Oskarsson, Osten

MECHANISMS OF MODIFIABILITY IN LARGE SOFTWARE SYSTEMS

Linkoping University (Sweden) PH.D. 1982

Linköping Studies in Science and Technology. Dissertations.
No. 77

MECHANISMS OF MODIFIABILITY IN LARGE SOFTWARE SYSTEMS

By
Östen Oskarsson

Software Systems Research Center
Linköping University, S-581 83 Linköping, Sweden

Linköping 1982

ISBN 91-7372-527-7

Printed in Sweden by VTT-Grafiska, Vimmerby 1982

Linköping Studies in Science and Technology. Dissertations.
No. 77

MECHANISMS OF MODIFIABILITY IN LARGE SOFTWARE SYSTEMS

Akademisk avhandling

som för avläggande av teknisk doktorsexamen vid Tekniska Högskolan i Linköping kommer att offentligt försvaras i sal C3, Universitetet i Linköping, onsdag den 19 maj 1982 klockan 13.15.

By
Östen Oskarsson

Abstract

Large software systems are often characterized by a continuing evolution, where a large number of people are involved in maintaining and extending the system. Software modifiability is a critical issue in such system evolution. It is desirable that the basic design is modifiable, and that subsequent evolution maintains this modifiability. This thesis is an investigation of the mechanisms behind the exhibited modifiability and lack of modifiability in a large commercial software system during a part of its evolution.

First, the relation between modifiability and different types of modularizations are discussed, and a dichotomy of software modularizations is proposed. As a measure of modifiability at system level, i.e. disregarding the internal modifiability of modules, we use the number of modules which are influenced by the implementation of a certain system change. The implementation of each requirement in one release of the system is examined, and the underlying causes of good and bad modifiability are explained. This results in a list of factors which were found to influence the modifiability.

Software Systems Research Center
Linköping University, S-581 83 Linköping, Sweden

Acknowledgements

Many friends and colleagues have supported and encouraged me during the work with this thesis, both at LM Ericsson and at the SSRC group in Linköping, and I am very grateful to all of them. There are some which I would like to mention especially.

The topic of this thesis was suggested by Göran Sundelöf at LM Ericsson, who has made many helpful suggestions. Eva Salomonsson has been responsible for the project. My reference group at LM Ericsson, which kept me on the right track in the case study, contained Margareta Josefsson, Erik Örnulf, Erik Nordenberg and Ivar Jacobsson. Johan Törnström has patiently answered my questions and explained to me the basic facts around AXE.

At Linköping University, I am indebted first of all to Lennart Jonesjö and Lars Wikstrand, who have read and reread the manuscript and suggested important improvements. I also owe my thanks to my thesis advisor, Erik Sandewall, and all other members of the SSRC group, who have read and discussed the manuscript.

I would also like to thank Margareta Löfgren and Malt Terlegård, who have been very helpful typing and retyping early versions of the manuscript, Lillemor Wallgren and Lena Östling who have mastered the administrative side of my dissertation, and Anthony Skeat who has proof read my English.

This work was sponsored by Telefonaktiebolaget LM Ericsson.

Contents

Chapter 1

INTRODUCTION

1.0 BACKGROUND

To construct a large software system for one specific application only is usually not economically feasible. An ideal is a system with a long life-time and availability, successively being adapted to new needs of users, new hardware, environmental changes and new applications, and where new software technology can be introduced. Furthermore, it is tempting to construct software systems so that they will cover a large class of applications, for example process control. Controlling automatic trucks in an iron mine is a different application from controlling a paper mill process. Yet these applications are of the same type, and have much in common, so process control systems intended to cover both applications may be wanted.

Thus, there is a strong economic pressure for flexibility in large software systems. Flexibility can be achieved either by making the software general (e.g. through parameterization), or by making it modifiable. From a point of view of competition, the use of general software has limitations because of the extra costs of memory and processing capacity for customers with small needs. In real-time applications the need for fast responses may also rule out the general software approach. Generic software and conditional compilers are used to avoid these disadvantages to some extent, but these techniques are not always feasible, e.g. because their administration adds a large extra overhead to system complexity. Specializing general programs automatically by partial evaluation [BEC76] is not yet of practical interest. Furthermore, it is impossible to prepare general software for all possible new requirements, so modifications will always be necessary. There is also a tendency for marketing departments to stretch the area of application beyond what was originally intended, which leads to unanticipated changes.

So, modification is an important activity in handling large software systems which are intended for large areas of application and many different environments. Modification, that is, apart from normal maintenance activities such as error correction and optimization. This continuing development and adaptation may be termed ***software evolution*** [BEL79, GOR80] or ***program evolution*** [LEH76]. Sandewall [SAN78] uses the term ***structured growth*** in a similar meaning. Examples of modifications are extension of functional capabilities, removal of functional capabilities, adaptation to new processor hardware and adaptation to new environments. In the following, we will use "maintenance" to mean error correction and optimization, and "evolution" to mean a continuing development.

There are several approaches to the problem of software evolution in the literature. Viewing software as ***program families*** (Parnas [PAR76, PAR79]), means an emphasis on what is common between the different applications, rather than on what is specific for each. An important part of Parnas' philosophy is the search for the minimal subset of capabilities that will perform a useful service. Probably there is no application for just this minimal subset of capabilities, but it defines something that is common to all intended applications. The maximum flexibility is then achieved by extending the subset with the smallest possible increments in capability. However, Parnas thus requires that the intended applications be defined at outset. This means that the main aim is not to cope with unanticipated requirements for change.

A practical method to achieve flexibility in large systems is to create libraries of ***software components*** [BEL77, GOR80, MCI72, SOR74]. The system then consists of a core that is delivered to all customers, and a set of optional modules. By choosing among the optional modules and by creating new modules, the delivered system can be tailored for the customer's needs.

This thesis is intended to study evidence from a practical case as to what actually constituted modifiability and lack of modifiability when a large software system was subject to a continuing evolution. The case study concerns one release of a large commercial telephone switching system, *AXE*, designed and produced by Telefonaktiebolaget LM Ericsson. The software of this system consists of more than one million lines of high-level program code.

2.0 OBJECTIVES

The purpose of this thesis is to find and describe some of the intrinsic mechanisms of system modifiability and lack of modifiability in a class of large software systems. The term "system modifiability" refers to ease of change through addition of modules or exchange of modules, as opposed to "program modifiability", meaning ease of change within modules, at the level of source program statements. "Exchange of modules" also includes cases where the new modules are not completely new, but rather modified versions of the old ones.

The goal is *not* to provide a method for structuring systems to achieve modifiability, but the goal is to provide information about the problem. This is done by reviewing literature concerning software modifiability and software structure, and by describing different factors in an existing system, which have influenced system modifiability. The possible relations between different factors are described. Often, avoiding one type of problem will cause another.

Conclusions are supported by references to literature, examples from a case study, and by common-sense reasoning. No formalism is involved.

Maintenance activities (error corrections, optimizations) are not studied. It was assumed that the factors which affect system evolution are not necessarily the same as those affecting maintenance. The activities concerning system evolution could be studied alone in the case study, since maintenance and evolution of AXE software are handled and documented separately. Studies of maintenance activities have been reported by e.g. Weiss [WEI81].

Only one aspect of software modifiability is discussed in the case study, namely modifiability at system level. For each requirement for a change, the modifiability is judged as the number of influenced modules in relation to the complexity of the requirement. This is a coarse measure, but just this coarseness ought to make it easy to agree about.

3.0 MOTIVATION

The need for modifiability in software systems is well established. Examples of reasons for this need are:

1. Large software systems will have to be changed because their environments and their applications will change during the systems' life-times, even if the software systems were intended for only one application and one environment each from the beginning.

2. The pressure for cost-effective software leads to design of software systems intended for large classes of applications and for a variety of environments. However, it is neither possible to foresee nor reasonable to implement all the relevant applications and environments when basic design is performed. Usually, the aim is to make a modifiable system which can be adapted for new applications and environments when needed.

It is the author's opinion that the problem of making software systems modifiable is not very well understood at present. On the lowest level, the discussion about structured programming has led to valuable understanding of how to structure program code within modules so as to make the modules easy to change. On the level of system structure, research has not yet reached this far.

This kind of problem is treated by some authors, e.g. Parnas [PAR72, PAR76, PAR79], Goodenough et al [GOO74], Belady and Lehman [BEL79, LEH76], Stevens et al [STE74]. Usually, these authors treat the whole subject of software design and software structure without special stress on modifiability. Except for Belady and Lehman, they treat small systems only, and when modifiability is discussed, none of the authors goes into any detail.

Belady and Lehman [BEL79] describe largeness of a software system to be not so much a question of size as the number of people involved in the specification, design, testing, maintenance and operation of the system. In this thesis we will adopt this view as being the most important where modifiability is concerned.

This thesis is based on the work of the abovementioned authors and focuses on a more specific problem: modifiability at system level in large resource-oriented systems. A major part of the thesis is the case study of the modifiability of AXE software. Common-sense reasoning and toy examples are important tools in software engineering research because of the difficulties of experimenting with large systems. However, much experience has been gained in the work on large commercial systems. Case studies of such authentic real-world system evolution is important, since the support of such tasks is an important aspect of software engineering research. Also, it is difficult to judge modifiability without a case study, since in reality, the result depends both on properties of the system, and on what actually happened to it.

4.0 METHOD OF WORK

One might describe the stated goal of this research to be to find out from a practical case what has been there the major "mechanisms" of modifiability. It is tempting to start by setting up strict definitions and exact measures. For example, we might give a strict definition of the concept of modifiability and of its constituents, and exact measures of complexity, extensibility, etc.

This temptation has been resisted in this thesis. No formal definitions are given. Rather, explanations are given in an informal way, and we have avoided defining the exact dividing line between different concepts. When measures are needed, only coarse and obvious measures are used.

Such an ambition in scientific work needs some explanation and motivation. There are two main reasons for adopting the described attitude, the state-of-the-art of software engineering research, and the form of the data for the case study. Also the undeveloped state of the application terminology for computer controlled telephone switching has influenced us.

Compared with many other research areas, software engineering is in its infancy, and at the same time it is a highly complex discipline of human creativity in connection with complex artificial systems. There is no common terminology for e.g. software modifiability, and there is very little of firm theoretical basis. This situation is not caused by some extraordinary laziness of software engineering researchers, making them neglect basic problems. The truth seems to be that we know too little yet about our problems to be able to find out e.g. exactly what terms are needed and how to give them strict definitions. For example, a formal proof that one type of structure is more easy to understand than an other seems impossible to obtain, taking into account our current knowledge of systems and humans.

In this situation it was felt that a bold attempt to be formal and strict would only mislead the reader and obscure the practical experiences and general lessons which the thesis is intended to convey. Our attitude is similar to (and inspired by) Belady and Lehman: "That is, the initial development of any science is phenomenology-based. It is not in the first place built, as is mathematics, on abstract concepts, axioms, that are gradually developed into a total structure of models that pass tests of reasonableness and elegance. A formal framework and axiomatic theory follow when basics are clear, when it is known what is fundamental or critical, and what is fortuitous." [BEL79]

The other reason is the state of the input data to the case study. This information mainly consists of statements in English written by a large number of people with different background, and by oral information from interviews. To force this information into a narrow formal model would not add anything, but only remove information. Instead an informal model of functions and interfaces was found sufficient.

The methodology of this research can be described as consisting of four steps:

1. A study of existing experience and knowledge about system structures from the vantage point of modifiability.
2. A measure of system modifiability is presented and motivated.
3. To all modifications in one release of a specific system (AXE), the measure is applied, and occurrences of good and bad results are explained.

4. The underlying causes of good and bad measures are generalized.

5.0 OVERVIEW OF CONTENTS

Chapter 2 is an overview of literature relevant to the study of the relations between software structure and modifiability. A tentative terminology of software modifiability is proposed and discussed. Two different modularization principles from literature are described, and their contributions to the modifiability qualities of systems are discussed.

In chapter three, the concept of system level modifiability is introduced and discussed. A measure of system level modifiability is proposed and motivated. The case study is described as a set of experiments, where the reasons for good and bad modifiability measures are explained.

Chapter 4 describes a case study where the design and implementation of one release of a large commercial software system, the software of the AXE telephone switching system, is described and analyzed. The AXE release concerned adaptations for 7 different markets. The number of new or changed high-level program statements of the release amounts to about 20 000. The analysis results in a list of factors which supported the implementation of changes in the existing system structure, and factors which made changes complicated, seen at system level. These factors are discussed and explained.

Chapter 2

MODULARIZATION AND SOFTWARE SYSTEM MODIFIABILITY

1.0 INTRODUCTION

Most authors agree that one key to software modifiability is modularization. Ideally, a software system should consist of a set of well isolated modules of a suitable size, and it should be possible to implement any individual new requirement by simply changing a module or adding a new module to be connected in a prepared way. The reason for this wish is quite obvious. The human ability to understand how to implement a change depends very much on how easily one can find the information needed. If a change can be implemented in one module, this means that one can find most of the information needed there. In large systems where changes in different parts have to be performed by different persons, this is still more important. A successful modularization is one way to achieve this. Two kinds of changes are to be facilitated:

1. Prepared changes, where the designer has strived to simplify foreseeable changes. This practice of preparing for future change involves various risks:

   When the change is finally needed, the requirement may be different to what was originally imagined, so the preparation may have been in vain.

Unprepared changes of the system cannot be avoided, and when the prepared change is finally to be effectuated, an unprepared change may very well have invalidated the preparation.

The preparation may never be needed. In this case it is an unnecessary burden, making the system unnecessarily complex.

2. The second kind is unprepared changes, which are simplified if the modularization is done so that the modules themselves are easy to understand and change, and there is little "coupling" between modules [STE74]. Also, it is the present author's view that proper regard at system design for the application area, can support unprepared changes.

Basically, a software module is a collection of source program statements. The further criteria used to distinguish software modules depend on the purpose of the concept. In the literature there are many views of what constitutes a module, see e.g. [CHE77, DEN77, EDW75, FEA73, GOO73, HOL75, TUR80, PAR72]. Most of those views make assumptions about basic modularization principles. Since we want to cover different modularization principles, those views are not sufficient. Also, in contrast to those authors we are going to discuss modularization in relation to modifiability only. The following criteria seem to be a suitable choice from suggestions in literature, to use for our purposes.

1. Modules are disjoint and can communicate with each other.

2. Modules can be compiled, tested and produced separately.

3. Modules are responsibility assignments [PAR72]; it is the responsibility of one person or one group to design and evolve one module. This means that module size must be limited.

4. Modules are independent. The internals of a module can be designed without knowledge of the implementation of its environment.

These criteria rule out such source statement collections as Data Division in Cobol and

Block Data subprograms in Fortran. Examples of modules fulfilling our criteria are subroutines and Simula classes.

This chapter is intended to discuss different modularization techniques found in literature, and focuses on their ability to support a continuing evolution of large systems. Section 2 describes four main activities of software evolution. The efforts required to perform these activities are discussed. Section 3 lists some system properties which support the evolution efforts. Section 4 identifies two classes of modularization. In sections 5-6 these two classes are described in more detail. In section 7 the modularization principles are compared regarding modifiability, and it is argued that the two classes of modularization are to some extent complementary regarding system modifiability. Different ways of combining the two classes are discussed.

2.0 SOFTWARE EVOLUTION ACTIVITIES

Evolution of an existing software system may be discussed in terms of five main activities:

1. ***Extension of capabilities.*** This includes both the extension of an existing capability and the introduction of completely new capabilities. It is not always possible to classify a requirement exactly into one of these categories since the extension of an old capability can be achieved by introducing a new, minor capability. Yau and Collofello [YAU80] use the expression "enhancement of capabilities", which also includes the activity of adaptation to new environments.

2. ***Deletion of unwanted capabilities.*** It is not always worthwhile to remove extraneous parts of a software system. In many cases the costs in memory size and execution time are minor compared to the cost of removing the extra program code, especially when dealing with capabilities whose realizations are woven in throughout the system. Systems within the scope of this thesis, however, can be expected to accumulate obsolete or seldom-needed capabilities since they are subject to repeated modifications. For a commercial software manufacturer offering products to many different customers, it is usually undesirable if his products require unnecessary resources. Also, such "leftover" capabilities tend

to increase the complexity of systems, thus making debugging and further evolution more difficult.

In general it is more difficult to remove the implementation of a capability than to introduce a new one. When introducing a new capability, the designer can choose to connect it to the system in the simplest possible way. When removing the realization of an unwanted capability, there is no such choice. [TEN81]

3. ***Adaptations to environments.*** Long-life evolving software systems are influenced by environmental changes in two ways.

   New areas of application will include new environments to which adaptation must take place. Examples are personnel with new needs, and new types of external hardware equipment.

   Technological and social development will affect existing systems. New hardware devices are connected to existing software systems, new laws and new demands from labour unions will change the system's interaction with its environment. An example is new security requirements. This kind of environmental change will affect both systems in operation and new deliveries. It is not always clear if a new requirement concerns adaptation to the environment or extension of capability. An adaptation may very well be realized through a new capability.

4. ***Adaptation to new processor hardware.*** If a software system is to live and be delivered for twenty years, most likely the last systems which are delivered will be implemented on hardware that differs from the first implementations. Also in a shorter perspective, many types of systems are moved to different computers. This concerns the problem of software portability.

   The difference between points 3 and 4 is not always distinct. For example, to an operating system a tape driver may be part of the environment, while for an application program it may be part of the processor hardware.

5. ***Cleaning-up.*** A large part of the evolution effort has to be invested in keeping the handling qualities of the system at an acceptable level. These activities are

usually not directly visible for the users, and often they are performed for the vendor's own sake.

To be able to discuss ways to simplify the task of evolving existing systems we need a picture of the efforts required for modification of software. References [GOO74, YAU80] address this subject. The main tasks may be categorized thus:

1. ***Understanding where to change.*** The person responsible for realization of a certain requirement for change (presumably not the person who designed the software originally), must find out where, and in what way, the concerned capabilities are implemented. He must understand the implementation well enough to be able to devise a modification of the software that will effect the required change of the system.

2. ***Analyzing the ripple effects.*** Because of dependencies between different parts of the software, the devised change may introduce inconsistencies in dependent software. The corrections to these inconsistencies may create new inconsistencies, and corrections can thus spread as ripple effects through the system. Dependencies between software units must be understood so that such ripple effects can be tracked down. [YAU80]

3. ***Performing the software modifications.*** The main effort required when changing software to achieve a specified change of the system is not the manual task of editing source-code. It is more difficult to manage the updating process (especially if several persons are involved), to understand what one is doing, and to perform and document it in a correct way. A complicated task can be very difficult in this way, even if it is completely well-defined.

4. ***Testing system behaviour.*** We must test both that the required changes in system behaviour have been effected, and that all other aspects of the system are left unchanged. In an evolving system, it should be easy to test capabilities that were not to be changed, since old test data can be used. Reference [EDW75] is concerned with the employment of modularity for the sake of testability.