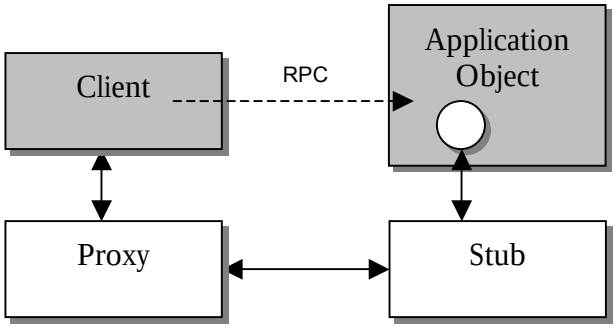


Figure 5. Clients communicate transparently with the server with Remote Procedure Calls (RPC).

Objects are stored in an interface repository where they can be found and activated on demand from the clients. Figure 5 shows how the client communicates with the server through remote procedure calls (RPC). RPC is a location-transparent way to communicate; the real communication is by means of proxies and stubs. The stubs and proxies are generated from the IDL specification which each object provides for its interfaces.

2.2.4 Comparison of COM, EJB and CORBA

To be able to compare the different technologies it is important to compare them at the same level of service. Since the basic level of EJB is JavaBeans, JavaBeans is to be compared with COM and CORBA. Many features of the component models presented are similar but there are also differences. To explain the differences, a short comparison of the three models is presented in Table. 2.

Feature	**COM**	**JavaBeans**	**CORBA**
Instantiation	Dynamic	Dynamic	Dynamic
Distribution	DCOM	RMI/IIOP	IIOP
Interfaces	Multiple	Single	Single
Interface Inheritance	Single	Multiple	Multiple
Basic rule of implementation	Every object implements IUnknown	Every object implements Java.rmi.remote	Every interface inherits from CORBA.Object

Feature	COM	JavaBeans	CORBA
Dynamic invocations	Dispatch interfaces	Dynamic wiring with reflection and introspection	Dynamic Skeleton Interface (DSI) and Dynamic Invocation Interface (DIII)
Type information	Type libraries and ITypeInfo interface	Introspection	Interface Repository
Mapping of object name to implementation	Handled by the Registry	Handled by the RMIRegistry	Handled by the Implementation Repository
Event model	Advice sink model	Components declare outgoing events and registers for incoming events.	Event Interfaces
Platforms	Primarily Microsoft but also available on all major platforms.	Independent, Needs Java environment	Each vendor of CORBA implementations provides products for different platforms.
Security	User authentication and authorization, data encryption and integrity checking.	User authentication and authorization.	Secure domains, data encryption and integrity checking.

Table. 2. A comparison of concepts in COM, EJB and CORBA

The different models have advantages and disadvantages and it is therefore difficult to determine which is the “best”. When a system is designed a range of models must be considered, depending on what us required of the product. Often when the platform is chosen, the selection of component model is not so difficult. For instance if the target environment is C++ on top of a mix of UNIX and Windows platforms, CORBA is probably the best choice. In the same way EJB is more suitable for a heterogeneous Java language environment. A third example is the choice of COM when the target is Windows based PC machines.

2.3 Patterns

Design patterns were introduced to permit the reuse of successful designs [29]. Each pattern describes a problem, which occurs repeatedly in the environment, and then describes the core of the solution to that problem. By means of design patterns, knowledge of good software design can be documented and the experience gained within software projects becomes widely applicable. With design patterns, a common design vocabulary is introduced, simplifying communication between engineers. Design patterns may be components themselves but according to several definitions a component shall be executable and patterns cannot be executed. Rege [47] introduced four useful patterns for component-based development. These patterns are described below and can be used for designing with components.

The *Dynamic Factory* pattern provides a definition of an interface with operations for the creation of components. Components are bound at a late stage, i.e. the concrete implementation is determined at runtime. They can thus be created by different vendors and composed dynamically to give a specific machine.

The *Aggregation* pattern aggregates components into larger entities. Components can be loaded as separate entities and be configured dynamically. Other aggregates which extend the functionality can be associated with those already existing. Components are tied together into aggregates.

The *Embedding* pattern provides a definition of a uniform interface to one or more components. This makes it possible to adjust the interface and the components implementation to the specified requirements. A component may not be reusable because its interface or part of its implementation does not match the specified requirements of the application. The *embedding* pattern can then be used to wrap the component to fulfil the requirements. An example of how components can be wrapped to modify the requirements is given in [57].

The *Propagator* pattern defines a network of coupled components such that if the state of one component changes, the dependent components are notified and may be adjusted. In a system, there is a logical interdependence between the states of the components. If one component changes its state, it is necessary to notify all dependent components, since these may need to adjust their states accordingly.

Patterns can be used to understand and design a complex configuration of components. Understanding the architecture is crucial when implementing a system. Design patterns are related to architectural styles as presented by Bass et al [13] and can be used as metaphors communicating the architecture and design of component-based systems.

2.4 Interfaces

Components communicate through interfaces which the user accesses when interacting with the components. The actual interaction with the component is performed using connectors, i.e. Connectors mediate interactions between components [52]. If an interface is changed the user needs to know that it has changed and how to use its new version. When the interface is changed, the user frequently becomes aware of the change too late, when, for example, attempting to access or link with the component. The component providers must therefore announce the change before implementing it.

Functions exposed to the user are usually designated Application Programmable Interfaces (API). If there is a change in the API, the user must also recompile his code. This is the case with compiled languages such as C/C++ but not interpretative languages such as Smalltalk or Java.

In the object-oriented programming world, an interface is a set of public methods defined for an object. The object can usually be manipulated only through its interface. In C++ the user need only recompile the code when an interface, referred to from the code, is changed. There is also the additional drawback that the user of the class must use the same programming language throughout the whole development.

Separating the interface from the implementation is one way of avoiding this tight coupling. This kind of separation is performed with binary interfaces in CORBA [6] and COM [4]. Binary interfaces are defined in an interface definition language (IDL) [6,15] and an IDL compiler, which generates stubs and proxies, making the applications location-transparent.

By defining interfaces as unchangeable units, COM solves the interface versioning problem. Each time a new version of the interface is created, a new interface will be added instead of changing the older version. A basic COM rule is that an interface cannot be changed once it has been released. This makes couplings between COM components very loose and makes it easy to upgrade parts of the system independently.

Even if an interface has not been changed, its implementation can be changed. This increases the flexibility of possible updates, but also introduces the possibility of uncontrolled effects. For this reason, it is of interest to know if the implementation has been changed.

2.5 Commercial Off-The-Shelf Components

Buying commercial off-the-shelf (COTS) software is a common way to gain functionality without needing to develop everything from scratch. Components are sometimes wrongly referred to as COTS software. Certainly, components may be COTS but this does not mean that COTS must be components. A vendor sells COTS products as unmodified units which can be used in developments. For example, a class library linked to a source code is not a component but it is COTS software. Components, however, are often distributed as COTS software, which means that the issues with COTS apply to component-based development as well as to other developments. Development with COTS components has many advantages [58]:

- Functionality is instantly accessible to the developer.
- Components may be less costly than those developed in-house.
- The component vendor may be an expert in the particular area of the component functionality.

Despite all the advantages, there are several disadvantages [36]:

- Often, only a brief description of its functionality is provided with a COTS component.
- The component carries no guarantee of adequate testing.
- There is no, or only a limited description of the quality of the component.
- The developer does not have access to the source code of the component.

To make the decision to buy or to build is not easy, knowing all the disadvantages. COTS components are typically “black boxes” without their source code or other means of introspection available. Developers must identify certain properties of COTS components to integrate them properly with a system under development. Examples of relevant properties are functionality, limitations, correctness, preconditions robustness and performance. To determine its properties, extensive testing of the component is necessary. There are various approaches to this kind of testing, e.g. random, “black-box” and “white-box” test generators.

Thane [56] presents a model for determining the reliability of components. To acquire confidence in a component it must be supplied with a contract and be tested with a certain input. A contract specifies the functionality and the run-time conditions for which the component has been designed, i.e. assumptions about inputs, outputs and environment. If the component supplier provides such a contract, it can be used to calculate the probabilities of the occurrence of errors. Evidence based on the component’s contracts and the experience accumulated must be obtained. The environment must be considered when components are integrated in new systems; the input domain may differ considerably from the input domain for which it was tested. Confidence in a component’s reliability is only warranted when the component is used in the environment for which it is intended.

COTS components with same functionality can be categorized in groups even if their implementations are different. Components in the same category can be exchanged transparently if their interfaces are the same. If the interfaces differ, wrappers can be used to provide the same interface towards the user. When more than one vendor provides components with the same functionality, it is advantageous to design the system for component exchangeability. An architecture which supports the exchange of components is more stable, e.g. if the support for a component selected is discontinued a new component can replace the obsolete.

2.6 Component-based Development

This section describes the differences between developing components and developing with components. It is important to make this distinction to make it clear how to use different methods. The developer of a component must think about how to make the component open to integration with other components and less about how to integrate other components.

Recommendations to developers and users of components are given in this section. As a result of studying different aspects of component-based systems, we provide a list of recommendations for this area. The section is divided into two parts, one for the component developer and one for the component integrator (the user who develops systems incorporating components).

2.6.1 Developing Components

There are many known difficulties to be encountered when developing components. It is generally difficult to design and develop a component intended for unknown products. Developing internal components to fit into a product line is therefore easier [14]. Satisfying all the requirements demanded of a component by customers is not easy, as the requirements are often conflicting and impossible to fulfil. To determine the “right” requirement is an important issue for the component developer. To develop a generic component takes more time. Lampson [37] states that it takes around three times more effort. If the time required is too long, the time to market may be prolonged and the market window might be too small to make a profit for the component.

When developing and designing components, the following is recommended:

- Always document all the features of the component. Do not restrict the documentation to functionality, but document all other properties such as performance, resource consumption, limitations and robustness.
- Provide test-suites with the component so that the customer can test the component in their own environment. It is extremely important to test an imported component in the environment in which it is to operate. Remember the Ariane 5 rocket explosion [41] which was due to a change in the environment requirements and not in the software design.

- Provide source code to help the application developer understand the semantics of the component.
- Design the components so that they can be integrated into existing component models. Describe models in which the component works and describe how to make it work with other models.
- Carefully generalize the components to permit reuse in a variety of future contexts. Note, however, that solving a general rather than a specific problem requires more work.
- Make sure that the application developers can adapt the component to their requirements. This can be done with sink interfaces to which the user adds an interface to the component so that the component can utilize that interface to communicate with the user.

2.6.2 Developing with Components

Development with components is a complex process as there is always a trade-off between buying and developing the components concerned. It is generally better to buy general-purpose components, e.g. operating systems, databases and user interface components. Many different aspects must be considered before choosing an existing component over an internal. The development of proprietary components takes resources, requires maintenance and support. Before making decisions when building applications with components, the following questions and thoughts should be considered:

- The market for a product is limited in time and it is therefore important to deliver a.s.a.p. There is a risk that a component vendor may discontinue support, and if the support is discontinued, there may be a loss of time and a postponed release. The risk is considerably lower with a well-established vendor.
- The functionality provided by the component may not remain the same over time, forcing the integrator to create wrappers, which provide or prevent functionality, around the components. If the support from the component vendor is inadequate, this could be a serious issue.
- The functionality of the component may be more than actually needed, requiring restrictive wrappers to be written. In this case unwanted functionality is paid for unnecessarily. The use of unintended functionality may cause problems.
- If the source code is in fact available from the component vendor, is it really maintainable if something goes wrong?
- A malfunction in the component may cause an error in the product. The end user wants the product to function without needing to think about the internal design. The product vendor must solve problems arising even if the error is in the third-party component.
- If an external component is customized for a product, it makes the product strongly dependent on the component vendor. The vendor can then set his own price for continued support of the component.

There are many more issues surrounding CBSE to be addressed before making decisions on how to design a system with components. Takeshita [55] lists several questions concerning metrics and risks. The subtopics discussed are: metrics for components, difficulty in acquiring proper components, metrics for completed applications, insufficient analysis/design and architecture mismatching, insufficient technical support, use of low-quality components, cost-related risks and difficulty in managing computing in enterprises.

Josefsson [35] presents the following recommendations to the component integrator:

- Make a thorough evaluation of the component suppliers. Are they suitable as suppliers? Do they have good quality products and support? Check their financial position for economic stability.
- Ensure that the legal agreement with the supplier is comprehensive. This may save time and efforts if the supplier goes out of business or if they refuse support of their component.
- Create good and long term relations with the supplier for better cooperation.
- Limit the number of partners and suppliers. Too many will increase the costs and the dependencies.
- Buy "big" components where the profit is greatest. The management of too many small components can consume the profit.
- Adjust the development process to a component-based process.
- Have key persons assigned to supervise the component market, monitoring new components and trends.
- Try to gain access to the source code. Through special agreements with the vendors etc.
- Test the components in the target environment.

These recommendations do not provide a complete solution to all the problems which may occur, but they indicate that developing for and with components must be performed carefully.

2.7 Development Cycle

The development cycle of a component-based system is different from those of the traditional models, such as the waterfall, iterative, spiral and prototype models.

Development with components differs from traditional development. There is, for example, a new component development process for CBSE [16] which differs from the traditional waterfall model. A similar process for development of COTS components which emphasizes requirements, design, coding and integration is described by Morisio et al [42]. Figure 6 shows a comparison between two different development processes. Determining requirements and designing in the waterfall process correspond

with the finding and selection of components. Implementation, test and release correspond to create, adapt, deploy and replace.

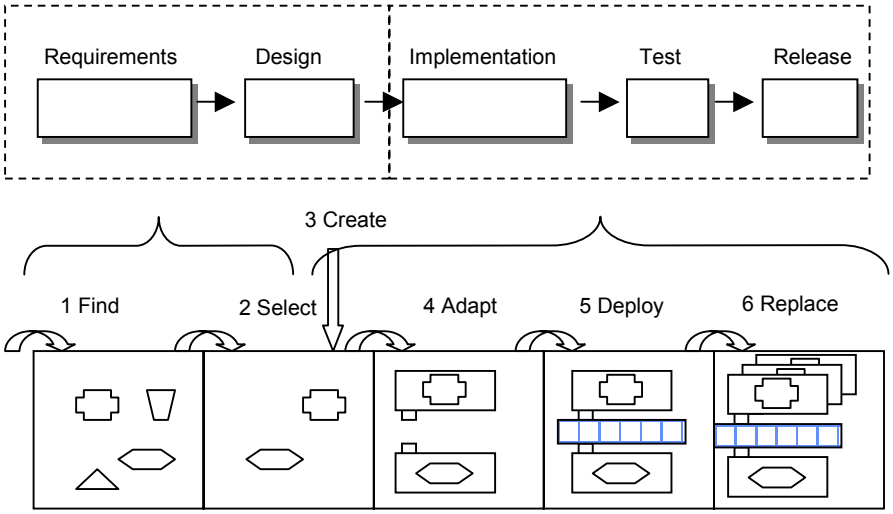


Figure 6. An example of a development cycle with components compared with the waterfall model.

The different steps in the development with components process are:

1. Identifying components which could be used in the product. All possible components are listed here for further investigation.
2. Selecting the components compatible with the requirements of the target product.
3. Creating proprietary components to be used in the product. These components need not be found since they are developed inside the enterprise.
4. Adapting the selected components to suit the existing component model or requirement specification. Some components need more wrapping than others.
5. Composing or deploying the product. This is done with a framework or infrastructure for components.
6. Replacing old versions of components with new, i.e. maintaining the product. There may be bugs to be eliminated or new functionality to be added.

In Figure 6, the find phase appears to replace the determination of requirements. The figure should be interpreted however, as showing that finding and requirement determination are performed in the same phase of development. There is a need for requirement determination by means of analysis, design and testing when performing component-based development.

Different technologies are available to support developers at each step. There are tools such as Agora [51] and Jcentral [8] to find components while Cool: Spex [1] from Sterling software can help with the selection process. There is also COMCAD [46], which supports the deployment of COM components.

3 Configuration Management

Configuration management (CM) is the control of the development and evolution of complex systems [19,27]. A complex system is characterized by a large number of components developed by many persons with usually rigid time constraints and quality requirements. These kinds of systems are often developed concurrently and are intended to last for longer periods of time, between ten and twenty years. Version control is usually used in all types of projects. When it comes to more complex and sophisticated projects, a systematic use of configuration management methods is needed. Change, build, release and workspace management are other disciplines covered by CM. This section gives a brief introduction to all of these disciplines.

3.1 Version Management

An element of software or hardware placed under version control is designated a *configuration item*. The most common example of a configuration item is a source code file but executables and documents might also be considered to be configuration items. Version control covers the management of different versions of items, usually represented by a tree structure [12]. Version management supports concurrent development in which the concurrent versions are usually implemented as branches. Selected versions of configuration items together form a *baseline*. Tags are used as marks or labels to identify a specific version of an item. It is possible to tag all the different configuration items included in one release of a system, for later retrieval of the very same configuration. This may be needed e.g. when bugs must be eliminated. The same version of a configuration item can be included in many baselines by attaching multiple tags to it.

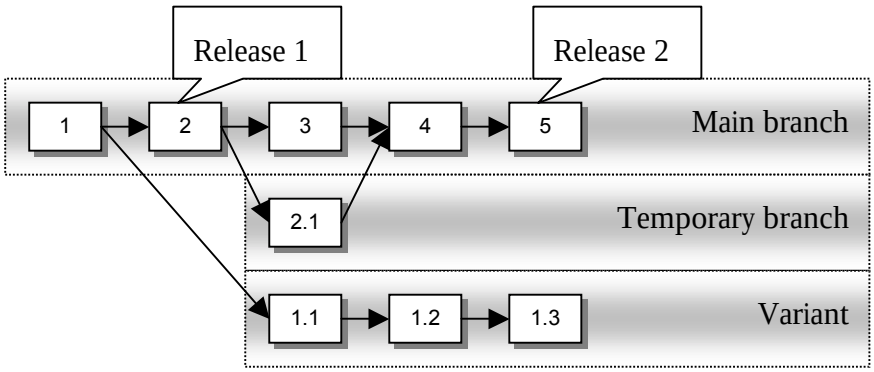


Figure 7. Variants are branches which do not merge back into the main stream.

Items are checked out and in from the version database when needed. There are different models for concurrent work but the most common is to use locks on the configuration items. When a version of a file is to be modified, the locking mechanism which prevents concurrent work is used. After the file is locked the developer can be

sure that no other person is modifying the same file. When the developer is ready, the file is checked in to the version repository and the lock can be is released. If a developer intends to develop a locked file it is possible to create a temporary branch to permit this. Another model is to permit concurrent development with an optimistic approach. In this model the files are not locked but the file is checked when returned to the database. If the file has been changed in the meantime, a merge will take place with the developer responsible.

Branches are used to permit work in parallel or to create multiple variants of a release as shown in Figure 7. A branch can span over many different versions and is thereafter usually merged back into the main branch. An example of this is when a developer has created a branch 2.1 because the original version 2 was locked by another developer. After both developers have finished their assignments, branch 2.1 can be merged with the main branch as version 4. Version 4 in the main branch will now contain the changes made in both versions 3 and 2.1. Another example of the use of a branch is when there is a need for extra functionality to be added to subsequent releases but not to the current release. To accomplish this, the new functionality is inserted in the branch until the subsequent release and the functionality is then merged. If the functionality is not merged back into the main branch, the branch is designated a variant.

With version management, it is possible to control configuration items included in a release of a complete system and it is also possible to recreate the very same configuration on demand with the help of baselines.

3.2 Change Management

The reasons for changes are multiple and complex. Changes can originate from many different sources. *Change management* handles all changes in a system. The reason for a change can be an error, improvement of the component or added functionality. Change management includes tools and processes which support the organization and track the changes from the origin to the actual source code [22]. Examples of Change Management tools are PVCS tracker [9], Visual Intercept [10] and Clear Quest [3].

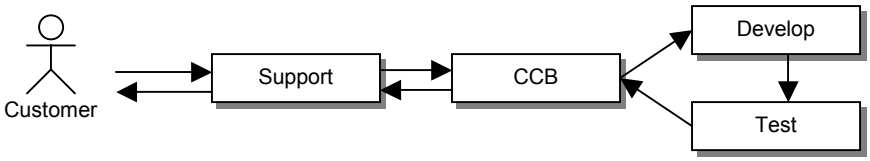


Figure 8. An example of a change process in which the customer files requests to the support organization.

When a change is initiated, change requests are created to track the change until it is resolved and closed. Figure 8 shows how a customer requests a change to correct an error in the system. The support organisation receives the change request, taking direct action and solving the problem if possible. If they cannot address the issue, the request is passed to the next instance. The configuration control board (CCB) analyses the

change request and decides which action is to be taken. If the change is approved, the change request is filed to the developer responsible for implementing the change. When the developer has performed the change its status becomes “implemented” and a test is performed. When the subsequent new release is to be built, the CCB decides which changes are to be included. The customer receives a patch with the new release including documentation of the changes made in the new release.

Various tools are used to collect data during the process of tracking a change request. Change management data can be used to provide valuable metrics about the progress of project execution [24]. From this data it can be seen how many changes have been introduced between two releases. It is also possible to check the response time between the initiation of the change request and its implementation and acceptance.

3.3 Build Management

Build management supports the user by collecting source code for a particular release and then using build tools, such as Make to create configurations. Make describes the dependencies between source code files at build-time and ensures that the dependent source code is built in the correct order. Ongoing research for automatic assembling of components, shows that it is very difficult to apply build management to component-based systems. Zeller [61] proposes the use of description logic to assemble components in design time. This does not solve however, the problem with dynamic configurations in run-time.

Daily builds can be performed when build management is supported. If the system is built every day the integration time is reduced, since broken dependencies and faults are discovered early. The daily build process also permits rapid development and early testing of the system. Support for parallel development with version control which resolves inconsistencies is required for daily build [44].

3.4 Release Management

The identification and organisation of all documents and supplements incorporated in a release is designated *release management*. It is possible with appropriate release management to create installation kits automatically to ease the task of the build manager. The build manager is responsible for providing the finished product with the correct configuration and features. Products such as Windows installer and Install shield [11] can be used to create installation kits. Hoek et al [33] describe a prototype, designated Software Release Manager (SRM), which supports both developers and users in the software release management process. SRM incorporates the concept of components and helps in assembling them into systems. Dependencies are explicitly recorded so that users can investigate and understand them.

3.5 Workspace Management

Introducing CM in an organization is cumbersome without effective support from tools. Changing an existing culture requires massive education, support and not least motivation. To motivate developers to use all the tools and methods available with CM, support for integrated tools in the development environment is needed. Developers want to work independently of the configuration management, this alternative being denoted workspace management. Developers usually focus on solving particular problems and have less interest in administrative tasks. An example of integrated features is when the developer "logs-in" to a project environment in which project structures and data repositories are already prepared for the developer. The developer then enters a transparent environment in which the development with configuration management is handled behind the scenes. This approach is adopted in such major configuration management tools available on the market today as Clear Case and Continuus [2,5].

3.6 Related Work

Distributed and parallel development is related to component-based development since the sub-systems delivered are often treated as separate entities. In distributed development, a distributed database for all configuration items is needed. This model does not suit the component-based approach since most components are developed independently using a variety of tool sets. External components can be treated as outsourced parts of a project with more restrictions on source code. A model for outsourced software development is presented in [12]. The outsource model gives more control to the integrator since he is a direct customer to the component provider and can require desired functionality for a specific target environment.

When a CM process is to be defined, different issues for users of the CM system must be considered [25] and the user's roles must be defined. The CM system must have integration capability with existing development tools for transparent workplace management. To obtain support for build and release tasks, the degree of automation must be defined and the desired level of automation achievable by using the existing tools must be known. These issues must be considered when a configuration management process for component-based development is defined.

4 Dynamic Configurations

Dynamic configurations are needed to permit architectures to evolve during the life-time of a system. This occurs in practice when components create instances of other components during the system execution. Architectural Description Languages (ADL) can be used to express such architectures [52]. A system with a configuration of components in which the components may be replaced with new components is designated a subject of change, i.e. its contents change during its lifetime. Figure 9 shows an example in which components are replaced. Such a configuration is dynamic, since its components can be updated after the configuration is first deployed. A static configuration has the same set of components over its lifetime and nothing changes after the system is launched. For example, a simple program in which all its elements are statically linked together into one monolithic program.

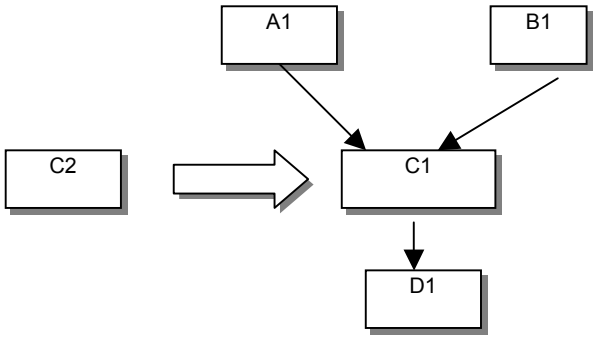


Figure 9. Component C1 is replaced dynamically with component C2.

One example of a dynamic configuration is a PC running with executables and shared libraries which might be upgraded during runtime, another is a control system in which the control algorithm component is detached from the memory and a new component is inserted. Configuration management typically addresses this type of problem with identification, versions and dependencies but has not been applied to the particular issue of dynamic configurations. These issues have been explored in [38,39].

A configuration consists of a number of components which interact to perform the task of the system. Traditional CM can very well be applied to the development of the component but it cannot handle systems in which the configuration might change during run-time. Examples of systems in which the configuration might change are: web applications, systems built on component models with dynamic linking and real-time systems in which tasks might be changed on the fly. Applications are also written today which upgrade themselves when needed, Real player for example which checks with the home server for newer versions each time it is invoked. The management of systems in which components or applications upgrade themselves without notification is a challenge. Imagine components using an other component which suddenly upgrades itself. How is deterministic behaviour possible in such a system?

Another example of a system which configures itself dynamically in different environments could be a personal device connected permanently to the internet. When the device is used on the owner's premises, high-speed communication without security might be acceptable. But when the device is used outside the company building, a more secure, slower connection must be used. This kind of systems must be reconfigured dynamically. How can the function be guaranteed if it is not know before what components are to work together?

Run-time reconfiguration is possible by altering interface bindings [45]. It is possible to change to a new component since the interfaces of a component arbitrate all communication to and from the component through proxies. If the components are state-less, the component can be removed and replaced with another which provides an equal or greater interface. In the case of a component with state, the state must be stored by the old component and reloaded to the new component. This requirement means that new versions of a component must be able to load and interpret the storage format from all older versions. The persistent data must be tagged with protocol version to permit the use of multi-versions of data.

Related to dynamically configurable component-based systems are web systems, which have highly dynamic contents and are easy to upgrade by changing the contents behind the dynamic links. Susan Dart [26] has identified the problems with dynamic configurations for web systems and she claims that all CM knowledge can be applied to solve these problems. However, the challenges span technical, process and company-political issues and are not easy to address. Dynamic content, variant explosion, performance, scalability and corporate politics are examples of such challenges.

Traditional graph theory, as presented in [31,53], can be used to monitor the dependencies between components. It is also useful to represent a graph with a matrix or adjacency lists to be able to determine if components are dependent or not.

5 Component Configuration Management

As shown in [20,38,39] it is difficult to manage components during the life-time of a system. A system of components is usually configured once only during the build-time when known and tested versions of components are used. Later, when the system evolves with new versions of components, the system itself has no mechanism to detect if new components have been installed. There might be a check that the version of replacement component is at least the same as or newer than the original version. This approach prevents the system from using old components, but it does not guarantee its functionality when new components are installed. To apply ideas from configuration management, such as version and change management in managing components is an approach which can be used to solve some of the problems.

This section describes the application of configuration management to systems built with components, and presents a model for checking dependencies between components. Some level of configuration control will be achieved if it is possible to identify components with their version and dependencies to other components. Information about a system can be placed under version control for later retrieval. This makes it possible to compare different baselines of a system configuration. To manage dependencies, a graphic representation of the configuration is introduced. The graphs are then placed under version control. This information can be used to predict which components will be affected by a replacement or installation of a new component. Workspace management can be achieved with an environment which provides the user with information about the current configuration.

Change management of components is the same as that of ordinary software. Traditional change management can also be applied to third party components to track of detected errors until the component provider resolves them.

Using component configuration management, it is possible to answer questions, such as:

- Which components have been added/removed after a reconfiguration?
- Which dependencies have been added, removed or affected by a reconfiguration?
- If a component is updated, which other components in the system are affected?
- What is the effect on a system if a new system of component is installed?
- What is the difference between two configurations?

These are some of the difficult questions which, when answered correctly give a better understanding of a system and permit a certain level of predictability when upgrading systems.

It is generally difficult to identify components during run-time and to obtain their version information. A short outline of this issue is given in the next section. When the components are identified it is possible to build graphs of dependencies, which can be

represented in various ways and placed under configuration control. A prototype developed for detecting components on the Microsoft platform is described in section 5.8.

5.1 Component Identification

There is a current lack of information for us in identifying components in systems. No information about version, change history or creation is available. There is no standard interface which can be used to gather sufficient information about the component to permit the creation of a dependency graph. Such a dependency graph is necessary to predict the effects of updating the system with new components.

To identify a component, name, creation time, size and a magic number (a unique number set by the compiler) are used. If a version identifier is provided for a particular component, this is also used. The identification data is used to calculate a unique key to be used to compare components. The key is divided into two parts, one for identification, the other for version.

If no version information can be obtained from the component itself, it can be added manually using the *embedding* pattern described in section 2.3 to wrap components with a version information interface.

Hoek [32] presents ideas about how product line architecture components can be identified. The properties he defines are: name, revision, interface, connection, behaviour, constraints, representation and origins. These are properties to be considered and placed under a version interface as proposed in [38]. A version interface can be used to build the dependency graph presented in Definition 5.

5.2 Configuration Model

A configuration model defines how components are treated and put under version control. The set of components installed in a system is called a system of components as defined in [39].

Definition 1. System of components.
A *system of components S* is a set of installed components in a system,
$S = \{c \mid c \in$ Components installed in the system$\}$

A system of components *S* can also be treated as a baseline and be placed under version control. The components themselves are not under version management but the unique key, which identifies them, is used as a representative. This means that *S* is a set of keys. For improved performance *S* is to be sorted to permit the easy location of components. This is also a prerequisite for comparing configurations by means of deductions from the dependency graph.

Grouping many different components providing the same functionality extends the model. This makes it possible for example, to change components if one component is no longer available.

5.3 Change Management

Change management can be applied to both internal and external components. In the case of internal components, it is possible to use the same tools for change management as for the development of the component itself. External components can be placed under change management to permit the monitoring of changes and bugs which occur. Instead of attaching source code files to change requests, the name of the component can be used to track changes. When, for example, the problem report is analysed, the outcome can be a change request for each component involved. Each such change request contains a list of all the changed source files or a description of the patches if the component is external. Patches from the component vendor must be stored to permit recreation of the same configuration later. The change requests serve as a container for information of that kind.

5.4 Managing Dependencies

Dependencies between components are to be tracked and stored for further management. The benefits of this are multiple. It is possible to analyse what has been affected in the system and to create determinism when updating the system with new components. Dependencies between components can be represented with a directed graph, as defined in [31]. The definition is shown below.

Definition 2. Directed Graph.
Let V be a finite nonempty set, and let $E \subseteq V \times V$. The pair (V, E) is then called a directed graph, in which V is the set of *vertices*, or *nodes* and E is a set of *directed edges* or *arcs* represented by ordered pairs.
Such a *directed graph* is denoted $G = (V, E)$. The notation $a \rightarrow b$ denotes $(a, b) \in E$

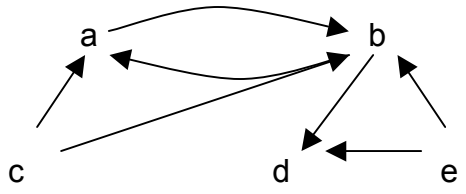


Figure 10. An example of a graph G with the nodes a, b, c, d and e.

Figure 10 shows an example of a graph G = (V, E), in which V={*a*, *b*, *c*, *d*, *e*} and E={(*a*, *b*), (*b*, *a*), (*b*, *d*), (*c*, *a*), (*c*, *b*), (*e*, *b*), (*e*, *d*)}. Placing an arrow on the edge indicates the direction of the edge.

Definition 3. Path.
Given a graph $G = (V, E)$ a non-empty sequence of vertices $<v_1, v_2, \ldots, v_n>$ in which $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$ is called a *path* from v_1 to v_n in G.

Paths are introduced to be able to define dependencies between components. An example of a path from a to d in Figure 10 is $<a, b, d>$ since each pair (a, b) and (b, d) is a part of the set of edges E. Knowing that there is a path from *a* to *d* indicates that *a* is dependent on *d*, since *a* is affected if *d* changes. Definition 4 gives a formal definition of transitive closure.

Definition 4. Transitive closure. The *transitive closure* of a graph $G = (V, E)$ is a new graph $G^* = (V, E^*)$ where $E^* = \{(a, b) \mid$ there is a path from *a* to *b* in $G\}$.
The notation $a \rightarrow^* b$ will be used to denote $(a, b) \in E^*$. □

In Figure 10 there are examples of dependencies such as a $\rightarrow$ b, c $\rightarrow^*$ d and e $\rightarrow^*$ a.

There are different ways to represent a directed graph [53]; examples are matrices, lists and nodes with pointers to their children and parents. The transitive closure can be calculated to gather all the possible edges of a graph, see Figure 11. It is possible to gather all the dependencies for each of the different representations, but some representations are more efficient than others.

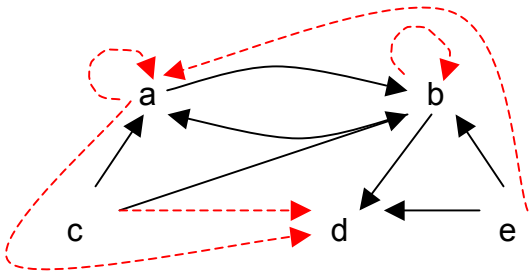


Figure 11. The graph in Figure 10 with the transitive closure calculated. The dashed lines are edges which have been inserted in the graph.

Different algorithms can be used to calculate the transitive closure to obtain all paths in a graph. Examples are three algorithms from Warshall, Nuutila, and Eve and Kurki-suonio [28,43,59]. Warshall's is a simple well-known algorithm which works on matrices. It is described in the next section.

5.5 Dependencies Between Components

Components are the nodes in a graph and the dependencies the edges. Now that dependencies are defined and described it is possible to define the set of all dependencies as a set of dependency pairs.

Definition 5. Dependencies.
Let S be a non-empty set of installed components in a system. Then the *dependencies* D is defined as $D = \{(c_i, c_j) \mid c_i, c_j \in S \wedge c_i \rightarrow c_j\}$

When the dependencies have been calculated, it is possible to create a system structure, as defined in [21], with different levels of components. On the lowest level of components are components without dependencies to other component. This system structure is used as a model to calculate quality properties such as complexity and localization factors. The complexity is proportional to the number of dependencies between the components. The localization factor denotes the number of levels between components. There are, for example, three levels in the path <*c*, *b*, *d*> from component *c* to *d* in Figure 11.

A configuration is a set of components and their dependencies to other components. The configuration is a baseline since it represents a version of a system at a particular time. Configurations are defined in Definition 6.

Definition 6. Configuration.
Let S be a non-empty set of installed components in a system. Let D be the dependencies for the components in S. Then a *configuration* C is defined as a graph $C = (S, D)$.

Configurations are stored under version control for later retrieval. New installed components can be compared with a configuration to permit recognition of the affected components in the system. When new components are installed, new nodes in the dependency graph are added. In the same way, nodes are removed if components are removed.

Broken dependencies are detected when the old configuration is compared with the new. New versions of an existing component are identified by the version part of the unique key which identifies all components. New versions simply replace the older in the graph. When comparing graphs, new versions are detected since the keys will be different. Proper dependency analysis requires that a component and its version can be identified.

It is important to have the possibility of marking selected components as critical to indicate that they must not be affected by a component update. A critical component must not be upgraded or affected by an upgrade of another component, whether critical or not. Critical paths are therefore introduced. The critical components are a subset of the components.

Definition 7. Critical path.
A *critical path* is a *path* from a critical component to any other component. The set of all critical paths is a subset of all the paths in a configuration.

No component in a critical path may be upgraded without making an active decision to accept the change. Hence more than only the critical components are placed under supervision. On the other hand, components which depend on critical components need no special treatment and can be upgraded without notice. The algorithms presented in this thesis make it possible to keep track of the components which have been affected after the upgrade has taken place.

5.5.1 Matrix Representation

Configurations and dependencies can be stored in a matrix format. This format is appealing since it is easy to understand and existing mathematical methods can be applied to it. Each component is represented by a column and a row in the matrix. If a component v_i is dependent on another component v_j a 1 is set at the position where the two components meet in the matrix.

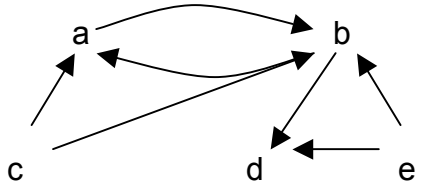
Or more formally the position in the matrix $D_{ij} = \begin{cases} 1 \text{ if } v_i \rightarrow v_j \\ 0 \text{ otherwise} \end{cases}$

The dependencies can be represented in this matrix and the transitive closure can be easily calculated to gather all indirect dependencies in the system. If the graph is represented by a matrix, the transitive closure can be calculated, using Warshall's algorithm as described in Figure 12. The algorithm takes a given boolean matrix $m[i, j]$ of size $n \times n$ representing the graph and calculates the transitive closure.

```
For 1 ≤ i ≤ n do
  for 1 ≤ r ≤ n do
        if m[r,i] then for 1 ≤ k ≤ n do
              m[r,k] := m[r,k] or m[i,k]
```

Figure 12. Pseudo code for Warshall's algorithm to calculate the transitive closure.

If position $m[i, j]$ in the matrix is *true* then there is an edge from i to j. After the calculation each position denotes that there is a path from a vertex to another vertex.



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad T(A) = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 13. A graph represented with an adjacency matrix A and the transitive closure T(A).

Figure 13 shows how the matrix representation might appear when Warshall's algorithm has been applied to calculate all the paths in the graph. Figure 11 shows all the calculated paths as dashed arrows. This matrix representation is quite memory-consuming since a configuration with N components will use N*N bits of memory assuming that one bit is used for each edge.

As a configuration is defined as the system of both components *S* and the dependencies *D*, the rows of dependency matrix are to be sorted in the same way as *S* is sorted. They are sorted with a key which uniquely identifies a component and its version.

Since the number of relations is usually sparse, more efficient ways of representing graphs, such as list representation can be used.

5.5.2 List Representation

Another way to represent a graph is with linked lists. Each component has its list of adjacent components. This representation reduces the resource consumption, if there are sparse relations between components, and can be used for calculating the transitive closure.

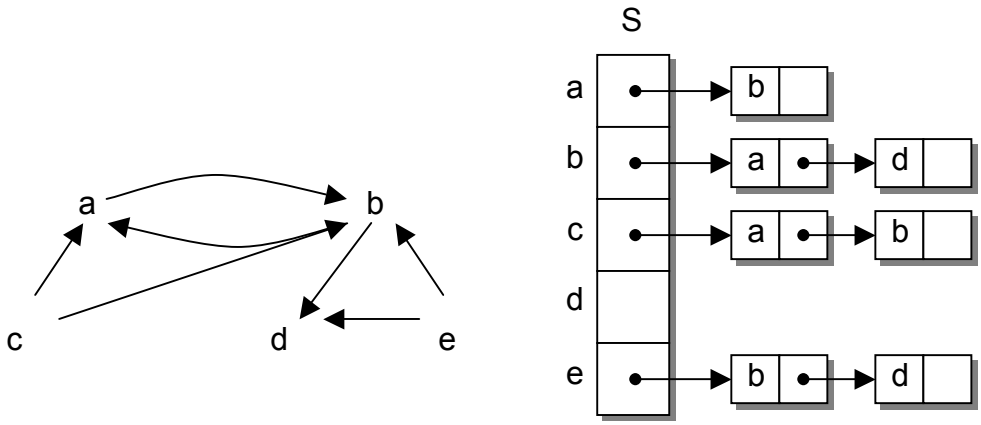


Figure 14. A graph represented with linked adjacency lists.

In Figure 14, the system of components *S* is represented with a list of lists. Each list represents all components on which a component in *S* is dependent. Logically, the lists are sets but they are implemented as lists, i.e. each component has a set of components on which it is dependent. To better reflect this representation, Definition 5 can be revised to define the dependencies to $D = \{(c, \{c_i \mid c \rightarrow c_i\}) \mid c \in S\}$. The length of a dependency list can be used as a measurement of dependency complexity. The higher the number, the more complex is the relation with other components. Using this representation reduces the memory consumption to one list node per component with an additional list node per edge in the graph. This representation can also be extended to include an additional list of in-edges instead of only out-edges. Using this extension is more resource consuming but permits faster searches for all components dependent on a component.

5.6 Differences between Configurations

This section describes how differences between configurations can be computed and used to obtain a deeper understanding of what has been changed in the system.

The dependency graph, which is represented by a matrix, can easily be used to determine what has been affected when a new version of a component has been installed. If the matrix is transposed, all the directed edges in the graph are reversed. It is possible to obtain version history from different configurations by simply comparing different versions of the dependency graph.

To compare two configurations, $C_1 = \{S_1, D_1\}$ and $C_2 = \{S_2, D_2\}$ where the second is more recent, there are two steps to be taken to determine the changed components and edges. Firstly, to determine which components have been added, use set difference to subtract S_1 from S_2, i.e. Added components $S_{new} = S_2 - S_1$. Using the same method, the removed components are $S_{Removed} = S_1 - S_2$. Secondly, D_1 and D_2 must be compared to calculate which edges have been removed or inserted. To determine what is affected when a new version of the same component is installed, take $S_2 - S_1$ to obtain the added components. Since new versions have different keys, in the sense of version, the difference will indicate the new versions installed. The dependencies D_1 can now be used to analyse what has been affected.

If new components have been added or old components removed, the matrices are of different logical size (as shown in Figure 15). In this case the matrices must be extended to the same size. Take the union of the two sets S_1 and S_2 to get all components in the two configurations. Create two new matrices which describe the dependencies of all components including removed and added components. These unchanged components must be inserted without dependencies, i.e. add a new row and column with zeros for each component remaining unchanged. The two new matrices S_{1p} and S_{2p} will now be of the same size and have the same order of rows and columns. It is now an easy task to make the subtraction to determine which edges have been changed between the two configurations. If the value is negative it shows that a path has been removed and if it is positive, a path has been added.

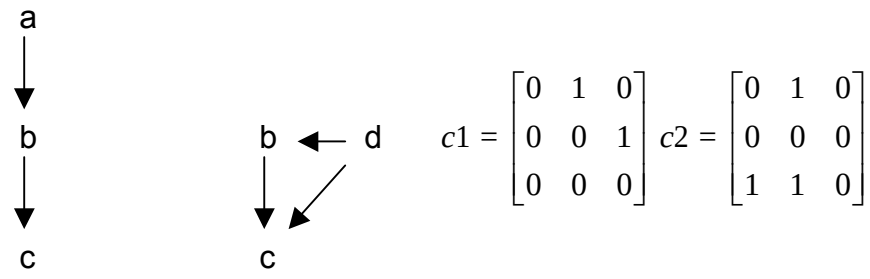


Figure 15. *c*1 and *c*2 represents two different dependency graphs with different components.

The configurations in Figure 15 are used in the following example. In c1 the rows represent *a*, *b* and *c* but in *c*2 they represent *b*, *c* and *d*. To be able to compare these configurations *c*1 and *c*2 must be modified to have the same size and order. Two new matrices are created in which the changed components are inserted without dependencies to any other component. A row for *c1* and a column for *d* are added. In the case of *c2* a row and column for component *a* are added.

To add empty columns into a matrix, multiply with the modified unity matrices E_1 and E_2 where zeroed rows and columns have been added. If the new size is to be $m \times m$ and the original matrix is of size $n \times n$ then E_1 is $m \times n$ and E_2 is $n \times m$. Take the configurations in Figure 15 and increase the size to a 4×4 matrix designated c1´.

$$c1' = (E_1 c1)E_2 = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \right) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$c1' = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad c2' = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad c2' - c1' = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

It is now possible to subtract the two matrices to calculate which edges have been inserted or removed in the system between different configurations. If components have been removed, the result will be negative and if components have been added, the result will be positive. In the example in Figure 15 the edges from d to b and d to c have been inserted while the edge from a to b has been removed.

If the dependencies are represented by lists, it is easier to calculate the difference between configurations. There is no need to expand the lists to the same size since a set subtraction will automatically exclude the components removed and only focus on the new. The components which have been added or removed can be calculated in the same way as for matrix representation. For each component in S_2 take the corresponding set of dependent component in D_2 and subtract the corresponding set of dependent components from D_1. This gives the added edges for each component. The removed edges are calculated in the same way. A short pseudo C++ code with STL (Standard Template Library) presents the algorithm in Figure 16.

```
typedef map<Component, set<Component>> Dependencies;
typedef set<Component> System;
typedef pair<System, set<Dependencies>> Configuration;
Dependencies      dep1, dep2;
System            system1, system2;
Configuration     config1, config2;
for each comp in system2 do
  addedEdges[comp] = dep2[comp] – dep1[comp];
for each comp in system1 do
  removedEdges[comp] = dep1[comp] – dep2[comp];
```

Figure 16. Pseudo C++ algorithm which calculates added respectively removed edges.

The example in Figure 15 is used as an example. The configurations can be represented as C_1 = ({a, b, c}, {(a, {b}), (b, {c}), (c, {})} and C_2 = ({b, c, d}, {(b, {c}), (c, {}), (d, {b, c})}. The component a is removed and d is added. This is easily calculated by subtracting S_1 from S_2 and vice versa. Applying the algorithm in Figure 16 gives that the edges added are (d, {b, c}) and the edges removed are (a, {b}). The components without edges represented with the empty set have been removed in this example.

5.7 Managing Multiple Versions of a Component

The presence of multiple versions of a component installed in a system at the same time is a common situation to be handled. Three different approaches to this are presented below.

Cook [20] presents a model for configuring components with respect to the problem domain. Each component has an explicitly described input domain and a version identifier. The execution environment then determines which version to use at run-time. If a newer component with the same input domain exists, this is used. With this model, many versions of a component are available at the same time in the system until older versions are removed from the system. Since the components to run are selected from the input domain, newer versions will replace the old without further action. The disadvantage of this model is that the arbiter must be integrated in the environment to be able to schedule the correct component for a given input. It is not possible to apply this technology in existing component models such as COM or CORBA implementations.

Isovic et al [34] present a way to upgrade a reliable system with components. When a new component is installed, the previous version is retained as a backup if the new version provides unacceptable output. If the output is within a valid range, it is assumed that it is correct, but if the output is not within a valid range, the previous version is requested to perform the task instead. A supervisor is used to monitor the outputs from each component before they are passed on to another component. This approach can be used in combination with dependency analysis as a backup if critical components are affected by an update.

COM [48] handles multiple versions in an awkward way. It does not accept new versions of components and states that all new versions must be treated as new components with unique identifiers. This prevents new versions from interrupting the configuration in an unpredictable way. However, this is not always reliable since the component vendors do not always adhere to recommendations. They can provide new versions with the same identifier without being required to create a new component. This is often the case if a component has been maintained and the interfaces are the same. The next generation of windows services (NGWS) [60], is the new runtime environment which supports C#, handles multiple components by introducing unique identifiers for each component. Private and shared components are two means of classifying components in NGWS. If a component is private, it is not possible for other components to use it and this avoids the versioning problem since no other component can use it and the number of dependencies is one. If a component is declared as shared, many different applications can use it but with the risk of breaking dependencies when updating the system.

5.8 Dependency Browser

A tool, designated the dependency browser has been developed for the evaluation of the presented configuration model. The main requirement for the prototype was to be able to parse a Windows 2000 system for its components and dependencies. An iterative development model was used to be able to show results more quickly with a working prototype. The prototype is able to browse the dependencies in a system and to store them under version control. The prototype incorporates ideas from [38,39]. It is also used to gather information about changes made between two configurations. Certain measurements such as complexity analysis are also provided.

There are different levels of dependency between components in a system; in a Windows system there are dependencies between shared libraries, as well as between static and dynamic COM components. Applications such as Word, Excel or Explorer, are treated as executables with their dependencies obtainable from the executable file itself. Since all Windows executable files comply with the portable executable format it is fairly easy to track the shared libraries but not so easy in the case of COM components. Scanning all shared libraries and executables in a system creates a basic dependency graph. Various features of the tool then extend this graph. The windows registry has been used to gather information about each component, which is then added to the dependency graph. Step by step, a configuration graph is built up for use in configuration management. Processes can be supervised and when new components are dynamically loaded into the memory, the graph is extended with dynamic dependencies. However, the creation of a complete dependency graph at the Windows platform has been a tedious task as there are too many dynamic dependencies difficult to detect because they have not been activated during periods of time when the system is supervised.

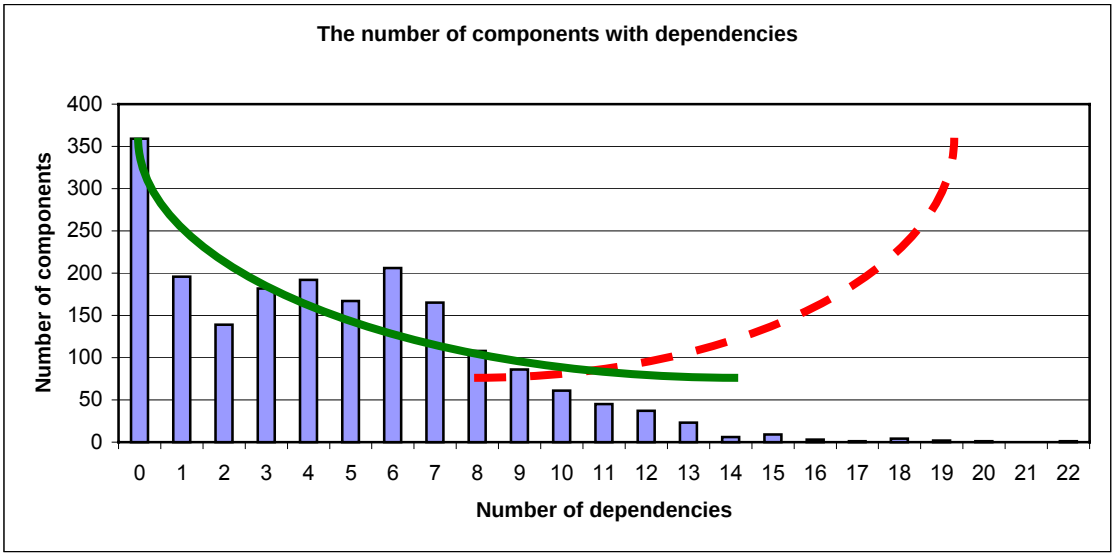


Figure 17. Analysis of number of dependencies from each component on Windows 2000. A system with a decreasing trend, shown with a solid line, is less complex than a system having an increasing trend, shown with a dashed line.

Experiments have shown 1993 components and 8936 edges in a Windows 2000 workstation configured for software development. It took 2 minutes and 4 second to compute all the dependencies on an Intel Celeron 300 Mhz cpu. The number of components and edges differ slightly between systems as a result of small differences in the installations. Not all the workstations in the department were configured in the same way since all the developers have their own tools installed.

Figure 17 shows the number of library dependencies for each component in the Windows 2000 system. In the graph it is shown that most of the components have less then five dependencies, this figure shows that such a system is less complex than a graph where the majority of the components have more than five components. The general complexity can be derived from the graph since a low number of dependencies results in a less complex system. The number of components with zero dependencies can be treated as basic components with low complexity and the component that depends on 22 other components has a high complexity. In this example, only direct dependencies have been measured before the transitive closure has been calculated. If the trend is decreasing as show with the solid line in Figure 17 it is a measurement that the system is less complex compared with a system having an increasing trend of dependencies between components.

A graph similar to that shown in Figure 17 can be created to express the reverse dependencies. Such a graph shows the components on which most other components depend. If there are few components on which many other component depend, these few components should be changed with discretion. On the other hand, all components

without dependents can be exchanged without risking the function of the system. All the dependencies must be reversed before performing this kind of analysis. A measurement such as this describes how many dependencies there are to a particular component.

Preliminary results show that it is difficult to identify all the components and their dependencies on the Windows 2000 platform. The configuration theory can be applied when the dependencies are discovered. More effort is needed to gather dynamic dependencies.

6 Future Work

The most obvious future work is to enhance the dependency browser prototype. Configurations which express dependencies can be used to predict which components will be affected by an update. The prototype developed does not implement this functionality, but it can certainly be implemented by saving a configuration for later comparison. The introduction of critical components is also a subject of improvement. Updating a component in a critical path must not take place before it is approved by the configuration control board. Additional future work is to investigate how introspection into Java beans can be used to determinate dependencies between components. The methods presented here will be explored more in detail and the results will be presented in a future article.

Preliminary results show that it is possible to obtain different metrics from the dependency graph, more research into the production and use of these metrics are planned for the future.

The performance of a case study of a component-based information organiser is planned. The method presented in this thesis will be used to analyse the dependencies. The information organiser is an assembly of standard components with very little added proprietary code, and an appropriate subject for a case study.

The development process used at ABB will also be the subject of an investigation. An analysis of this process is required, applying CBSE methods where appropriate. The results from this investigation will also be presented in an article.

6.1 A Proposed CM Process with Components

A process for managing components is to be developed. Preliminary ideas are the use of traditional CM for developing components, but incorporating a version interface as proposed in [38]. This interface will be used to gather dependencies and version information from the component when it is deployed. It is of great importance that the components can be identified when the system is assembled; if it is not possible to identify a component, it is not possible to predict what that component will affect during updates.

When external components are used, a dependency tool will identify the component dependencies and provide version information for each component. It is also possible to use the *embedding* pattern (se section 2.3) to wrap components to make them provide version information.

After a system has been assembled the configuration graph will be stored separately under version control. A baseline can be created to manage the components of a particular configuration.

7 Conclusion

Building software systems with reusable components has many advantages. The development becomes more efficient, the reliability of the products is enhanced, and the maintenance requirement is significantly reduced. However, it is shown in this thesis that there are still many problems with component-based systems. It is, for example, difficult to analyse the dependencies in a system.

A model for analysing and managing dependencies is presented in this thesis. By applying configuration management techniques to component-based systems together with dependency analysis it is possible to predict the effects of a component update.

It was shown that many basic principles from CM, such as identification, version management and change management could be applied in CBSE. The implementation of these principles is however not easy because there is no precise definition of a component, but rather many definitions which only describe the qualities a component should exhibit.

This research has been accomplished by performing a survey of the use of different technologies in various industrial projects. During the survey many problems with dynamic upgrades have been identified and served as a source for the research of analysing dependencies. The theory presented has been implemented in a dependency browser prototype.

The main challenge is to find all the dependencies when implementing the configuration model on a concrete platform. Here Windows 2000 was used as a target and it has proven very difficult to extract all the dynamic dependencies. However, the model presented in this thesis is applicable also for the development of new systems able to predict the effects of dynamic configurations. If systems are developed with support for dependency analysis it is possible to obtain complete dependency information. This information could later be used to support dynamic configurations.

Matrix and list representations have been experimented with as different implementations of dependencies between components. The matrix approach requires much memory especially when the dependency graph is sparse, but it has advantages such as easy understanding and fast calculation of transpose and transitive closure. On the other hand the list representation is less memory consuming but makes use of more complex algorithms for the calculations.

To mature software development, engineering methods, such as configuration management architectures and development models related to the component development paradigm, must be introduced. There are no “silver bullets”, instead the combination of many different techniques is needed.

8 References

[1] Sterling, Cool:Spex, http://www.cool.sterling.com/products/Spex/index.htm.

[2] Rational, Clear Case, http://www.rational.com/products/clearcase.

[3] Rational, Clear Quest, http://www.rational.com/products/clearquest.

[4] Microsoft, COM, http://www.microsoft.com/com .

[5] Continuus Software Corporation, Continuus, http://www.continuus.com/.

[6] OMG, CORBA, http://www.omg.org/corba.

[7] Sun Microsystems, JAVA, http://java.sun.com.

[8] IBM, JCentral, http://www.ibm.com/java/jcentral/basic-search.html.

[9] Merant, PVCS Tracker, http://www.merant.com/products/pvcs/tracker/.

[10] Elsitech, Visual Intercept, http://www.elsitech.com/.

[11] Microsoft, Windows Installer, http://www.microsoft.com/.

[12] Asklund, U., *Configuration Management for Distributed Development - Practice and Needs*, Dissertion 10, Department of Computer Science Lund University, 1999.

[13] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*, Addison-Wesley, 1998.

[14] Bosch J., *Design & Use of Software Architectures,* Addison-Wesley, 2000.

[15] Box D., *Essential COM*, Addison-Wesley, 1998.

[16] Brown A. W. and Wallnau K. C., "Engineering of Component-based Systems", In *Proceedings of 2nd international conference on Engineering of Complex Computer Systems*, IEEE Computer Society, 1996.

[17] Brown A.W. and Wallnau K. C., The Current State of CBSE, *IEEE Software*, volume 15, issue 5, 1998.

[18] Cherinka R., Overstreet C. M., and Ricci J., "Maintaining a COTS integrated solution-are traditional static analysis techniques sufficient for this new programming methodology?", In *Proceedings of international conference on Software maintenance*, IEEE Computer Society, 1998.

[19] Conradi R. and Westfechtel B., Version Models for Software Configuration Management, *ACM Computing Surveys*, volume 30 ,issue 2, 1998.

[20] Cook J. E. and Dage J. A., "Highly Reliable Upgrading of Components", In *Proceedings of 21st International Conference on Software Engineering*, ACM Press, 1999.

[21] Crnkovic, I., *Large Scale Software System Management*, Ph.D. Thesis, Departmnet of Electrical Engineering, University of Zagreb, 1991.

[22] Crnkovic I., "Experience with Change-oriented SCM Tools", In *Proceedings of 7th Symposium on Software Configuration Management*, Lecture notes in Computer Science, nr 1235, Springer Verlag, 1997.

[23] Crnkovic I. and Larsson M., "A Case Study: Demands on Component-based Development", In *Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000.

[24] Crnkovic I., Larsson M., and Lüders F., "Software Process Measurements using Software Configuration Management", In *Proceedings of 11th European Software Control and Metrics Conference*, IEEE Computer Society, 2000.

[25] Dart S., "Concepts in Configuration Management Systems", In *Proceedings of 3rd International workshop on Software Configuration Management*, ACM Press, 1991.

[26] Dart S., "Content Change Management: Problems for Web Systems", In *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1675, Springer Verlag, 1999.

[27] Estublier J., "Software Configuration Management: A Roadmap", In *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, ACM Press, 2000.

[28] Eve J. and Kurki-Suonio R., On computing the transitive closure of a relation, *Acta-Informatica*, volume 8, issue 4, 1977.

[29] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[30] Garlan D., Allen R., and Ockerbloom J., Architectural Mismatch: Why Reuse is so Hard, *IEEE Software*, volume 12, issue 6, 1995.

[31] Grimaldi R. P., *Discrete and Combinatorial Mathematics*, Adison-Wesley, 1999.

[32] Hoek A. v. d., "Capturing Product Line Architectures", In *Proceedings of 4th International Software Architecture Workshop*, ACM Press, 2000.

[33] Hoek A. v. d., Hall R. S., Heimbigner D., and Wolf A. L., "Software Release Management", In *Proceedings of 6th European Software Engineering Conference*, Lecture Notes on Computer Science, nr 1301, Springer, 1997.

[34] Isovic D., Lindgren M., and Crnkovic I., "System Development with Real-Time Components", In *Proceedings of 1st workshop on Pervasive Components*, 2000.

[35] Josefsson, Margareta, Programvarukomponenter i praktiken -att köpa tid och prestera mer, report V040078, Sveriges Verkstadsindustrier, 1999.

[36] Korel B., "Black-Box Understanding of COTS Components", In *Proceedings of 7th international workshop on program comprehension*, 1999.

[37] Lampson B., "How Software Components Grew Up and Conquered the World", In *Proceedings of 21st International Conference on Software Engineering*, ACM Press, 1999.

[38] Larsson M. and Crnkovic I., "New Challenges for Configuration Management", In *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1675, Springer Verlag, 1999.

[39] Larsson M. and Crnkovic I., "Component Configuration Management", In *Proceedings of 5th Workshop on Component Oriented Programming*, 2000.

[40] Larsson M. and Crnkovic I., "Development Experiences from a Component Based System", In *Proceedings of 7th Engineering Conference on Computer Based Systems*, IEEE Computer Society, 2000.

[41] Le Lann G., "An analysis of the Ariane 5 flight 501 failure - a system engineering perspective", In *Proceedings of 4th international conference of engineering of computer based systems*, IEEE Computer Society, 1997.

[42] Morisio M., Seaman C. B., Parra A. T., Basil V. R., Kraft S. E., and Condon S. E., "Investigating and Improving a COTS-Based Software Development Process", In *Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000.

[43] Nuutila, E., *Efficient Transitive Closure Computation in Large Digraphs*, Ph.D. Thesis, Department of Mathematics and Computing in Engineering, Helsinki University of Technology, Finland, 1995.

[44] Olsson, Kent and Karlsson, Even-André, Daily Build - The Best of Both Worlds Rapid Development and Control, report V040083, Sveriges Verkstadsindustrier, 1999.

[45] Oreizy P., Medividovic N., and Taylor R. N., "Architecture-Based Runtime Evolution", In *Proceedings of 20th international conference on software engineering*, ACM Press, 1998.

[46] Peltz C., "A Hierarchical Technique for Composing COM based Components", In *Proceedings of 2nd international workshop on CBSE*, SEI, 1999.

[47] Rege K., "Design Patterns for Component-Oriented Development", In *Proceedings of 25th EUROMICRO in the workshop on Software Process and Product Improvment*, IEEE Computer Society, 1999.

[48] Rogerson D., *Inside COM*, Microsoft Press, 1996.

[49] Roman E., *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*, Wiley, 1999.

[50] Sametinger J., *Software Engineering with Reusable Components*, Springer, 1997.

[51] Seacord R.C., Hissam S. A., and Wallnau K. C., AGORA: a search engine for software components, *IEEE Internet Computing*, volume 2, issue 6, 1998.

[52] Shaw M. and Garlan D., *Software Architecture: Perspectivs on an Emerging Discipline*, Prentice-Hall, 1996.

[53] Standish T. A., *Data Structures, Algorithms & Software Principles in C*, Addison-Wesley, 1995.

[54] Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[55] Takeshita T., "Metrics and risks of CBSE", In *Proceedings of 5th international symposium on software tools and technologies*, IEEE Computer Society, 1997.

[56] Thane, H., *Monitoring, Testing and Debugging of Distributed Real-Time Systems*, Ph.D Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000.

[57] Voas J., Cerifying Off-the-Shelf Software Components, *IEEE Computer*, volume 31, issue 6, 1998.

[58] Voas J., COTS Software: the Economical Choice?, *IEEE Software*, volume 15, issue 2, 1998.

[59] Warshall S., A Theorem on Boolean Matrices, *Journal of the ACM*, volume 9, issue 1, 1962.

[60] Wille C., *Presenting C#*, SAMS Publishing, 2000.

[61] Zeller A., "Versioning System Models Through Description Logic", In *Proceedings of 8th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1439, Springer Verlag, 1998.

DEVELOPMENT EXPERIENCES FROM A COMPONENT-BASED SYSTEM

Magnus Larsson & Ivica Crnkovic

Abstract:

Building software systems with reusable components brings many advantages. If the reuse concept is utilized on several levels of a system development, the development becomes more efficient, the reliability of the products is enhanced, and the maintenance requirement is significantly reduced. The levels of reuse are spread out from the reuse of source code and common libraries, through the reuse of large business components, up to the reuse of the standard products in the configuration of large systems. Designing, developing and maintaining components for reuse is, however, a very complex process which places high requirements not only for the component functionality and flexibility, but also for the development organization. In this paper, we discuss the different levels of component reuse, and certain aspects of component development. As an illustration of reuse issues, we present a successful implementation of a component-based system, which is widely used for industrial process control.

1 Introduction

Reuse and an open component-based architecture are the keys to the success of systems with a long lifecycles. Designing a system that supports this approach, requires more effort in the design phase and the time to market might be longer, but in the long run, the reusable architecture will prove profitable. The reuse concept can be used on different levels: On a low level it is a reuse of source-code, and small-size components. More reuse is obtained with larger components encapsulating business functions. Finally, the integration of complete products in complex systems can be seen as the highest level of reuse. On each level of reuse there are specific demands on the reusable components, on the component management and on the integration process.

This paper describes important issues related to the development and maintenance of reusable components and as an example uses the ABB Advant industrial process control system. In chapter 2 we give an overview of the Advant system design and in chapter 3 the main characteristics of Advant reusable components. Chapter 4 outlines all the development and maintenance aspects of a component based system, which must comply with customer requirements. During evolution of the system new technologies

were developed which resulted in the appearance on the market of many components with the same functionality as the proprietary ones. The fact that new components must be incorporated into the existing systems introduces new demands on the system development process. These new issues are discussed in chapter 5.

2 ABB Advant Open Control System

ABB is a global electrical engineering and technology company, serving customers in power generation, transmission and distribution, in industrial automation products, etc. The ABB group is divided into companies, one of which, ABB Automation Products AB, is responsible for development of industrial automation products. The automation products encompass several families of industrial process-control systems including both software and hardware.

The main characteristics of these products are reliability, high quality and compatibility. These features are results of responses to the main customers requirements: The customers require stable products, running around the clock, year after year, which can be easily upgraded without impact on the existing process. To achieve this, ABB uses a component-based system approach designing the extendable and flexible systems.

The Advant Open Control System (OCS) [1] is component-based to suit different industrial applications. The range includes systems for Power Utilities, Power Plants and Infrastructure, Pulp and Paper, Metals and Minerals, Petroleum, Chemical and Consumer Industries, Transportation systems, etc. An overview of the Advant system is shown in Figure 1.

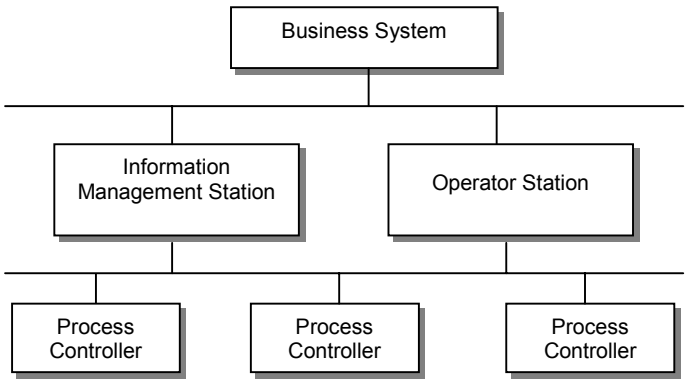


Figure 1. An overview of the conceptual architecture of the Advant open control system.

Advant OCS performs process control and provides business information by assembling a system of different families of Advant products. Process information is managed at the level of process controllers. The process controllers are based on a real-time operating system and execute the control loops. The Operator Station (OS) and Information Management Station (IMS) gather and supervise product information, while the business system analysis information for optimization of the entire processes.

Advant products use standard and proprietary communication protocols to satisfy real-time requirements.

2.1 Designing for Reuse

The Advant system architecture is designed for reuse. Different products such as Operator and Information Management Stations are used as system components in assembling complete systems. Examples of the products are. The two operator station versions, Master OS and MOD OS are used in building different types of operator applications.

Having up-to-date information at the right time and in the right place is critical to the success of any industrial operation and Advant OCS therefore includes information management functions with real-time insight into all aspects of the process controlled. Advant Information Management has an SQL-based relational database accessible to resident software and all surrounding computers. Historical data acquisition reports, versatile calculation packages and an application programming interface (API) for proprietary and third party applications are examples of the functionality provided. Advant components have access to process, production and quality data from any Process Control unit in a plant or in an Intranet domain.

2.1.1 Scalability.

Advant OCS can be configured in a multitude of ways, depending on the size and complexity of the process. The initial investment can consist of stand-alone process controllers and, optionally, local operator stations for control and supervision of separate machines and process sections. Subsequently, several process controllers can be interconnected and, together with central operator and information management stations build up a control network. Several control networks can be interconnected to give a complete plant network which can share centrally located operator, information and engineering workplaces.

2.1.2 Openness.

The system is further strengthened by the flexibility to add special hardware and software for specific applications such as weighing, fixed- and variable-speed motor drives, safety systems and product quality measurements and control in for example the paper industry. Second- and third party administrative, information, and control can also be easily incorporated

2.1.3 Cost-effectiveness.

The step-by-step expansion capability of Advant OCS allows users to add new functionality without making existing equipment obsolete. The system’s self-

configuration capability eliminates the need for engineers to enter or edit topology descriptions when new stations are physically installed. New units can be added while the system is in full operation. With Advant OCS, system expansion is therefore easy and cost-effective.

2.2 Designing with Reuse

Designing with reuse of existing components has many advantages [2]. The software development time can be reduced and the reliability of the products increased. These were important prerequisites for the Advant OCS development.

Advant OCS products can be assembled in many different configurations for use in various branches of industry. Specific systems are designed with the reuse of Advant OCS products and other external products. This means which customers get a tailor-made system that meets their needs. External products and components can be used together with the Advant OCS due to the openness of the system. For example a satellite communication component, which is used to transmit data from the offshore station to the supervision system inland, can be integrated with the Advant OCS (Figure 2).

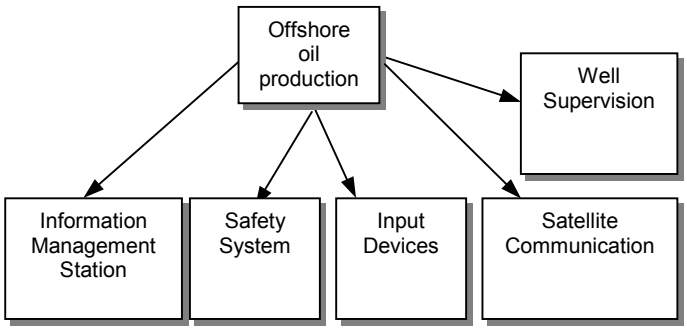


Figure 2. Solution for an offshore oil production platform.

The offshore system in Figure 2 uses the Information Management Station to gather all relevant data from the oil producing process and this is then transmitted to the headquarters on shore via the external satellite component. A safety component is used to provide a more secure system. Another component is the well supervision unit which monitors the oil wells.

Component-based systems for different types of applications can be easily designed and produced because of the open and scalable architecture of Advant OCS.

2.3 Experiences

The Advant system is a successful system that can be used to build different types of process automation systems. It has effective build and integration procedures. The main

reason for the success is - component-based architecture and the component features (flexibility, robustness, stability and compatibility).

However, the cost to achieve these features has been high. To be able to suit the requirements of an open system, new ABB products had always to be backward compatible. It would have been easier to develop a new system that did not have to be compatible with the previous systems. To guaranty that the system is backward compatible works as a warranty that the current system will integrate with new products and this makes the system trustworthy. The system is carefully designed and a lot of effort has been put in test and maintenance.

Development with big components that are easy to reuse increase the efficiency significantly compared to reusing a smaller component that could have been developed in-house to the same cost as buying it. The Advant OCS products are examples of big components that have been used to assembly process automation systems.

3 Reusable Components

3.1 Components

The Advant OCS products are component based to minimize the maintenance and development cost. Figure 3 shows the component architecture of the operator station.

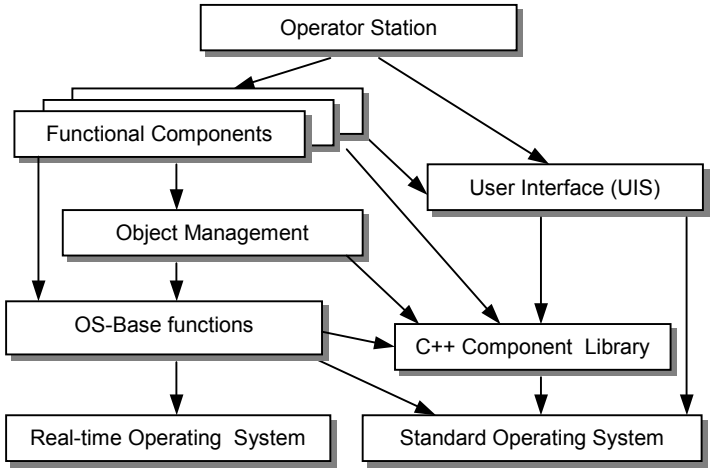


Figure 3. The operator workstation is assembled from components.

The operator station consists of a specific number of functional components and of a set of standard Advant components. These components use the User Interface System (UIS) component. Object Management Facility (OMF) is a component which handles the infrastructure and data management. OMF is similar to CORBA [3] in that it provides a distributed object model with data, operation and event services. The UxBase component provides drivers and other specific operating system functions. Helper

classes for strings, lists, pointers, maps and other general-purpose classes are available in the C++_complib component. The components are built upon operating systems, one, a standard system(such as Unix or Windows), and the other a proprietary real-time system.

To illustrate different aspects of component-based development and maintenance, we shall further look at two components:

- Object Management Facility (OMF), a business type of component with a high-level of functionality and a complex internal structure;
- C++_complib is a basic and a very general component.

3.2 Object Management Facility (OMF)

OMF is object-oriented middle-ware for industrial process automation. It encapsulates real-time process control entities of almost every conceivable description into objects that can be accessed from applications running on different platforms, for example Unix and Windows NT. Programming interfaces are available for many languages such as C, C++, Visual Basic, Java, Smalltalk and SQL while interfaces to the IEC 1131-3 [4] process control languages are under development. OMF is also adapted to Microsoft Component Object Model (COM) via adapters and another component called OMF COM aware. The adapters for OPC (OLE for Process Control) [6] and OLE Automation are also implemented. Thanks to all these software interfaces, OMF makes process and production data available to the majority of computer programmers and users i.e. even to those not necessarily involved in the industrial control field. For instance, it is easy to develop applications in Microsoft Word, Excel and Access to access process information. OMF has been developed for demanding real-time applications, and incorporates features, such as real-time response, asynchronous communications, standing queries and priority scheduling of data transfers. On one side OMF provides industry-standard interfaces to software applications, and on the other, it offers interfaces to many important communication protocols in the field (see Figure 4), including MasterNet, MOD DCN, TCP/IP and Fieldbus Foundation. These adapters make it possible to build homogeneous control systems out of heterogeneous field equipment and disparate system nodes.

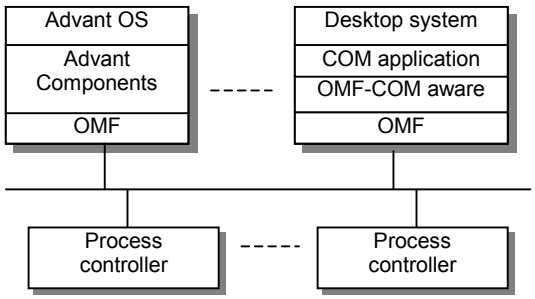


Figure 4. Many different components and products use the OMF component.

OMF reduces the time and cost of software development by providing frameworks and tools for a wide range of platforms and environments. These utilities are well integrated into their respective surroundings, allowing developers to retain the tools and utilities they prefer to work with.

3.3 C++_complib

C++_complib is a class library that contains general-purpose classes, such as containers, string management classes, file management classes, etc. The C++_complib library was developed when no standard libraries, such as STL [5], were available on the market. The main purpose of having this library was to improve the efficiency and the quality, and to get the uniform usage of the basic functions. The c++_complib library is reused whenever possible even in cases in which similar services are supported by a specific platform or development package. The library was ported to several platforms, and in some cases the implementation part had small variations. The declaration part is the same on all platforms.

4 Different Reuse Aspects

4.1 Component generality and efficiency

Reuse principles place high demands on the reusable components. The components must be sufficiently general to cover the different aspects of their use. At the same time they must be concrete and simple enough to serve a particular requirement in an efficient way. Developing a reusable component requires three to four times more resources than developing a component, which serves a particular case [7]. In the case of C++_complib, the situation was simpler, because the requirements from the theoretical point of view were clear. It was relatively easy to define the interface, which was used by different components in the same way. The situation was more complicated with complex components, such as OMF. Although the basic concept of component functionality was clear, the demands on the component interface and behavior were different in different components and products. Some components required a high level of abstraction, others required the interface to be on a more detailed level. These different types of requirements have led to the creation of two levels of components: OMF base, including all low-level functions, and OMF framework, containing only a higher level of functions and with more pre-defined behavior and less flexibility.

4.2 Evolution of Functional Requirements

The development of reusable components would be easier if functional requirements did not evolve during the time of development. As a result of new requirements for the products, new requirements for the components will be defined. The more reusable a component is, the more demands are placed on it from products using that component.

A number of the requirements coming from different products, may be the same or very similar, but this is not necessarily the case for all requirements passed to the components. In addition to the requirements stated from the products, a component also collects demands on the internal behavior (for example code improvement, improvement of the maintainability, etc.). This means that the number of requirements of reusable components grow faster than of particular products or of a non-reusable piece of software. To satisfy these requirements the components must be updated more rapidly and the new versions must be released more frequently than the products using them.

The evolution process is illustrated in Figure 5. The first graph shows the growing number of requirements for certain products. The number of requirements of a common component grows faster. Some of the product requirements are satisfied with each new release of a product, which are shown as steps on the second graph. The component satisfying the requirements by its releases, which normally precede the releases of each product.

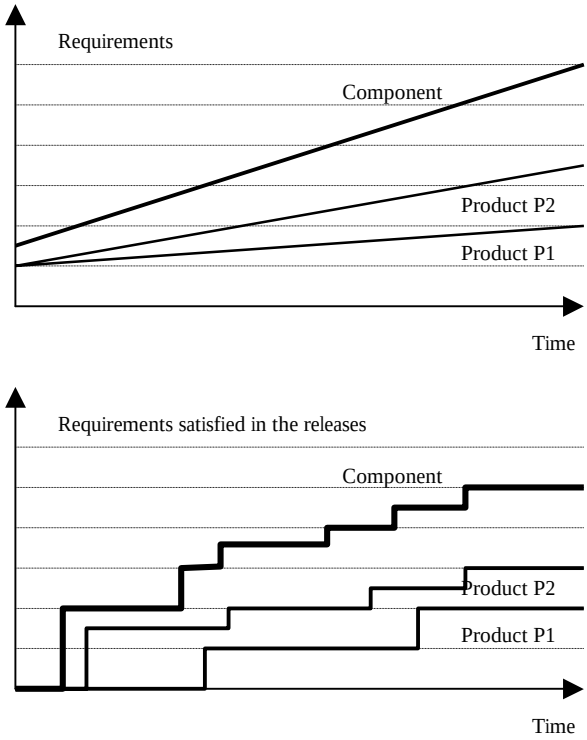


Figure 5. To satisfy the requirements the reusable component must be modified more often.

Indeed this was the case with both components we are analyzing here: New functions and classes were required from C++_complib, and new adapters and protocol support were required from OMF. The development time for these components was significantly shorter than for products: While new versions of a product are typically

released each six months, new versions of components are released as least twice as often.

4.3 Migration Between Different Platforms

During their several years of development, Advant products have been ported on different platforms. The reasons for this were the customer requirement, that the products should run on specific platforms, and general trends in the growing popularity of certain operating systems. Of course, at the same time, new versions and variants of the platform already used appeared, supporting new, better and cheaper hardware. Figure 6 shows the migration path of Advant products on different platforms.

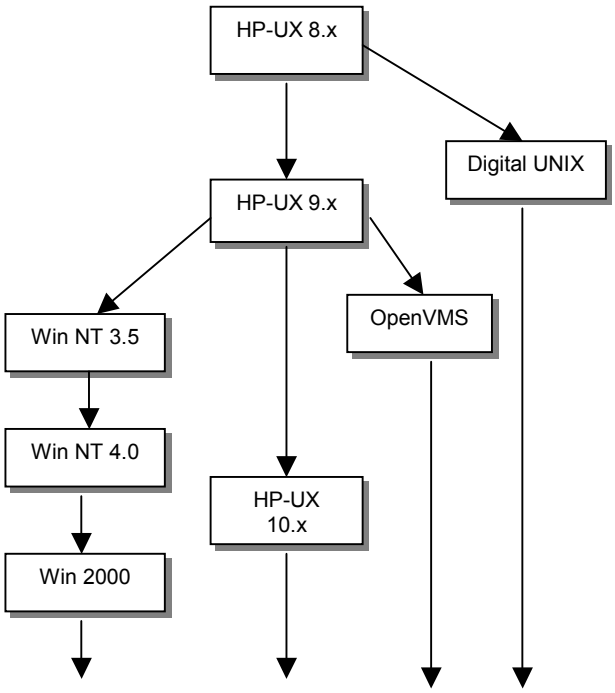


Figure 6. Different platforms supported by OMF.

As an important part of the reuse concept was to keep the high-level components unchanged as far as possible, it was decided to encapsulate the differences between operating systems in low-level components. This concept works, however, only to some extent. The minimal activity required for each platform is to rebuild the system for that platform. To make it possible to rebuild the software on every platform, standard-programming languages C and C++ have been used. Unfortunately, different implementations of the C++ standard in different compilers, caused problems in the code interpretation and required the rewriting of certain parts of the code. To ensure that standard system services are available on all platforms, the POSIX standard has been used. POSIX worked quite well on different Unix platforms, but much less so on Windows NT. The second level of compatibility problem was Graphical User Interface (GUI). The main dilemma was whether to use exactly the same GUI on every platform,

or to use the standard "look and feel" GUI for each platform. This question applied particularly on NT in relation to Unix platforms. Experience has shown that it is not possible to give a definitive answer. In some cases it was possible to use the same GUI and the same graphical packages, but in general, different GUIs were implemented.

The main work regarding to the reuse of code on different platforms was performed on low-level components, such as UxBase and OMF. While UxBase provides different low-level packages for every platform (for example different drivers), OMF capsulated the differences directly in the code using conditional compilation. OMF itself is designed in such a way that it was possible to divide the code into two layers. One layer is specific for each operating system, and the other layer, with the business logic, is implemented for all of the supported platforms. Reuse issues on different platforms for C++_complib were easier, strictly the package contains general algorithms, which are not hard connected to a specific operating system. Some problems appeared however, related to different characteristics of compilers on different platforms.

4.4 Compatibility

One of the most important factors for successful reusability is the compatibility between different versions of the components. A component can be replaced easily or added in new parts of a system if it is compatible with its previous version. The compatibility requirements are essential for Advant products, since smooth upgrading of systems, running for many years, is required. Compatibility issues are relative simple when changes introduced in the products are of maintenance and improvement nature only. Using appropriate test plans, including regression tests, functional compatibility can be tested to a reasonable extent. More complicated problems occur when new changes introduced in a reusable component eliminate the compatibility. In such a case, additional software, which can manage both versions, must be written.

A typical example of such an incompatible change, is a change in the communication protocol between OMF clients and servers. All different versions of OMF must be able to talk to each other to make the system flexible and open (Figure 7 It is possible to have different combinations of operating systems and versions of OMF and it still works. This has been solved with an algorithm that ensures the transmission of correct data format. If two OMF nodes have the same version, they talk in their native protocol.

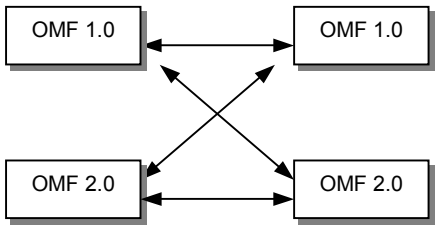


Figure 7. Different versions of OMF must be compatible with all older versions.

If an old OMF node talks with a new, the new OMF is responsible for converting the data to the new format, this being designated RMIR ("receiver makes it right"). If a new OMF sends data to an older, the older OMF can not convert the data since it is unaware of the new protocol. In this case the newer OMF must send in the old protocol format, SMIR ("sender makes it right"). This algorithm builds on that fact all machines know about each other and that they also know what protocol they talk. However, if an OMF-based node does not know of the other node then it can always send in a predefined protocol referred to as “well known format”. All nodes do recognize this protocol and can translate from it. This algorithm minimizes the number of data conversions between the nodes.

In the case of C++_complib the problems with compatibility were somewhat different. New demands on the same classes and functions appeared because of new standards and technology. One example is the use of C++ templates. When the template technology became sufficiently mature, the new requirements were placed for C++_complib: All the classes were to be re-implement as template classes. The reason for this was the requirement for using basic classes in a more general and efficient way. Another example was a Unicode support in addition to ASCII-support. These new functions were added by new member-functions in the existing classes and by adding new classes using the inheritance mechanism for reusing the already existing classes.

4.5 Development Environment

When developing reusable components several dimensions of the development process must be considered:

- Support for development of components on different platforms;
- Support for development of different variants of components for different products;
- Support for development and maintenance of different versions of components for different product versions.

To cope with these types of problems, it is not sufficient to have appropriate product architecture and component design. Development environment support is also essential.

The development environment must permit an efficient work in the project - editing, compiling, building, debugging and testing. Parallel and distributed development must also be supported, because the same components are to be developed and maintained at the same time on different platforms. This requires the use of a powerful Configuration Management (CM) tool, and definition of an advanced CM-process.

The CM process support exists on two levels. First on the source-code level, in which source-code files are under version management and binary files are built. The second level is the product integration phase. The product built must contain a consistent set of the component versions. For example, Figure 8 shows an inconsistent set of components. The product version P1-V2 uses the component versions C1-V2 and C2-

V2. At the same time the component version C1-V2 uses the component version C2-V1, an older version. Integrating different versions of the same component may cause unpredictable behavior of the product.

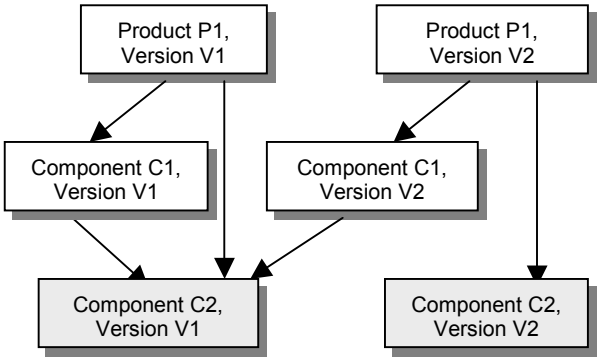


Figure 8. An inconsistent component integration.

Another important aspect of CM in developing reusable component is Change Management. Change management keeps track of changes on the logical level, for example error reports, and manages their relations with implemented physical changes (i.e. changes of source code, documentation, etc.). Because change requests (for example functional requirements or error reports) come from different products, it is important to register information about the source of change requests. It is also important to relate a change request from one product to other products. The following questions must be answered: What impact can the implemented change have on other products? If en error appears in one product, does it appear in other products? Possible implications must be investigated, and if necessary, the users of the products concerned must be informed.

The development environment designated Software Development Environment (SDE) [8] is used in developing Advant products. It is an internally built program package, which encapsulates different tools, and provides support for parallel development. The CM tool, based on RCS [9] provides support for all CM disciplines, such as Change Management, WorkSpace Management, Build Management, etc. SDE runs on different platforms, with slightly modified functions. For example, the build process is based on *Makefiles* and *autoconf* on Unix platforms, while Microsoft Developer Studio with additional *Project Settings* is used on Windows NT. The main objective of SDE is to keep the source-code in one place under version control. Using baselines, and change requests, the different versions of components are managed. The whole development process is complex and requires an organized and planned support, but it is unavoidable for an efficient and successful development of and with reusable components.

5 A New Paradigm -Standard Components

In recent years the demands of customers on systems have changed. Customers require integration with standard technologies and the use of standard applications in the products they buy. This is a definite trend on the market but there is little awareness of the possible problems involved. An improper use of standard components can cause severe problems, especially in distributed real-time and safety-critical systems, with long-period guarantees. In addition to these new requirements, time-to-market demands have become a very important factor.

These factors and other changes in software and hardware technology have introduced a new paradigm in the development process [10]. The development process is focused now on the use of standard and de-facto standard components, outsourcing, COTS and the production of components. At the same time, final products are no longer closed, monolith systems, but are instead component-based products that can be integrated with other products available on the market.

This new paradigm in the development process and marketing strategy has introduced new problems and raised new questions:

- The development process has been changed. Developers are now not only designers and programmers, they are also integrators and marketing investigators. Are the new development methods established? Are the developers properly educated?
- What are the criteria for the selection of a component? How can we guarantee that a standard component fulfills the product requirements?
- What are the maintenance aspects? Who is responsible for the maintenance? What can be expected of the updating and upgrading of components? How can we satisfy the compatibility and reliability requirements?
- What is the trend on the market? What can we expect to buy not only today but also on the day we begin delivering our product?
- When developing a component, how can we guarantee that the "proper" standard is used? Which standard will be valid in five, ten years?

All these questions must be considered before beginning a component-based development project. Josefsson [11] presents certain recommendations to the component integrator for use as guidelines: Test the imported component in the environment where it is to run and limit the practical number of component suppliers to minimize the compatibility problems. Make sure that the supplier is evaluated before a long-term agreement is signed.

The focus of development environment support should be transferred from the “edit-build-test” cycle to the “component integration-test” cycle. Configuration management must give more consideration to rune-time phase [12].

5.1 Replacing Internal Component With Standard Components

In the middle of the eighties, ABB Advant products were completely proprietary systems with internally developed hardware, basic and application software. In the beginning of the nineties, standard hardware components and software platforms were purchased while the real-time additions and application software were developed internally. The system is now developed further using components based on new, standard technologies.

During this development, further new components become available on the market. ABB faced this issue more than once. At one point in time, it was necessary abandon the existing solutions in a favor of new solutions based on existing components and technologies. To illustrate the migration process we the discuss possibility of replacing OMF and C++_complib with standard components.

Experience from these examples showed that it is easier to replace component if the replacement process is made in small incremental steps. Allowing the new component to coexist with the old one makes it easier to be backward compatible and the change will be smooth.

5.2 Replacing OMF with DCOM

Moving from a UNIX based system to a system based on Windows NT had serious affect on the system architecture. Microsoft components using a new object model were available, namely COM/DCOM [13]. DCOM has functionality similar to that of OMF and this became a new issue when DCOM was released. Should ABB continue to develop its proprietary OMF or change to a new standard component? The problem was that DCOM did not have all the functionality of OMF and vice versa. The domains overlap only partially as shown in Figure 9

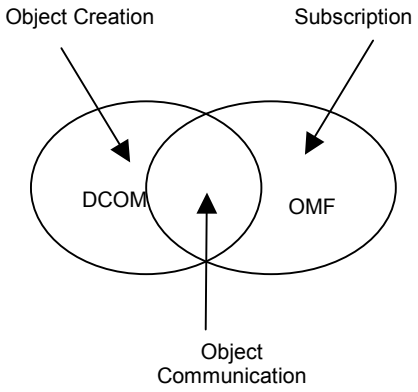


Figure 9. The functionality domains of OMF and DCOM do not overlap completely.

A subscription of data with various capabilities can be made in OMF, and this subscription functionality is not supported by DCOM.

On the other hand, DCOM can create objects when they are required and not like OMF where objects are created before the actual use of them. Both technologies support object communication and in this area it is easier to replace OMF. with DCOM.

If the decision was made to continue with OMF, all the new components that run on top of COM could not be used, which would drastically reduce the possibilities of integration with other, third-party components. On the other hand, it would require considerable work to make the current system run on top of COM. This was the dilemma of COM vs. OMF.

To begin with OMF was adapted to COM with an adapter designated OMF COM aware. This functionality helped COM developers access OMF objects and vice versa. However, this solution to the problem using two different object models was not optimal since it added overhead in the communication. Nor it was possible to match the data types one to one, which made the solution limited. A decision was taken to build the new system on COM technologies with proprietary extensions adding the functions missing from COM. All communication with the current system was to be through the OMF COM. Adapters are very useful when a new component is to used in parallel with an existing one [14]. This solution makes it easy to remove the old OMF and replace it with COM in small steps over time.

5.3 Replacing C++_complib with STL

To switch from C++_complib to STL [5] was much easier because STL covers almost all the C++_complib functions and provides additional functionality. Still, much work reminded do since all the code using C++_complib had to be changed to be able to use STL instead. The decision was taken to continue using both components and to use STL whenever new functionality was added. After a time the use of old components was reduced and the internal maintenance cost reduced. In some cases in the same components both libraries were used, which gave some disadvantages, especially in the maintenance process.

6 Conclusion

The Advant OCS has been used as an example of a successful component based system and we have shown what it means to develop with components that fit into a large software system. A careful design and awareness of future demands on components are necessary to be able to integrate the existing system with new technologies. When Advant OCS was developed no one really though about Windows NT and ABB had to pay the price for that when it suddenly became clear that Windows NT would be the next operating platform. It was possible to move from one platform to another, but the cost was greater than if the design would have been more independent from the platform. We have shown certain problems with developing reusable components and given examples for this. The experiences from the development of Advant OCS has been that it is better to put more effort to create an open and extendable architecture than to rush the development focusing on only current technologies.

7 References

[1] Advant, ABB Automation Products, http://www.advantocs.com

[2] Sommerville I., *Software Engineering*, Addison-Wesely, 1999

[3] CORBA, http://www.corba.org

[4] International Electrotechnical Commision (1992), Programmable Controllers Part 3, Programming Languages, IEC 1131-3, IEC Geneva.

[5] Austern M., Generic Programming and the STL, Addison-Wesely, 1999

[6] OPC Foundation, http://www.opcfoundation.org

[7] Szyperski C., Component Software, Addison Wesely, 1999

[8] Crnkovic I., *Experience with Change-Oriented SCM Tools*, Software Configuration Management ICSE'97 Symposium, 1997, proceedings, Springer

[9] Tichy W., RCS - *A System for Version Control*, *Software and Practice Experience*, 15(7):635-654, 1985

[10] Aoyama M.: *New Age of Software Development: How Component-Based Software* Engineering *Changes the Way of Software Development*, 1998 International Workshop on CBSE

[11] Josefsson M., Oskarsson Ö., *Programvarukomponenter i praktiken – att köpa tid och prestera* mer, Report from Sveriges Verkstadsindustrier 1999

[12] Larsson M., Crnkovic I., *New Challenges for Configuration Management*, System Configuration Management Symposium, 1999, proceedings, Springer

[13] Box D., *Essential COM*, Addison-Wesley, ISBN 0-201-63446-5

[14] Rine D., Nada N., Jaber K., *Using Adapters to Reduce Interaction Complexity in Reusable* Component-*Based Software Development*, Proceedings of the fifth symposium on software reusability, ACM Press, 1999

NEW CHALLENGES FOR CONFIGURATION MANAGEMENT

Magnus Larsson & Ivica Crnkovic

Abstract:

More and more systems are developed using components. There is a move from monolithic to open and flexible systems. In such systems, components are upgraded and introduced at run-time, which affects the configuration of the complete system. Keeping up-to-date information about which components are installed is a problem. Updating a component also affects the compatibility of the system. It is therefore important to keep track of changes introduced in the system. In the product life cycle, CM is traditionally focused on the development phase, in particular on managing source code. Now when changes are introduced in systems at run-time and systems are component-based, a new discipline, component configuration management *is required. This paper analyses component management and highlights the problems related to component configuration. Requirements on component configuration management are outlined, and some directions to possible solutions of the problems are given.*

1 Introduction

In recent years we have recognized a new paradigm in the development process: From a complete in-house development, to a development process which has focused on the use of standard and de-facto standard components[1], outsourcing, COTS (commercials off the shelf). The final products are not closed, monolithic systems, but are instead component-based products which can be integrated with other products available on the market [3]. Developers are not only designers and programmers, they have become integrators and marketing investigators. The new paradigm increases the efficiency of development and the flexibility of delivered products, but at the same time increases the risk of losing product configuration consistency. The higher risk reduces the product reliability, which is a critical factor for certain types of systems, such as real-time and safety-critical systems. Configuration Management (CM) is a discipline, which controls the consistency between the parts of the entire system, and can increase the reliability of component-based products.

[1] Definitions of components are presented in chapter 3.

Software systems based on standard components are the results of a combination of pure development and integration of components. The requirements for conventional use of CM remains, but new requirements related to component management appear in all phases: in the design, integration and run-time. We can expect that the source code management will become less critical, because there is less internal development and because of the fact that source code management in CM is very well established in both theory and implementation. The integration part, i.e. configuration, and version management of the components becomes more important. New aspects of CM arise in the run-time phase, as components are usually loosely coupled, and their update is allowed in the run-time environment.

The importance of CM, and challenges in research and implementation of CM support, are emphasized in the 1998 CBSE (Component-based Software Engineering) workshop [2], as quoted: "In particular, high composeability in a product line setting amounts to mass customization and this introduces tremendous configuration management challenges and support challenges."

Although CM provides good support in the development phase, especially in the coding phase, there is a lack of CM disciplines managing components already developed. This paper points to certain new aspects of CM in managing components. Chapter 2 shows different phases in component development processes and run-time environments and their relations to CM activities. The different compatibility levels of the components are discussed in chapter 3. Chapter 4 gives an overview of the component characteristics related to CM issues. The problems, which appear due to the lack of proper CM support, are presented. Chapter 5 outlines certain models for improving the support and improving the reliability of products.

2 Using CM in Component-based Product Life Cycles

Configuration management is applied in different phases of a component-based product life cycle. Figure 1 shows an example of a development and run-time process. In the development phase we build libraries from the source code. A component is built by assembling libraries and collecting other types of items such as documentation and executable files. Finally, a typical component-based product consists of a set of components.

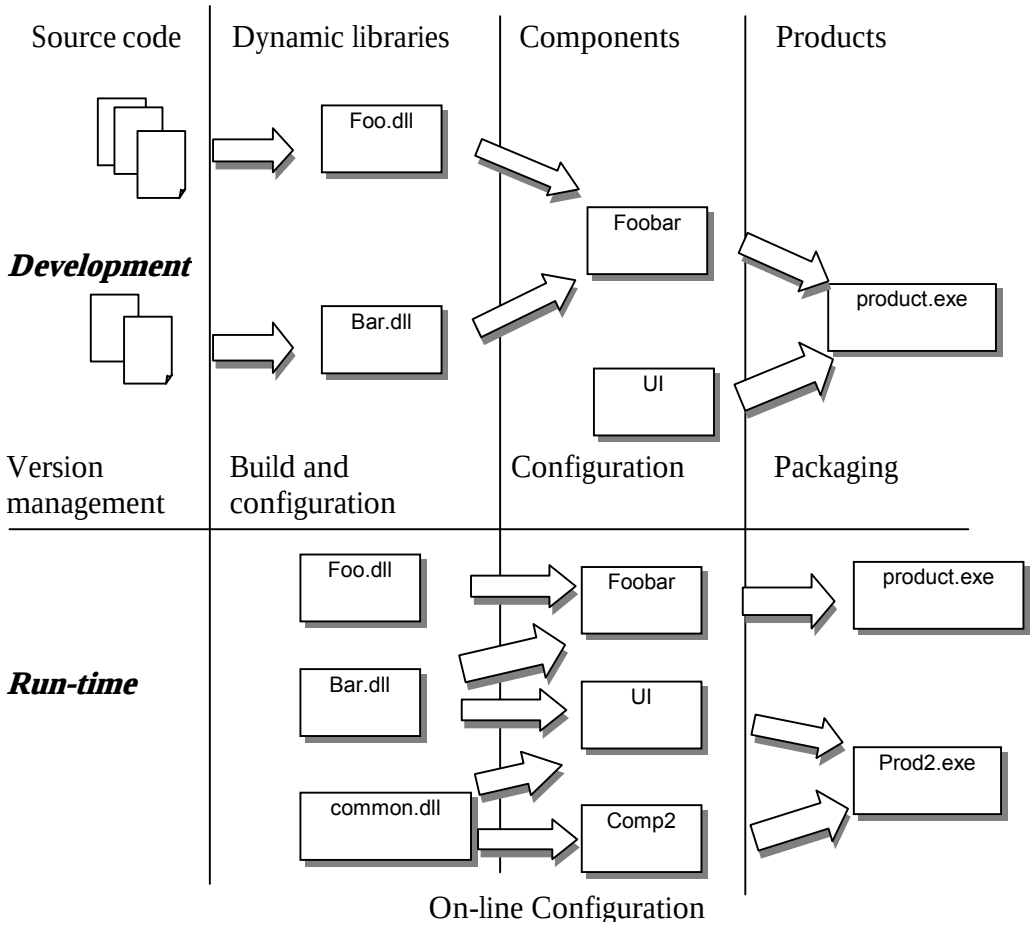


Figure 1. CM activities in different phases of a component-based product life cycle

In the first development phase, source code management is used to track the introduction of different versions of source code, to enable parallel development, etc. Many CM tools supporting this are available today. The building phase is also supported by CM tools such as different variants of make and configuration tools, the results of the building procedures being connected to the source code. One step closer to CM for components is to use description logic, to describe configurations, in combination with make to build a product[10]. However, this does not solve the run-time issues.

Having control over the source code and producing the system entirely from the source code makes it possible to control the target system configuration. When using imported components, we lose this control, because we only partially know their behavior. It is possible however to manage versions and configurations if we place the components under version control and deliver them as a part of the product.

When delivering components or products, which are part of a target system, we face two problems:

- We cannot predict the behavior of the entire environment of the target system. The system may contain another product, which uses the same component as our product. The relations between components, and the changes we may obtain by introducing a new version of a component, are uncertain.
- A more serious problem is the dynamic behavior of the system configuration in the run-time environment. If we permit component-updating during the run-time, by updating dynamic libraries, we could be facing a situation in which a new component version works for one product, but not for another. There are also different aspects of updating, such as moving or copying an application from one computer to another, or automatically generation of code.

CM can provide solutions to these problems, and those are new challenges for CM. To cope with the problem, the research and practical implementations must focus on the component management. The following chapters describe the mechanisms of component management and point at the problems related to their identification. Finally an outline of possible solutions for improvement of the component version and configuration management are presented.

3 Component Compatibility

There are different definitions of software components [1]: A component is a non-trivial, nearly independent, and replaceable part of a system which fulfills a clear function. A component conforms to and provides the physical realization of a set of interfaces. A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces which can be discovered at run-time. A component can be deployed independently and is subject to composition by third party.

The importance of components becomes significant where technologies for their development and integration are being standardized. The most prominent component technologies today are Java Beans, COM/DCOM and ActiveX, and CORBA. In this paper, we illustrate component-management problems using COM/DCOM technology, but the same principles are valid with other technologies.

A new component version might be added to introduce new functions in a system, or only to change its behavior, (better performance, better stability), without changing the interface. When replacing a component or a component version we must consider which type of change is permitted, and which type of compatibility is required. We define three levels of compatibility:

- *Input and Output compatibility*. A component requires input in a specific format and produces results in a defined format. The internal characteristics of the component are of no interest. An example of this type of compatibility is provided by different word-processors producing the same document format. This type of compatibility does not ensure that the interfaces or the behavior are preserved.

- *Interface compatibility* (at development time and at run time). The interface remains the same, but the implementation can be different. A typical example is given by different implementations of ActiveX objects, with the same interface. Interface compatibility is more demanding than input and output compatibility, but it does not need to have the same behavior.
- *Behavior compatibility*. Internal characteristics of the components, such as performance, resource requirements, etc., must be preserved. Such requirements can be appropriate for real-time systems. This is the strongest compatibility requirement and it includes the previous ones.

The compatibility criteria can be used in deciding if a component can be replaced or not. This decision can be especially important in case of a replacement "on the fly" in a run-time environment. It is important to maintain the required level of compatibility to avoid the risk of interrupting the whole system.

4 Managing Components

Components typically consist of shared libraries, where the component functions are implemented. The programs using components do not refer to the libraries directly but to the component interfaces. The libraries are implementations of the interfaces. We need to keep track of changes on both logical and physical levels as well as their relations. Both libraries and interfaces must be identified. Component Configuration Management must work on both levels. Versioning of interfaces is a more difficult task, because the interface is an abstraction without information about the physical representation. For this reason, we separate the problem of managing components onto two levels: Managing libraries and managing interfaces.

4.1 Libraries

Historically there were less problems in this area as all libraries were statically linked into the executables. This prevented the executable from being updated when a new version of the library was released. An advantage of this approach is that the executables are protected from uncontrolled use by the new version of the library. A disadvantage is the necessity to re-link the executable only to incorporate a new version of the library, which is unnecessary work when the library is interface-compatible. Another disadvantage is that all executables which shared the same library must be linked with their own copies of the library. The concept of shared libraries was introduced to avoid this. This was a significant improvement since we could now share libraries and make updates without re-linking the executables while functions were interface-compatible. In Microsoft platforms, shared libraries are designated dynamic link libraries or dlls, which can be loaded and unloaded whenever needed. On other platforms, such as different Unix platforms, shared libraries are loaded together with the main executable.

Unfortunately, the concept of shared libraries introduces new problems related to the consistency of the system, as illustrated by Figure 2.

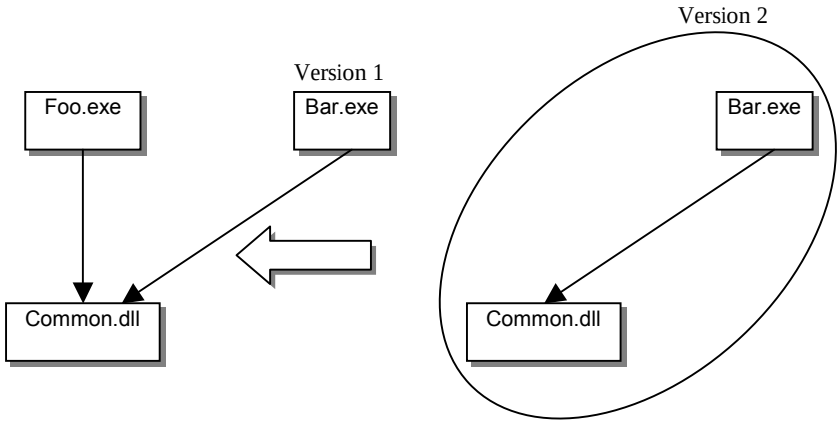


Figure 2. Foo.exe stops work when the new incompatible version of Common.dll is introduced.

The figure shows how a new version might damage the system. Common.dll version 1 will be overwritten with version 2 when the new version of bar.exe is introduced. The replacement could be successful if version 2 of Common.dll is interface-compatible with version 1, but definitely not if the compatibility level is less. There is a risk that Foo.exe will stop working after the new version of Common.dll is introduced.

The new interface-compatible version of Common.dll may contain undetected errors as it was tested with Bar.exe only and not with Foo.exe. Foo.exe may then access some erroneous code and crash even if the library was interface-compatible.

One way to handle multiple versions of libraries is to insert version information into the actual library name as Microsoft does in MFC [9]. For example, names such as MFC40.dll and MFC42.dll can be used for version 4.0 and 4.2. This prevents name collisions problems such as developed in Figure 2. With different names for different version, the situation may be as in Figure 3.

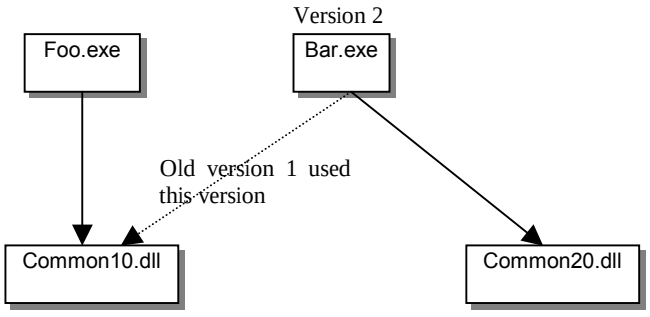


Figure 3. Common10.dll can now coexist with Common20.dll

This solution is to some extent similar to the static linking of executables, because an executable always uses the same version of the shared library. The solution however becomes cumbersome when several versions and variants must be installed in the system. There are, for example MFC42d.dll, MFC42u.dll and MFC42ud.dll which are respectively debug, Unicode and debug/Unicode versions of the MFC library. This tight coupling emerges from the design of the C/C++ compilation model, which was not intended to support independent binary components.

Another way to circumvent the problems is to upgrade all executables dependent on a particular library when the new release is ready. This means that both Foo.exe and Bar.exe will be updated instead of Bar.exe only (Figure 2). This approach can be taken on the assumption that complete control over the whole deployment exists, and from that perspective is very limited.

Suitable support can be achieved with the help of CM functions which keeps track of changes, and by checking which changes are permitted for an executable or a component.

4.2 Interfaces

An interface is a connection between a component and its user. If an interface is changed, the user needs to know that it has been changed and how to use the new version.

Functions exposed to the user are usually designated Application Programmable Interfaces (API). If a change is made in the API, the user must recompile his code. This is the case for compiled languages such as C/C++ but not for interpretative languages such as Smalltalk or Java.

In an object-oriented world, an interface is a set of the public methods defined for an object. Usually the object can be manipulated only through its interface. In C++ the user need only recompile the code when an interface, referred to from the code, is changed.

A disadvantage is that the user of the object must use the same programming language throughout the whole development.

Separation of the interface from the implementation is a means of avoiding this tight coupling. This kind of separation is performed with binary interfaces as in CORBA [3] and COM [6]. Binary interfaces are defined in an interface definition language (IDL) and an IDL compiler, which generates stubs and proxies to make the applications location transparent.

COM solves the interface versioning problem by defining interfaces as unchangeable units. Each time a new version of the interface is created a new interface will be added instead of changing the older version. A basic COM rule is that an interface cannot be

changed when it has been released. This makes couplings between COM components very loose and it is easy to upgrade parts of the system indifferent from each other. Figure 4 shows that it is possible to run new clients together with old server components or vice versa.

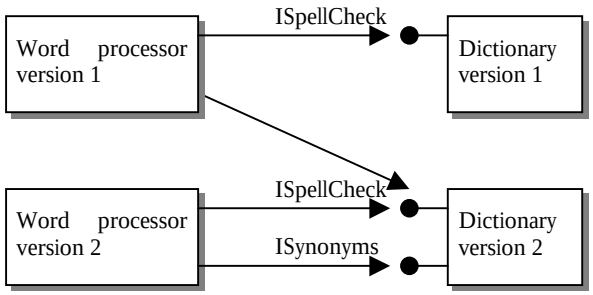


Figure 4. Possible combinations between old and new clients and their server component.

Even if an interface has not been changed, its implementation can be changed. This increases the flexibility of possible updates, but also introduces the possibility of resultant uncontrolled effects. For this reason, it is of interest to know if the implementation has been changed.

Today there is no support for the handling of components in the configuration management perspective. CM functions should provide information about the changes on the interface level.

5 Proposed CM for Libraries and Components

No or insufficient information is available when a system is assembled from components. There is no standard way to track the dependencies between components. When a system is upgraded with a new program, the programs running already might be affected without notice because the new program may introduce new versions of existing components in the system (see Figure 2). It is necessary to determine which interfaces (i.e. components) are used by a program or a component.

As a component is placed in a set of shared libraries some control may be obtained by keeping the libraries under control. We propose a component configuration management on two levels, the library level and the component level.

5.1 CM for libraries

Which shared libraries are linked to another library or program can be seen. This can be used to list the dependencies between different programs and libraries. When installing a new program containing libraries the following steps shall be taken:

1. Take a snapshot of the current system configuration.
2. Install the new modules.
3. Take a snapshot of the new system configuration.

The contents of a snapshot are all programs and libraries installed in the system and are treated as nodes in a graph. A number of different attributes are associated with each library. The information for each node in the graph uniquely identifies the module. We propose that at least date, time, size and name shall be stored. Other attributes are which compatibility change is allowed or if a warning is to be given when a particular module is updated.

A snapshot of the system is presented as a dependency graph. Figure 5 shows an example of how one of the COM libraries depends on other libraries.

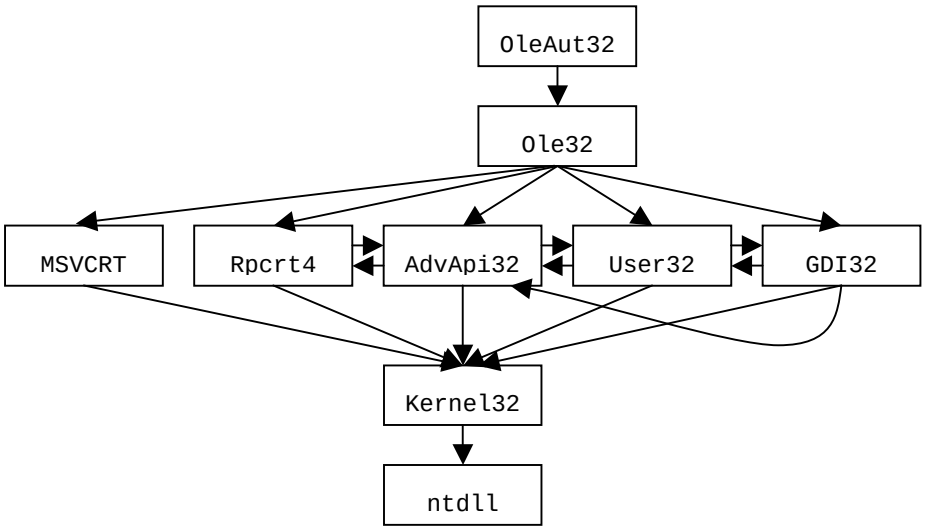


Figure 5. A dependency graph for OleAut32.dll.

Different versions of snapshots are placed under version control and treated as configuration items. A tool which could browse this information would present the differences graphically to the end user. The user would now gain an understanding of the effects of the introduction of new and updated libraries in the system. An alarm would be activated if a library which should not have been affected is changed. The configuration tool could browse different configurations and could label components as changeable or not changable.

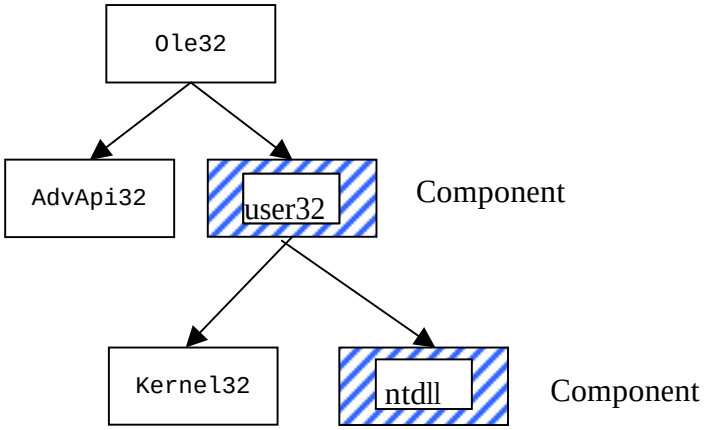


Figure 6. A dependency graph that shows all changed versions.

This kind of knowledge is useful if the cause of malfunction in the system is to be traced. An incorrect version of a library may have been installed by mistake. This kind of identification gives no direct information about which components are changed and which can be affected by the change, but indirect information is available since the physical representation of components are libraries.

5.2 CM for components

In this chapter, we discuss COM as an example. COM treats interfaces in a manner unlike other object models such as CORBA.

COM components expose themselves and communicate through COM interfaces only. Moreover, COM is designed to work with loose references between components. There is no requirement that the clients shall know the class declaration since every class declaration contains implementation details. Components should be able to add or remove interfaces without affecting existing clients.

As components are loosely coupled there is no information connecting different versions of components with each other. A COM component finds its fellow components through the Windows registry in which all installed components store their activation data, such as Interface id, class id, library locations and where to find their stubs and proxies. Connections between components are set up first at run-time. A client uses a unique key to find the server component in the registry and then the COM run-time will load the corresponding component or stub into the client memory.

Unfortunately, there is no capability in the target system for finding which interfaces are used by a component. This prevents us from getting proper information about all dependencies in the system.

If we do not know which components a program uses in run-time, we must request that knowledge. This can be obtained if the provider of the components implements a specific interface for version management, which we designate IVersion (Figure 7). The

`IVersion` interface can return facts about version, name, creation date, compatibility change, interfaces provided and components used. If the components had such an interface, it would be possible to write a tool that could browse and record the dependencies between the components.

```
interface IVersion : IUnkown
{
  HRESULT Name([out , retval] BSTR *name);
  HRESULT Version([out , retval] VERSION *version);
  HRESULT CreationDate([out , retval] DATE *date);
  HRESULT TypeOfChange([out , retval] BSTR *name);
  HRESULT History([in] LONG size,
       [out, size_is(size)] HISTORY history[*]);
  HRESULT HasInterfaces([in] LONG numOfElements,
       [out, size_is(numOfElements)]IID interfaces[*]);
  HRESULT UsesInterfaces([in] LONG numOfElements,
       [out, size_is(numOfElements)] IID
interfaces[*]);
}
```

Figure 7. IDL specification of IVersion.

- `Name`, `Version` and `CreationDate` identifies the component.
- `TypeOfChange` indicates the compatibility level affected by the change.
- `History` informs about previous versions of the component and which type of change applied between them.
- `HasInterfaces` shows all interfaces provided by the component.
- `UsesInterfaces` lists all interfaces used. This list makes possible the building of the dependency tree of the components.

In the absence of a standard version interface, another method is to parse in some way the dependency data from source code files to provide a list of dependencies with the release of a new product. This has some major disadvantages. Firstly, it cannot be applied to third party components. Secondly, it might work for the first level of dependencies where there is source code, but if other third party components are included, no information can be obtained because of the lack of source code.

A possible partial solution to the problem finding dependencies between components is to track the interfaces from the registry repository. All interfaces are registered in the Windows registry with information about where to find the dynamic link library which implements the stubs and proxies for that particular interface. This mechanism provides us with the information we need to see if an interface has been changed during an update. The snapshot browsing tool has a list of all interfaces apart from the libraries and programs installed. The tool can now warn if the implementation of an interface has

been changed. It is possible, using this method, to determine if new interfaces have been registered or if old interfaces have changed implementation.

6 Conclusion

We consider that there is a need for component configuration management, especially during the run-time when components can be changed on the fly. In this paper we have highlighted the different phases in component management in which CM is needed. Support from CM related to component management is rudimentary today and we propose beginning work in a new area, Component Configuration Management.

For want of standardized techniques in component management, we have proposed certain relatively simple methods to identify components and possible changes they can cause in the system. Further work will include a deeper investigation of how to snapshot a system for an insight into the interrelationships between different components. A tool capable of browsing and analyzing an existing system for this should be developed.

7 References

[1] Don Box, Essential COM, Addison-Wesley, ISBN 0-201-63446-5

[2] Alan W. Brown, Kurt C. Wallnau: An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering, 1998 International Workshop on CBSE, http://www.sei.cmu.edu/cbs/icse98/summary.html

[3] Continuus Software Corporation, http://www.continuus.com/homepage.html, 1999

[4] CORBA, http://www.corba.org

[5] Ivica Crnkovic, Magnus Larsson, Managing Standard Components in Large Software Systems, Position paper on Second International Workshop on Component-Based Software Engineering, Los Angeles, May 1999

[6] Microsoft corporation, http://www.microsoft.com/com

[7] Microsoft Source Safe, http://msdn.microsoft.com/ssafe

[8] Rational http://www.rational.com/products/clearcase/index.jtmpl, 1999

[9] Dale Rogerson, Inside COM, Microsoft Press, ISBN 1-57231-349-8

[10] Andreas Zeller, Versioning System Models Through Description Logic, Proceedings ECOOP'98 SCM-8 Symposium, vol 1439 of Lecture Notes in Computer Science, Springer-Verlag.

COMPONENT CONFIGURATION MANAGEMENT

Magnus Larsson & Ivica Crnkovic

Abstract:

Component-based programming is now a widely recognized approach in software development. There remain many open problems related to both technical and non-technical aspects of the components. In this paper, we point out the problem of component identification. Since the components are usually binary units deployed in the system at run-time, we do not have the same insight into their characteristics as into those of the software units which we manage at development time. This problem could be solved if the components had this information integrated together with the binary code, which can be achieve by defining a standardized identification interface. As such interfaces do not exist in standard component models today, this concept can only be used with components built in-house. For external components, extensive tests can, to some extent, compensate for the lack of information. To perform a successful testing efficiently we must limit the number of test cases. Which parts of our system can be affected by the introduction of a component, or by its updating? We can answer this if we can keep track of changes introduced in the system and their impact on the system. These problems are similar to the problems arising at development-time solved by Software Configuration Management (SCM) disciplines. In this paper we point out these problems and make proposals for their solutions at run-time using SCM principles.

1 Introduction

When developing a component independently of system development, we meet a number of problems due to the fact that information we usually have during the component development process is not available. One type of problem is related to the components themselves – the component interface, pre- and post-conditions and the nonfunctional component characteristics such as reliability, resource requirements, timing requirements, etc. Another type of problem is associated with the relationship between the component and the rest of the system. In this paper we address this second type of problem.

When integrated in a system, the new component has an impact on a part of the system. The new component may refer to certain components, and it can also be used by other components. In addition to these explicitly defined dependencies, we also have indirect dependencies, derived from the components which are used by the new component.

Finally, we have implicit dependencies, which are related to the system environment (for example timing or other resource constraints). In general, we can expect that some parts of the system are not affected by a change when introducing a new component or a new component version. This situation is shown in Figure 1.

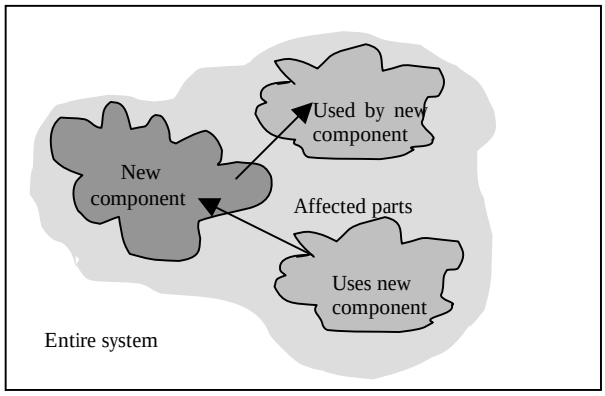


Figure 1. Dependencies between components

The dependencies are not directly visible in component models available today, such as COM [2] or EJB [3].

To limit the uncertainty of the system behavior, we must identify those parts of the system which might be affected by the introduction of a component.

If we could identify the component versions explicitly, we could specify the entire system as a set of component versions. Two systems, or two versions of one system can be compared and differences on the component level can be identified.

If we could automatically identify the dependencies between components and their versions we could avoid the well-known problem with different versions of shared libraries. The problem is illustrated in Figure 2: We have two programs *Foo.exe* and *Bar.exe*, which share a *Common.dll* library, version v_1. We then upgrade Bar.exe, obtaining a new version of *Common.dll*, v_2. The replacement could be successful if version v_2 of *Common.dll* is compatible with version v_1, but if this is not the case the *Foo.exe* program can fail. Even if the new version is interface-compatible, *Common.dll* may contain undetected errors, which appear in a combination with *Foo.exe*. *Foo.exe* may then import some erroneous code and crash, even if the library was interface-compatible.

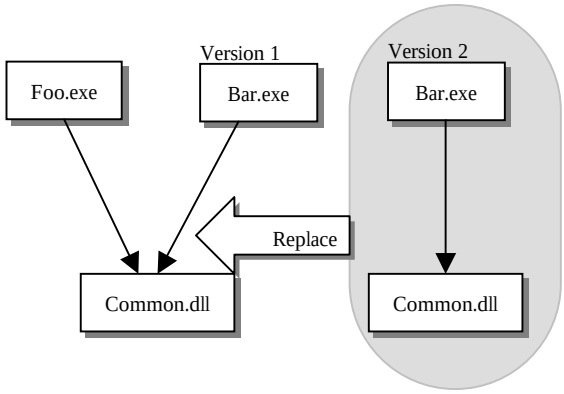


Figure 2. Uncontrolled update of a component

One way to handle multiple versions of libraries is to insert version information into the actual library name as Microsoft does in MFC [1]. For example, names such as *MFC40.dll* and *MFC42.dll* can be used for version 4.0 and 4.2. This prevents name collision problems but can introduce a vast number of versions over which we have no control.

To identify the parts of the system which can be affected by the change we must:

- Identify components including their versions.
- Identify direct and indirect dependencies.
- Obtain sufficient information to localize the implicit dependencies.

Identification and dependency management is a typical subject of SCM. The SCM disciplines and their possible implementations for managing components are discussed in section 2. In section 3 we discuss the problem with dependency information which is missing from component models available today. In its absence, we discuss a possibility of finding dependencies directly from the code. A Dependency Browser, an application which displays dependencies between binary assets, is depicted in section 4. Finally, section 5 outlines further investigations.

2 Component Management and SCM

As a component is a unit of composition, its management is natural related to Software Configuration Management, the main objective of which is to manage composite entities. However, most of the SCM functions are used at development-time, and are not utilized sufficiently at run-time[5]. The major disciplines of SCM are *Version Management*, *Configuration Management* and *Change Management* [7] [8], and we discuss their use for managing components.

Version management performs the identification of entities and recognizes different versions of enities. We can apply this principle to the components at run-time: Every component in the system should be identified by a name, version number and other

version attributes such as creation date, historical information, etc. We need the component version identification for two reasons: Firstly, when we update a component with a new version, we want to be able to identify that change. Secondly, in some cases, we wish to keep several component versions integrated in the system. Managing different versions of components is important for middle-size or large systems. A component might not have been originally designed to cover all the system requirements which evolve. In general, it is better to release a component containing currently required features and to upgrade it later, instead of releasing a fully-fledged component too late. Later, when new features are added to the component, it may happen that the new component version is not compatible with the previous, or that is not fully tested. In that case we want to keep both versions - the new one exploiting new features, and the old one, to be used by those parts of the system we had not yet changed or tested. When the system must support this type of environment, and when several versions of components are used at the same time, the development time and maintenance increase. Experience shows however that this type of evolution is appropriate for large systems [4].

Configuration and build management methods are used to select and identify specific versions of entities (i.e. to generate a *baseline* or a *configuration*) and to integrate them into a new version of the composite entity. It also includes build procedures. The building procedures use information about the dependencies between the entities. These principles can be applied in the run-time system: A system configuration is defined as a set of component versions. By adding a new component or a new version of a component, a new configuration of the system is identified. Similar to *Make* dependencies, which describe the dependencies at build-time, a component should include the specification of the components used (the references to the components used actually exist in the component, but they are hidden in the binary code).

Change management provides information about the changes introduced in the system on an abstract level, the logical changes which have been introduced in the process, rather then a physical. Change management becomes important when a new entity version is created. In a similar way, every component version can include information about the differences between it and the previous version. This information cannot be automatically generated (which is possible for other type of information, such as version identification and version attributes), and it must be explicitly defined by the component developers. A new component version might be added to introduce new functions in a system, or only to change its behaviour, (better performance, better stability), without changing the interface. When replacing a component or a component version we must consider which type of change we permit and which system characteristics we want to preserve, in order to guarantee the system behaviour.

To describe this possible impact on the system, we have defined three levels of compatibility:

Input and Output compatibility. A component requires input in a specific format (or perhaps no input at all) and produces results in a defined format. The internal characteristics of the component are of no interest.

Interface compatibility. The interface remains the same, but the implementation can be different

Behavior compatibility. Internal characteristics of the components, such as performance, resource requirements, must be preserved.

The compatibility criteria can be used to decide if a component can be replaced or not. This decision can be especially important in case of a replacement "on the fly" in a run-time environment.

3 Managing Component Dependencies

Binary components are delivered as shared libraries and executables, which usually have no additional information about any dependency between components. To be able to predict what will happen in a system when a component is installed we need to have information about which part of the system will be affected by the component.

As components can be loosely coupled there is no information connecting different versions of components with each other. In COM for example, a component finds components it refers to through the Windows registry. In the Window registry all installed components store their activation data, such as Interface id, class id, library locations and where to find their stubs and proxies. Connections between components are set up first at run-time. A client uses a unique key to find the server component in the registry and then the COM run-time will load the corresponding component or stub into the client memory [2].

To be able to get full dependency graphs over the system with coherent information about all the components, and the type of change introduced in a component, we need meta-data. By meta-data we mean additional information which is not crucial for the component to run but is valuable for the entire system. Meta-data can be provided as a new interface on the component [5] or stored in a repository where it have been placed during the component registration process. Facts about version, name, creation date, compatibility change, interfaces provided and components used, as mentioned in the previous section, are examples of meta-data which can be of help when building a system with consistent configuration management.

The World Wide Web Consortium has defined a standard to describe components and their dependencies. This language is XML-based and is called Open Software Description (OSD) [6]. However, OSD is mainly designed for web components and does not solve the problems with component dependencies. It is important that meta-data is accessible to third part users. A common standard making it possible to describe components in all component models is probably a utopia. We can expect that different

types of components will be described in different ways, which is vastly better than their not being described at all.

As we do not have meta-data incorporated in the standard component models, the only information about the components we can get through binary libraries and executables. The information about which shared libraries are linked to other libraries or programs can be gathered fairly easy. In general this information is linked into the binary code and can be extracted. This information can be used to list the dependencies between different programs and libraries.

The following is the formal procedure: A component version c is implemented as a library or an executable. A component version has a set of attributes (name, size, creation-date, and others [11] used in different component models), by which it is identified.

The set of all components installed in the system is designated *S*. We define a relation $\rightarrow$ called "depends on", where $c_i \rightarrow c_j$ if the correct operation of c_i requires the correct operation of c_j. This relation is transitive which means that we can derive all indirect dependencies from the direct dependencies.

The set of all dependencies is defined as

$$D = \{(c_i,c_j) : c_i,c_j \in S \wedge c_i \rightarrow c_j\}.$$

The dependency set *D* is stored as a baseline before new components are installed. A snapshot of the current configuration is the set of all components and their dependencies:

$$C = (S,D).$$

When new versions of existing components or new components are installed they will affect the configuration

$$C' = (S',D')$$

We identify all component versions which are placed in only one configuration

$$c_{new} \in S_\Delta \; : S_\Delta = \overline{S \cap S'},$$

and the dependencies D_Δ of components c_{new}

$$D_\Delta = \{c : c \rightarrow c_{new}\}.$$

All components in S_Δ and dependencies in D_Δ can change the behavior of the system and are subjects for futher investigation.

For the dependencies where new components use other components

$$D'_{\Delta} = \{ c_{new} : c_{new} \rightarrow c \}$$

we test if the input-output domain (i.e. expected outputs from c_{new} for inputs to c) have been preserved or not. If a new range of input to the component c occurs, this dependency should be tested in this new domain range [9].

If a system configuration can contain several component versions, specified ranges in input/output domain can be compared with the current values and used as criteria for selecting a component version to be executed [10][11].

4 Dependency Browser

To show how dependencies can be traced, we have designed an application for Windows NT 4.0, Dependency Browser which parses through the system, finds all shared libraries and generates the dependency graph. A snapshot of the current configuration can be shown and saved in a repository. Different versions of snapshots are placed under version control and treated as configuration items. The current configuration, or an earlier snapshot, can be compared with other configuration snapshots, and the differences between the configurations can be displayed. Typically, before installation of a component, a snapshot can be saved. The component is then installed, and a new snapshot can be taken. The difference graph shows which components have been changed and their relations to other parts of the system. The browser can show the entire system, or a specific component and its dependencies, which makes it possible to see a potential consequence of a component update. System integrators can use the dependency browser to view dependencies in the test system, when a new component has been integrated in the system.

All components which depend on the changed component are highlighted and the user can decide and take action upon this information as shown in Figure 3. The dependency browser helps the integrator of the system to verify that nothing unexpected occurs when the system starts. With this tool, it is possible to see all the files affected when a component has been updated or installed.

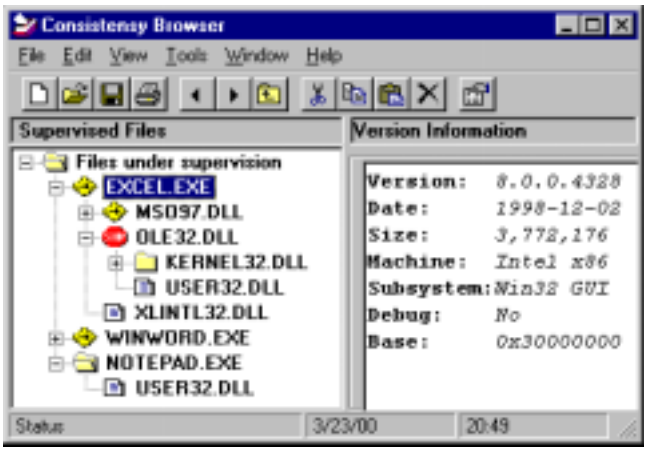


Figure 3. Affected components are highlighted in the browser to alert the user.

The changed or updated components have the stop sign icon while affected components are marked with an arrow icon. Version information of the component is presented in the right pane view. The browser can be used to browse through the information and to get an understanding of the effects of the introduction of new and updated components in the system. The tool can browse through different configurations and label components as changeable or not changeable. This kind of knowledge is useful if the cause of malfunction in the system is to be traced. An incorrect version of a library may have been installed by mistake and without dependency information it is difficult to find the real cause of the problem.

5 Conclusion

In this paper we have pointed out the problems encountered during the dynamic configuration of systems. Our contribution is a proposal for component configuration management in which components can be placed under version control. We tie together software configuration management (SCM) and component-oriented programming (COP) with ideas from both disciplines. A simple dependency model is presented and we have shown how to solve the dependency problem for this model when new components are installed. We plan to do more work on a formal description and management of dependencies.

Future work will include the realization of the Dependency Browser, its implementation for different component models and platforms. In this paper we have treated components as binary entities, i.e. executables or shared libraries. How dependencies between loosely coupled components can be recorded, will be studied in a more thorough investigation. The goal is to have ability to predict the behavior of a system before a system update.

6 References

[1] D. Rogerson, Inside COM, Microsoft Press, ISBN 1-57231-349-8

[2] D. Box, Essential COM, Addison-Wesley, ISBN 0-201-63446-5

[3] E. Roman, Mastering EJB, Wiley, ISBN 0-471-33229-1

[4] M. Larsson, I. Crnkovic, Development Experiences of a Component-Based System, 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)

[5] M. Larsson, I. Crnkovic, New Challenges for Configuration Management, System Configuration Management, SCM-9, Springer 1999, ISBN 3-540-66484-X

[6] W3C, Open Software Description Format, http://www.w3.org/TR/NOTE-OSD.html

[7] R. Conradi and B. Westfechtel, Version Models for Software Configuration Management, Software Configuration Management Symposium, SCM-7, 1977, Springer, ISBN 3-540-63014-7, ACM Computing Surveys, Vol. 30, No.2, June 1998

[8] J. Estublier, S. Dami, M. Amiour, Hifg Level Process Modeling for SCM Systems,

[9] H. Thane, A. Wall, Formal and Probabilistic Arguments for component Reuse in Safety-Critical Real-Time Systems, Technical report CBSE – State of the Art, Mälardalen University, 2000

[10] J. E. Cook, J. A.Dage, Highly Reliable Upgrading of Components, 21st ICSE, 1999, ACM ISBN 1-58113-074-0

[11] Henrik Lykke Nielsen, René Elmström, Proposal for Tools Supporting Component-based programming, Workshop on Component-based Programming, 1999

COMPONENT CONFIGURATION MANAGEMENT FOR FRAMEWORKS

Ivica Crnkovic, Magnus Larsson & Kung-Kiu Lau

Abstract:

Object-oriented Design frameworks are increasingly recognized as better components than objects. In this paper, we briefly explain the framework concept, show a COM implementation, and discuss the accompanying configuration management issues.

Keywords: Components, configuration management, frameworks, COM, objects, CBD, CBSE

1 Introduction

Object-oriented Design (OOD) frameworks are increasingly recognized as better components in software development than objects (see e.g. [4] and [7]). The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and OOD frameworks can capture this, whereas existing OOD methods (e.g. Fusion [1] and Syntropy [2]) cannot. The latter use classes or objects as the basic unit of design or reuse, and are based on the traditional view of an object, as shown in Figure 1, which regards an object as a closed entity with one fixed role.

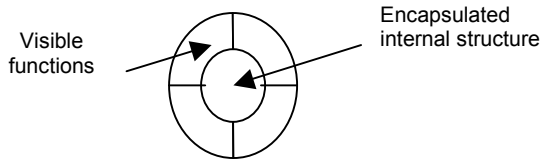


Figure 1. Objects with one fixed role.

On the other hand, frameworks allow objects that play different roles in different frameworks to be composed by composing frameworks. In *Catalysis* [3], for instance, this is depicted in Figure 2.

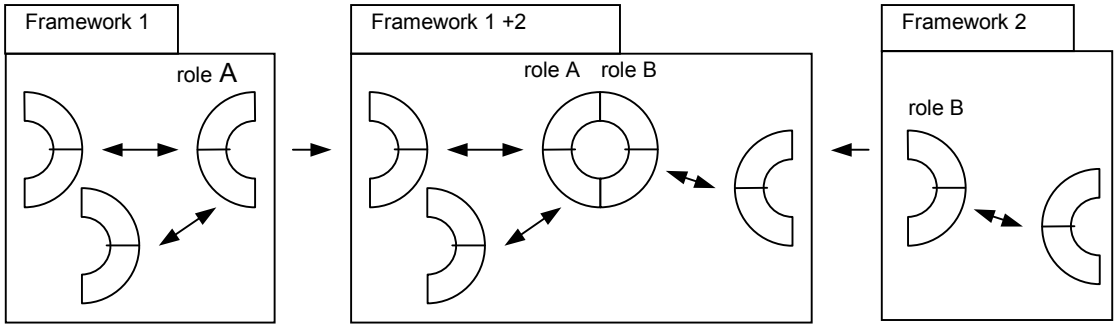


Figure 2. Objects with multiple roles in different frameworks.

In this paper we discuss a possible COM implementation of framework and the accompanying configuration management (CM) issues.

2 Frameworks: An Example

The following example illustrates the framework concept. Consider the framework for employees as depicted in Figure 3, in which a person plays the role of an employee of a company.

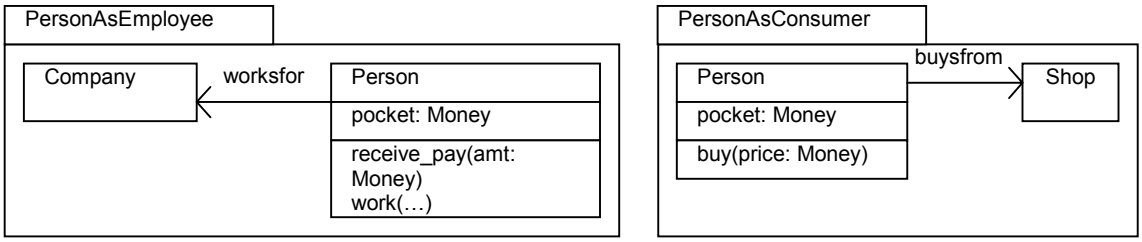


Figure 3. PersonAsEmployee and PersonAsConsumer framework.

A person as an employee has an attribute *pocket* representing the amount of money he possesses, and two actions *receive_pay* and *work*. Now consider another view of a person, e.g., a person plays the role of a consumer, as shown in the PersonAsConsumer framework in Figure 3. In this role a person also has the attribute *pocket*, but he has the action *buy* (instead of the actions *receive_pay* and *work*). We may compose the frameworks for PersonAsEmployee and PersonAsConsumer, to obtain a person with both roles together. A person now has all the actions of both roles, namely *receive_pay*, *work* and *buy*, and the attribute *pocket* in both roles. The composition is illustrated by Figure 4.

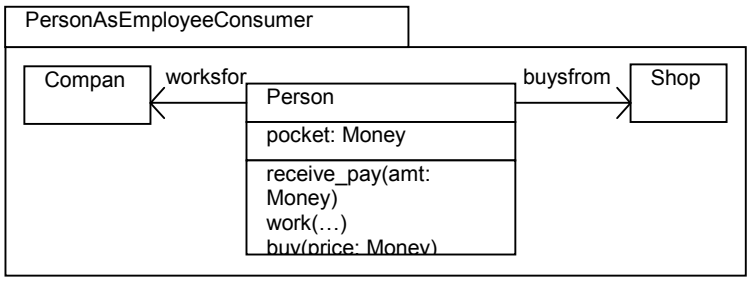


Figure 4. PersonAsEmployeeConsumer framework.

3 A COM Implementation of Frameworks

We illustrate the frameworks in Figure 3 and Figure 4 with an implementation example using COM [8]. COM suits multiple roles because it can use multiple interfaces for each role. We will use the aggregation mechanism in COM to compose frameworks. First, we implement the *Person* object, which corresponds to the encapsulated internal structure in Figure 1. The *Person* object is constructed so it supports aggregation of role objects and it has one *IPerson* interface (see Figure 5).

Figure 5. A COM object for the person object and the consumer role.

Second, the consumer and employee roles are implemented so they support being aggregated into a person object. Figure 5 shows the consumer role with one *IConsumer* interface. The consumer object needs also a reference to the person object to be able to work on the *pocket* variable. The person reference is set up when the consumer is aggregated into the person object (see Figure 6). In a similar way the *employee* role is implemented. Using aggregation we can reuse the different components that we have created. Figure 6 shows how *Person* aggregates the two already defined COM objects. Frameworks are created at run-time by adding roles to an object.

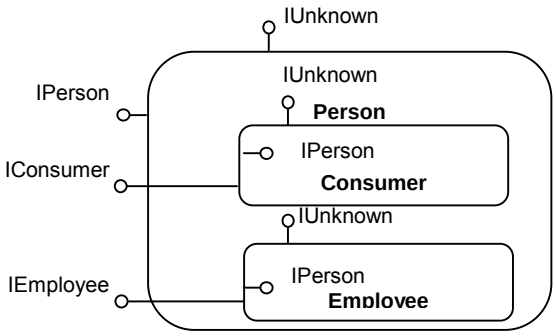


Figure 6. The Consumer and Employee roles are aggregated into the Person object.

The COM implementation of the framework concept has some limitations. The COM model defines frameworks as aggregates of the completed objects created at run-time, while a general framework model allows us to use incomplete objects (at run-time) or classes (at build-time).

4 Configuration Management Issues

Using frameworks instead of pure objects gives several advantages, but it also introduces an additional level of complexity when building them. Frameworks are composite types of entities – they have an internal structure which is built from objects, or from parts of them. A framework entity also has relations to other frameworks, and can be composed from other (sub)frameworks. The definition and creation of such a composite entity introduces configuration problems. Some of them will be illustrated here for a COM implementation.

Let us consider the following examples:

- Sharing objects in several frameworks;
- Composing frameworks from objects and frameworks.

4.1 Sharing objects in several frameworks

Suppose framework F_1 includes objects O_1 and O_2 with a relation R_{12} between them, and framework F_2 contains objects O_1 and O_3 with a relation R_{13}. The object O_1 is shared by two frameworks:

$$F_1 = \{O_1\ O_2\ ;\ R_{12}\},\quad F_2 = \{O_1\ O_3\ ;\ R_{13}\} \qquad (1)$$

Suppose we now add a new property to the object O_1, a property that is required in (an improved version of) framework F_2. This creates a new version of the object $O_{1;v2}$, (v2 denotes the new version) which is included into the framework F_2:

$$F_2 = \{O_{1;v2}\ O_3\ ;R_{13}\} \quad (2)$$

However, if we do not take versioning into consideration, then the framework specifications will remain the same. In this case, we can be aware of the change of the object O_1 in the context of framework F_2, but not necessary in that of F_1. Our specification of F_1 is defined by (1), but in reality we have

$$F_1 = \{O_{1;v2}\ O_2\ ;R_{12}\} \quad (3)$$

If the role of the object $O_{1;v2}$ used in F_1 is changed, then the behaviour of F_1 will be changed unpredictably, and a system using F_1 can fail.

To avoid these unpredictable situations we can introduce basic configuration management methods – a version management of objects and configuration of frameworks [9]:

- An object is identified by its name and version.
- A framework is identified by a name and a version. A new framework version is derived from object versions included in the framework.

These rules imply that new versions of frameworks will be configured when a new object version is created, as shown in our example:

$$F_{1;vi}=\{O_{1;vm}\ O_{2;vn}\ ;\ R_{12}\},\ F_{2;vk}=\{O_{1;vm}\ O_{3;vk}\ ;\ R_{13}\} \quad (4)$$

$$F_{1;vi+1}=\{O_{1;vm+1}\ O_{2;vn}\ ;\ R_{12}\},\ F_{2;vk+1}=\{O_{1;vm+1}\ O_{3;vk}\ ;\ R_{13}\} \quad (5)$$

As several frameworks can share one object, and a framework can contain several objects, the number of generated frameworks can grow explosively. It is, however, possible to limit the number of interesting configurations. Typically, in a development process, we would implement the changes on all the objects we want, collect the versions of objects we want in a *baseline* and derive the frameworks from the baselined object versions. In such a case, experience for similar cases [10] shows that the number of derived entities does not necessarily grow rapidly.

A shared object is not necessarily completely shared, but different parts of the object, defined by the object's roles, are used in the frameworks. In the COM implementation a complete object will be included, but a part of it will be used. In a general framework model, a class (or an object at run-time) includes only those parts which are specified in the object's role.

When we define a new role for an object in a framework or re-define the existing one, we need to change a specific part of the object class. We call this specific part an object *aspect*. The change of an object aspect will affect only those frameworks where the aspect is included. Other frameworks, though containing the object (or part of it), are not affected by the change. In this case, it is better to keep version control on the aspect level, and relate a framework configuration to the object aspects.

If we declare an aspect as a subset of an object $A_i(O_k) \subseteq O_k$, then an object version is defined as a set of aspect versions:

$$O_{i;vk} = \{ A_{j;vl} \} \quad (6)$$

and a framework version is defined as a set of aspect versions with relations between the aspects:

$$F_{vk} = \{ \{A_{j;vl}(O_{i;vk})\} ; R_{jl} \} \quad (7)$$

Having control over changes on the aspect level, we can gain control over the changes on the framework level. Now we can more precisely identify the frameworks being affected by changes in object roles.

4.2 Composing frameworks from objects and frameworks

In the framework model it is possible to compose new frameworks from existing frameworks. A new framework is a superset of the classes and relations from the frameworks involved. If a new framework is created at run-time, as in a COM implementation, then the objects from the selected frameworks comprise the new framework.

The following example illustrates the merging process of two frameworks F_1 and F_2 into F_3:

$$F_1 = \{O_1\ O_2 ; R_{12} \}, \quad F_2 = \{O_1\ O_3 ; R_{13} \}, \quad F_3 = \{O_1\ O_2\ O_3 ; R_{12}, R_{13}, R_{23}\} \quad (8)$$

The composition works fine as long as we do not need to consider the changes of objects within one framework.

Suppose we create a new object version (or a new object aspect version) in F_2 and keep the old version of the same object in F_1:

$$F_{1;vi} = \{O_{1;v1}\ O_{2;vk} ; R_{12} \}, \quad F_{2;vj} = \{O_{1;v1+1}\ O_{3;vl} ; R_{13} \} \quad (9)$$

In the merging process we have to recognise if different versions of the same objects are included in the frameworks being merged. If that is the case, we have two possible solutions:

- Selecting one specific version of the object (for example the latest):

$$F_{3;v1} = \{ O_{1;v1+1}\ O_{2;vk}\ O_{3;v1} ; R_{12}, R_{13}, R_{23}\} \quad (10)$$

- Selecting both versions and enable their consistence in the new framework:

$$F_{3;v1} = \{ O_{1;v1} , O_{1;v1+1} , O_{2;vk}\ O_{3;v1}; R_{12} , R_{13}, R_{23} \} \quad (11)$$

For the second case there must be support for identifying object versions. This support can be provided by introducing an identification interface [12] as the standard interface of an object. There must also be support for managing different versions of the same object in the running system.

5 Discussion

The framework approach gives a better possibility to reuse components in composing systems. A discussion on formal description of frameworks can be found in [5] and [6]. In this paper we have presented a possible implementation of the framework model using the COM technology. This implementation shows some limitations. Further investigation on how to improve the implementation should be carried out.

The second topic discussed in this paper is configuration management for frameworks. The paper emphasises a need for using CM methods for managing frameworks as composite objects. The CM issues are complicated and need further investigations: Questions of managing relations, concurrent versions of frameworks, inclusion of change management [11], etc., must be addressed. Since aspects and objects are not entities recognised as CM-items by standard CM tools (which recognise entities such as files, directories, etc.), new, semantic-based rules must be incorporated into the CM tools. For different OO and component-technologies, different tools have to be made. How different do they need to be, and are there possibilities to define common rules and implementation? These are questions for future investigation.

6 References

[1] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes: Object-Oriented Development: The Fusion Method, Prentice-Hall, 1994.

[2] S. Cook and J. Daniels: Designing Object Systems, Prentice-Hall, 1994.

[3] D.F. D'Souza and A.C. Wills: Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.

[4] R. Helm, I.M. Holland, and D. Gangopadhay: Contracts - Specifying behavioral compositions in OO systems, Sigplan Notices 25(10) (Proc. ECOOP/OOPSLA 90).

[5] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills: Interacting frameworks in Catalysis. In J. Staples, M. Hinchey and S. Liu, editors, Proc.Second IEEE Int. Conf. on Formal Engineering Methods, pages 110-119, IEEE Computer Society Press, 1998.

[6] K.-K. Lau, M. Ornaghi, and A. Wills: Frameworks in catalysis: Pictorial notation and formal semantics, In M. Hinchey and S. Liu, editors, Proc. 1st IEEE Int. Conf. on Formal Engineering Methods, pages 213-220, IEEE Computer Society Press, 1997.

[7] R. Mauth: A better foundation: development frameworks let you build an application with reusable objects. BYTE 21(9):40IS 10-13, September 1996.

[8] D. Box, Essential COM: Addison-Wesely, 1998

[9] R. Conradi, B. Westfechtel: Version Models for Software Configuration Management, ACM Computing Surveys, Vol. 30, No.2, June 1998

[10] U. Asklund, L. Bendix, H.B. Cristensen, B. Magnusson: The Unified Extensional Versioning Model, System Configuration Managemnt SCM-9, Springer, 1999

[11] I. Crnkovic: Experience with Change Oriented SCM Tools, Software Configuration Management SCM-7, Springer, 1997

[12] M. Larsson, I. Crnkovic: New Challenges for Configuration Management, System Configuration Management SCM-9, Springer, 1999

INDEX

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Licentiate theses from the Department of Information Technology

2000-001	Katarina Boman:	*Low-Angle Estimation: Models, Methods and Bounds*
2000-002	Susanne Remle:	*Modeling and Parameter Estimation of the Diffusion Equation*
2000-003	Fredrik Larsson:	*Efficient Implementation of Model-Checkers for Networks of Timed Automata*
2000-004	Anders Wall:	*A Formal Approach to Analysis of Software Architectures for Real-Time Systems*
2000-005	Fredrik Edelvik:	*Finite Volume Solvers for the Maxwell Equations in Time Domain*
2000-006	Gustaf Naeser:	*A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

UPPSALA
UNIVERSITY