M. Sc. Thesis

A model for optimisation of software development policies in the presence of evolutionary requirements

By J.A.R. Noppen

noppen@cs.utwente.nl

Enschede, March 2002

Trese group (Twente Research on Education and Software Engineering)
SE (Software Engineering)
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Graduation Committee:
Prof. Dr. Ir. M. Aksit
Ir. M. Glandrup
Dr. Ir. V. Nicola
Dr. Ir. B. Tekinerdogan

As all the living forms of life are the lineal descendants of those which lived long before the Silurian epoch, we may feel certain that the ordinary succession by generation has never once been broken, and that no cataclysm has desolated the whole world. Hence we may look with some confidence to a secure future of equally inappreciable length. And as natural selection works solely by and for the good of each being, all corporeal and mental endowments will tend to progress towards perfection.

Charles Darwin, the origin of species

Abstract

Nowadays companies and people rely on software systems more than ever before. Software is applied in almost every area imaginable, from a simple calculator to a very complex operating system for nuclear facilities. The application of software systems in a great variety of complex environments causes the systems themselves to become more complex. Despite the introduction of methods, CASE environments and languages, problems still occur when the systems need to be upgraded or modified to fulfil new customer requirements.

This process of customer requirements occurring after a software system has been delivered is called *software evolution*. The requirements that were applicable for the first system have evolved and therefore demand an evolving system. Because these new requirements generally are not analysed during the design process they can influence the way in which the existing system should be modified to supply the desired functionality. This process can become very complex and increase costs tremendously. Current methods hardly address these problems even though the maintenance activities can cause costs that are the major part of the available budget for a software project.

This thesis tries to address these problems by identifying possible changes at the point when the initial system is designed. By conducting a thorough problem domain analysis the possible extensions on a specific software system are identified. The different functional parts (from a user point-of-view) are then related to each other to be able to determine the impact of individual parts on the entire system. The evolution process is modelled as a probabilistic process, where costs can occur when new requirements are introduced. By doing this the uncertain character of evolution scenarios can be modelled accordingly.

The analysis model defines a model for relations between customer requirements from which it is possible to see the impact of adding a requirement will have on the entire system. Based on this impact model techniques and structures can be identified that simplify the implementation of these future requirements. The analysis model can calculate expected costs for choosing specific techniques to prepare a software system for the future requirements. This is done by relating cost-models to the techniques that can be chosen, and probability-models to the requirements that can occur.

Samenvatting

Tegenwoordig zijn bedrijven en mensen afhankelijker van software systemen dan ooit tevoren. Software is aanwezig in bijna elk denkbaar gebied, variërend van een simpele rekenmachine tot een complex besturingssysteem voor kerncentrales. Omdat software wordt ingezet in een grote variëteit van complexe omgevingen worden de systemen zelf ook steeds complexer. En hoewel er methodieken, CASE omgevingen and talen geïntroduceerd zijn blijven er zich problemen voordoen wanneer systemen moeten worden bijgewerkt of aangepast om aan nieuwe eisen van de klant te voldoen.

Dit proces van nieuwe eisen die voorkomen nadat een software systeem is opgeleverd staat bekend als *software evolution*. Het eisenpakket dat van toepassing was op het initiële systeem is geëvolueerd en dit wordt nu ook van het software systeem gevraagd. Omdat deze nieuwe eisen over het algemeen niet worden geanalyseerd in het ontwerpproces kunnen ze een onverwachte invloed hebben op de manier waarop het bestaande systeem moet worden aangepast. Dit proces kan zeer complex worden en de kosten kunnen zeer hoog worden. De huidige methodieken en processen schenken nauwelijks aandacht aan deze problemen, terwijl de onderhoudsactiviteiten het overgrote deel van het beschikbare budget kunnen opslokken.

Deze these probeert deze problemen af te vangen door het identificeren van mogelijke wijzigingen op het moment dat het initiële systeem wordt ontworpen. Door het uitvoeren van een gedegen domein-analyse worden de mogelijke wijzigingen gevonden. De verschillende functionele onderdelen worden vervolgens met elkaar gerelateerd om zo de impact van individuele wijzigingen op het gehele systeem te kunnen inschatten. Het evolutie proces wordt gemodelleerd als een probabilistisch proces, waar kosten worden veroorzaakt door het introduceren van nieuwe eisen. Hierdoor kan het onzekere karakter van evolutie scenario's accuraat worden gemodelleerd.

Het analyse model definieert een model waarmee relaties tussen eisen van de klant in kaart kunnen worden gebracht zodat de impact van wijzigingen kan worden ingeschat. Technieken die de implementatie van dergelijke wijzigingen kunnen vereenvoudigd kunnen worden gevonden op basis van dit relatie-model. Het analyse-model kan vervolgens de te verwachten kosten uitrekenen wanneer bepaalde technieken worden gekozen om een software systeem voor te bereiden. Dit gebeurt op basis van het relateren van kosten-modellen aan de desbetreffende technieken en de kansmodellen voor het voorkomen van nieuwe eisen van de klant.

Table of Contents

Preface

Dedication

I would like to dedicate this work to my parents

Theo and José Noppen

for giving me this possibility and I especially dedicate this work to my uncle who sadly has not been able to see me complete this work:

Ben Wubbels

From March 2001 until March 2002 I have worked on this graduation project, which has been a though year for me, but also a very fulfilling one. During the project I have learned a lot about doing research, applying knowledge and myself.

At the start of each chapter a quotation is made. These are taken from *The origin of Species* by Charles Darwin [Darwin]. This book is one of the most famous books on evolution. Although there is no direct connection between the evolution of organisms and software evolution, parts of the book sometimes come close to the problems associated with software evolution.

Acknowledgements

First I would like to thank my family for their support, their interest, and putting up with me during the course of this project. I would also like to thank my friends for keeping me going during the difficult times. I especially want to mention Mark van Benthem, Frank Vlaardingerbroek and Dennis Wagelaar for their valuable feedback, and Lotte Nijkamp, Luitzen van der Sluis and Jeroen Spit for their moral and emotional support.

I'd like to thank the members of my graduation committee, Mehmet Aksit, Maurice Glandrup, Victor Nicola and Bedir Tekinerdogan for their help; I would like to thank Michael Moran for the valuable feedback on the research and the concept version of this thesis and everybody I have not mentioned.

1. Introduction

Nothing at first can appear more difficult to believe than that the more complex organs and instincts should have been perfected not by means superior to, though analogous with, human reason, but by the accumulation of innumerable slight variations, each good for the individual possessor.

Charles Darwin, the origin of species

1 Introduction

An introduction to the M. Sc. Project

This M. Sc. Thesis describes the design of a method with which it is possible to analyse and prepare for evolution aspects in software systems. This work has been carried out for the TRESE group (Twente Research on Education and Software Engineering).

In this first chapter, a closer look will be taken at the problems of software evolution. The outline of the thesis will be described. Also a short description will be given of the context of the work, the Trese group and the field of software engineering.

1.1. The Trese group

The TRESE group (Twente Research on Education and Software Engineering) is the popular name of the Software Engineering chair of the University of Twente and its main goal is to perform research in the area of software engineering. This research covers all aspects of the area, but focuses in particular on compositional object technology. The aim is to provide models, methods, tools and frameworks for the creation and maintenance of adaptable software.

One of the goals of TRESE is to research models and methods that lead to higher quality software. By making the quality of software solutions an integral part of the design process the results will become better suited to customer needs and software will become more long lived. This is known as *Quality-oriented Software Engineering*. Eventually this should lead to high quality software design methods for designing high-quality software.

1.2. The problems of software evolution

In this part a closer look will be taken at what evolution actually means for software systems and how they are affected by it. First the main cause of evolution will be identified followed by “logical” actions to take. Then a closer look will be taken as to why these logical steps do not provide logical solutions.

1.2.1. Software systems and customer requirements

Software systems are designed based on the requirements introduced by the customer. The requirements define the way in which the system should behave when a user interacts with it. Once the software system is completed, the customer should be able to use it for a considerable time without the need for change.

This almost never is the case. After a certain amount of time the customer will have new or changed requirements on the system, which means the system should be modified to meet

these new requirements. This principle is called *software evolution*. Two different causes for evolving requirements can be identified:

The first cause is that software systems most of the time will not be used in static environments. As environments change, so will the requirements on the software system that is used in the environment. Functionality that was asked for initially can become irrelevant or incomplete, and therefore needs to be adapted to fit the new environment.

The second cause is that the customer does not always exactly know what is needed at the time the requirements are defined. During the design process and when the final product has been installed the customer realises certain aspects should be handled in a different way. The requirements on which the system has been based will be changed because of this.

These changes to the existing requirements can lead to high costs since the initial requirements have been used to design the software system. When software evolution occurs it is possible that the design choices that have been made, need to be re-evaluated because they might not be applicable in the context of the new requirements. This could mean that the system needs to be redesigned from the start, depending on the impact of the new requirements on the existing system.

1.2.2. Complex requirements changes

The cause of software evolution is clearly the modification of existing requirements and the addition of new ones. But why do these changes cause problems? Customer requirements are the base for the design of the software system. The requirements cover a wide range of properties of the software system, varying from functionality to performance issues. When the software system is designed, all these requirements are taken into account which means that the design will be optimised for the specific requirements.

When the requirements change at a certain point, this means the design is no longer optimised for the current requirements. The system needs to be modified to fulfil the evolved requirements. Depending on the type of change that is done to the existing requirements the upgrade of the software system can vary from very difficult to fairly easy.

Requirements changes that affect small parts of the system are not necessarily difficult to incorporate into the software system. Small additions or the removal of independent parts of functionality can be seen as simple requirements changes. When it is possible to modify the system by changing a small part, the impact of the evolution is fairly small. For example the addition of a new language to a multi-language spell-checker will not be much more expensive when it is done after the system has been used, than including it from the earliest design. This is because the system already is capable of working with multiple languages (assuming the correction of spelling of words is done based on dictionaries).

The costs of upgrading a system because of evolving requirements will increase when the requirements changes are more complex. It is possible that the costs of modifying the system will exceed the costs of redesigning the system from the start. Complex requirements changes affect several parts of the system at the same time. The system parts that are affected by the requirements changes need to be modified, but these changes might also make it necessary to modify other parts of the systems. The evolved requirements can demand system changes that cascade through the entire system. For instance when for an automated form-analyser the customer asks for a system that can analyse forms with more information, this has impact on several system parts. The system should be able to present the new form to the user, but also the new form should be analysed, the storage format needs to be changed, etc..

1.2.3. Handling software evolution

What possibilities are there to address the problem of software evolution? Because the requirements changes that will be introduced, are not known when the system is designed it is difficult to include them in the design. Even when possible changes can be identified, it is not sure whether they should be included because they will not occur with certainty.

The consequences of changing requirements can be addressed in two different ways (although they do not necessarily exclude each other): maintenance after the completion and preparation of the system for the possible requirement changes. When the second method is applied upgrading afterwards will most likely still be needed, but this will easier because the problems that can occur have already been addressed in part. This means the upgrade will be less expensive. The problems that can occur when the first approach is used have been described above, but the second method doesn't provide fail-safe solutions either.

When the software system should be prepared for possible future changes many different issues should be addressed before a usable product can be designed. First of all, how can the relevant possible changes to requirements be found? And when they have been found, their impact on the system needs to be assessed to identify the critical changes. Once this has been done the possible ways of preparing the system need to be found. To make a reasonable decision on how the system should be prepared for future changes an analysis should be done of the possible changes, the probability that they will occur and ways of preparing the system. This should for instance be done with respect to cost. Preparing the system should not be more expensive than upgrading the system afterwards with no preparation.

Software evolution is a process with several possible outcomes and numerous scenarios that can occur. If these scenarios would have been a certainty they would have been included in the initial requirements, but this is not the case. This means that preparing for possible evolution is an investment that might never pay off. This unpredictability should also be included in the analyses because for instance investing money in preparation of a highly unlikely scenario seems just as unnecessary as not investing at all.

In popular software design processes there are hardly any facilities to perform these types of analyses which makes it difficult to design a software system with the capabilities of dealing with software evolution. And since the budget normally is limited it is not possible to design a software system that will support every possible extension (when this is even possible).

1.2.4. Summary

Evolution of software systems caused by changing customer requirements can be a very tedious and labour-intensive task when the new requirements are difficult to implement in to the existing system. Because software systems are designed based on the initial system requirements, new requirements can affect several parts of the system at the same time. In the worst case it might be necessary to re-implement the entire software system.

Upgrading the system when the new requirements occur can become very expensive but preparing the system for the possible changes is difficult because in current methods there are hardly any facilities. The way in which the system is prepared will depend on the experience of previous projects, but will not be based on analysis results. This thesis will try to define an model with which it is possible to analyse the different ways of preparing a system for identified possible future changes. The results of this model can be used as a validation for including preparations in to the system at a time they are not explicitly needed.

1.3. Structure of the thesis

This thesis consists of three parts: an in-depth study of evolution and evolution aspects, the actual analysis method for evolution aspects in software systems, and the conclusions. In chapter 2 the problems in current processes will be analysed. Also a closer look will be taken at earlier work on software evolution problems. In chapter 3 the approach and model proposal will be described.

The analysis method will be presented in chapter 4. First an impact analysis method is described, that allows the user to gain an insight into how evolution aspects will affect existing software systems. The second part describes how different options for system implementations affect future efforts, how they can be found and assessed. This part is covered in Chapter 5.

The final part of the method will be about a decision-model that is able to determine what the best options are for system implementation based on probability-models. This will be described in 6.

In chapter 7 the analysis model will be applied to an example case. From this example case it becomes clear how the model can be used to find optimal development policies and how to define specific parts of the analysis.

In the final part conclusions will be made about the efficiency and usefulness of the analysis method. This will be done for the example case as well as in a general manner. Furthermore recommendations will be done for future extensions of the analysis model. This will be described in chapter 8.

2. Background and related work

Hence it seems to me, as it has to many other naturalists, that the view of each species having been produced in one area alone, and having subsequently migrated from that area as far as its powers of migration and subsistence under past and present conditions permitted, is the most probable.

Charles Darwin, the origin of species

2 Background and related work

A background study of current problems and solutions

In this chapter several popular software design processes will be analysed to see how the problems of software evolution are addressed. By studying other efforts in the field of software engineering a proper insight is attained on the current state of the art.

2.1. Introduction

Evolution of requirements for software systems is not new, but because the systems that are designed are becoming more complex these problems will become a bigger issue. It is difficult to analyse the problems that might occur, if and when requirements change after a certain amount of time.

To better understand the problems that might occur with software evolution two different software design processes will be analysed in this chapter. Also a closer look is taken at previous efforts on solving problems of software evolution. The information from this analysis will be used as a base for defining a model for analysing software evolution.

2.2. Evolution management in current processes

2.2.1. Rational unified process

The Rational Unified Process is a popular method that is defined by Jacobson, Rumbough and Booch for the Rational Company [Jacobs1999]. Because this method is fairly popular, it is good to see how this method copes with the difficulties presented by evolving software requirements.

The Unified Process is “use-case driven, iterative and architecture-centric”. This means that for requirements modelling a use-case approach is adopted and the method is aimed at finding a good architecture for the system to be made. The entire method can repeat itself, by applying use-case models on the found architecture. When this is done this leads to a better understanding of the system-complexity and leads to a better system architecture.

The iterative character of the unified process is able to work with changing customer requirements during the design of the software system. By applying the use-cases to the defined system architecture the customer can get an early insight into how the system will operate when it is done. For the customer it is possible to evaluate this behaviour and make adjustments to the requirements when necessary.

When requirements change after the system has been completed, it is more difficult to modify the existing functionality. The system architecture has been optimised for the use-cases that were relevant and known at the time of the design. The changes to the requirements force

changes on the use-case model that was defined and this could render the existing architecture less applicable or even invalid. This can be depicted in the following manner:

The initial customer requirements are used to define the use-case model for the initial system design. This design leads to a system architecture on which the uses-cases will be applied. The feedback will be used to refine the architecture design and eventually the final product will be delivered.

When new requirements occur, the use-case model also changes. The changes might be additions but also constraints or any other type of change. The new use-cases most of the time can not be applied to the existing architecture. The architecture needs to be modified to be able to handle the new use-case model. And as was stated before, the modifications might cascade through the entire design.

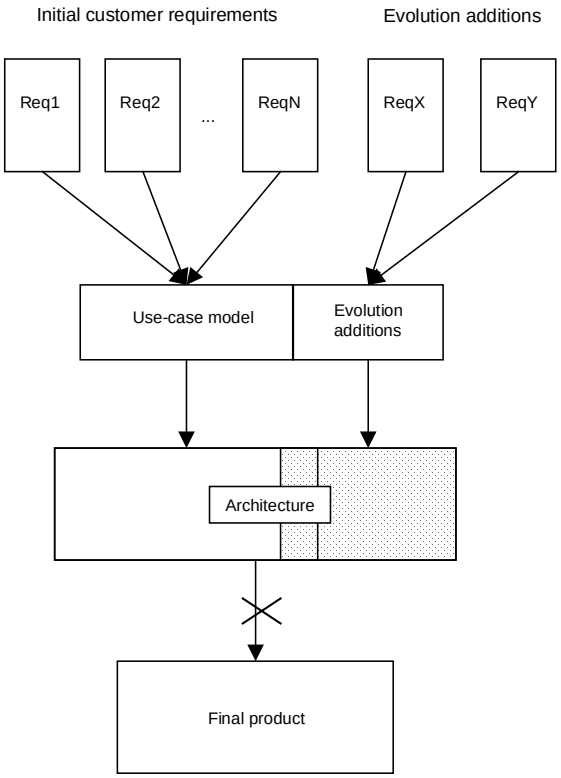


Figure 1: Evolution problems in RUP

The cause of these problems is that in the unified process there are no facilities of including uncertain use-cases in the system design. Either a use-case applies or it doesn't. To prepare a system for future changes it should be possible to include these scenarios and their effects into the design. At this point it is only possible to include these aspects in the design based on experience, and not based on analysis.

2.2.2. Product lines

Product Lines are a different approach to a specific type of problem in the software engineering process, defined by the Software Engineering Institute [Carnegie2001]. When several products should be made, that share a part of the system functionality a product line can be made. In this product line the product base supplies the common functionality and each individual product adds functionality to the product base.

For this purpose a problem domain is scoped, which means that the boundaries are defined for what is relevant for the architecture design. The problem domain should cover all the products that are relevant to the product line, because the problem domain will be analysed for variability and commonality of the possible products. Based on this analysis the product base will be designed.

Once the problem domain has been properly scoped a problem domain analysis is performed which identifies the possible products for the scoped domain. All requirements for affected products are gathered and from these requirements the requirements for the product base are made. This product base will be designed in such a way that it is easy to reuse for all products.

When the product base is ready, individual products can be designed and use the product base for common functionality.

This approach seems to be better equipped to deal with evolution since a highly reusable part is designed and when it is possible to find the commonalties for evolution scenarios the problem seems to be solved. The problem, once again, is the lack of facilities to analyse uncertain evolution scenarios. Because these scenarios have not been included in the analysis of commonality and variability of the problem domain products it is possible that the product base might become invalid and can't serve as a base for other evolution scenarios. To facilitate a proper product base for the evolution scenarios as well as the initial requirements, it might be necessary to re-scope the problem domain.

2.3. Relevant efforts in the field of software evolution

Although the problem of software evolution is by no means new, the amount of research that has been done is not very large. Most relevant researches are based on problem domain modelling and variability analysis. This paragraph will examine several efforts that have been made in the field of software engineering that address problems that relate to software evolution.

2.3.1. Variability analysis and domain modelling

The key to solving software evolution problems is knowing how software can vary before the design is done. For this a proper insight is needed into the system and its domain.

Several analysis techniques have been defined for this purpose two of which will be described in greater detail below. Domain modelling is aimed at analysing the domain in which a software system exists. A domain is an area that covers all relevant aspects for a certain (group of) system(s). In [Czar2000] a domain is defined as:

Domain: An area of knowledge

- *Scoped to maximise the satisfaction of the requirements of its stakeholders*
- *Including a set of concepts and terminology understood by practitioners in that area and,*
- *Including knowledge of how to build software systems (or parts of the software system) in that area.*

When a domain is analysed for a certain set of systems, a better insight is given into a specific system but also into its possible alternatives and extensions. Variability can be analysed and functional reusable parts from the domain can be used to cope with these problems.

Both techniques described below are domain analysis techniques, which means that they try to capture the knowledge about a domain. With this knowledge it is possible to identify common requirements for a certain domain and design reusable components for these parts. Furthermore will a better understanding of the domain lead to a better insight into the design of the system.

2.3.2. Feature oriented domain analysis

Feature oriented domain analysis (FODA) is a method for domain engineering that is developed by Carnegie Mellon University and is described in [Kang1990]. FODA is aimed at domain engineering and can be divided into two phases:

- Context analysis: This part defines the boundaries of the domain
- Domain modelling: the goal of this phase is to produce a domain model

The context analysis phase is aimed at finding the scope of a domain that leads to relevant results. Relations between this domain and candidate domains and elements are identified and their variability evaluated. The results of the context analysis phase are used to scope the domain.

The domain-modelling phase focuses on the commonalties and differences of the problems in the domain that was found in the context analysis phase. The results of this phase are several models that represent different aspects of these problems.

The domain-modelling phase is divided into three different parts: feature-analysis, entity-relationship modelling and functional analysis. For the thesis the part of feature-analysis will be described in more detail since this part is able to capture customer demands and relate them to each other. This can be used for a better understanding of evolution problems since new customer demands occur and they should be related to existing requirements.

The feature analysis focuses on capabilities that a system should posses that are relevant to a customer. These capabilities might include:

- Services provided by the application
- Performance offered by the application
- Etc.

These capabilities are commonly referred to as features. To be able to work with this concept of features, features are defined as *"attributes of a system that directly affect the end user"* [Kang1990].

The part that is particular interest is the feature-analysis model. This model provides the means to relate customer requirements (features) to each other. This way the model has its use towards both the customer and the engineer. The customer understands why certain features can only be asked at certain times, and the engineer can asses the complexity of implementing features without specifically needing to define computational functions that might confuse the customer.

2.3.3. Organisational domain modelling

Organisational domain modelling was developed by Mark Simmons of Synquiry Ltd. (formerly Organon Motives Inc.). ODM dates back to the work on knowledge-based reuse support environment by Simons and several projects over the years have refined the process. During this period ODM acquired several ideas from other domain engineering approaches and from non-software disciplines. A complete documentation on ODM can be found in [SCK1996].

Typical aspects of ODM include:

- *Focus on stakeholders and settings*
- *Types of domains (horizontal vs. vertical, etc.)*
- *A more general notion of features as opposed to FODA*
- *Analysis of feature combinations*
- *Flexible architecture*
- *Tailorable process*

The ODM process can be divided into three main phases:

1. *Plan domain*: This part corresponds to the Context Analysis phase of FODA and is primarily aimed at domain scoping and planning
2. *Model domain*: This phase corresponds to the domain modelling phase in FODA and should produce a domain model
3. *Engineer Asset Base*: In this phase the architecture for the system is produced and the reusable assets are implemented.

ODM is aimed at finding reusable assets , and doe this in a very detailed way. Every phase yields its own deliverable and this is used in the subsequent phase.

This leads to an asset base where several reusable assets can be found for implementation of systems that qualify for the domain. As with FODA a feature model is used but the feature model is more refined and more expressive.

2.4. Summary

The problem of evolution is mainly caused by the fact that requirements occur after a system has been completed and delivered. Although there is not necessarily a problem when a system should be extended, difficulties are likely to occur because the new requirements are almost certain to relate to existing requirements. And it is this complexity that causes system maintenance costs to rise above expectation.

Current methods hardly address evolution (some in part, others not at all) leaving the finished system vulnerable to future evolution changes. Previous efforts in this field stress the importance of requirements relations, so this seems the logical place to analyse and look for a methodology for working with future complexity. These methods are primarily aimed at finding reusable components that are common in the system's domain.

These models rather aim at domains than specific systems, in order to understand system complexity. This is good from an evolutionary point of view, since a system that has evolved will most likely be in the same domain. The scoping becomes very important for this type of problem, because when the scoping is done wrongly, this means that evolving requirements might force the system outside the scoped domain

The goal of the thesis will be to define an analysis model with which it is possible to predict which of the possible requirements are likely to occur, and prepare to system to meet these demands more easily. The thesis is not aimed at offering assistance for the design of an entire software system but rather at identifying possible extensions and the best way to prepare for these new requirements.

3. An approach to the analysis of software evolution

As on the theory of natural selection an interminable number of intermediate forms must have existed, linking together all the species in each group by gradations as fine as our present varieties, it may be asked, Why do we not see these linking forms all around us? Why are not all organic beings blended together in an inextricable chaos?

Charles Darwin, the origin of species

3 An approach to the analysis of software evolution

How to analyse software evolution problems

In this chapter a closer look is taken at evolution and an approach for addressing the problems. How should evolution be represented for analysis purposes? And what will a software evolution analysis model have to do to assist the software engineer in coping with the problems?

3.1. Introduction

Evolution is normally defined as a gradual process, where capabilities of organisms are changed in order to cope with a changing environment. This evolution is needed, because changes in the environment force new requirements upon the organisms. And when these new requirements are not met, the species might become extinct. It is taken for granted, however, that organisms are capable of adapting to a changing environment (although it is not the creature itself that adapts, but rather its species). The methods used to make species adaptable are remarkable, because the new requirements from the environment can't be known but estimated at best.

The same is true for software systems. At a certain point they meet customer requirements, but after a certain time the customer will require other properties from a system. The environment is represented by the area in which the system is used, and is influenced by customer goals, market, etc. Because none of these are static it is only a matter of time before requirements will change and force different expectations on the software system. But when the system can't provide the desired functionality, it will become useless (extinct). But as was mentioned earlier, at the current time systems are mostly not capable of adapting to new requirements.

The mechanism used by organisms to adapt to changing environments is based on being able to change the system (or species). These changes were possible from the start which means the relevant possible changes were prepared in the "design" of the species. This is what is also desired when designing software, and for this a software evolution analysis model is needed.

3.2. Representing evolution as a probabilistic process

3.2.1. The uncertainty of evolving requirements

Customer requirements are influenced by factors, most of which cannot be controlled. Factors such as market demand, company growth, future plans, etc. Because software systems are based on these requirements they are also affected when one or more of these factors cause a change in customer requirements.

The best way to make sure a system can handle new requirements, is to prepare the system for every possible change. This is impossible however, since the system would become extremely big and expensive (assuming it is possible at all). Therefore it should be possible to know to a certain extend what new requirements will occur, and prepare the system accordingly.

Although it is not possible to be certain what new requirements will occur, it is possible to identify several likely requirements to occur. These requirements can be based on events that influence customer requirements such as market expectations, company planning, etc.. These events will not occur with certainty, but there is a probability that they will occur. This means that it is possible to define the occurrence of requirements changes as a probabilistic process that is based on these events.

This makes it possible to work with evolution with all the tools that can be used with probabilistic processes. The main difficulty will be, however, defining a probabilistic process for evolution scenarios that model their real-world probabilities accurately.

3.2.2. Applicable reasoning techniques for probability models

After an accurate model has been defined for future requirements, the next step will be to determine which requirements will be likely to occur in order to prepare the system accordingly.

Because there are different ways to prepare a system for certain requirements, it is also necessary to determine what the best approach will be for system preparation. Reasoning techniques are needed, that can draw conclusions based on probabilistic models and implementation alternatives.

Similar reasoning techniques can be found in reliability theory where several system parts combined imply system reliability. In reliability theory for each part a probability model is defined that represents the chance that the part will fail. There are several possible relations between parts and these imply the total system reliability. Reasoning techniques are used in this field to determine where to increase reliability to achieve better system performance. This can be related to evolution and where to prepare a system to be able to cope with future requirements.

These reasoning techniques are based on states. At a certain point the system is in a certain state (a certain set of requirements have occurred) and the question will be how the system should be prepared for the possible future requirements. This will depend on what requirements will be likely to occur (or bring high risk) and the way in which the preparations are done (type of implementation, reusability, etc.). The reasoning techniques should be able to determine what requirement to prepare for and the best way of doing that. For this type of state-based reasoning Markov-models are appropriate since they typically can determine the best step to take based on previous states.

For this specific type of problem the stochastic dynamic program provides the best tools. This type of Markov-model incorporates stochastic behaviour (the uncertain occurrence of new requirements) and the ability to work with direct consequences of actions (the cost for preparing a system with a certain technique). Furthermore, there is an inductive algorithm that helps with reducing the computation complexity, which is necessary because of the state-space explosion that is likely to occur.

3.3. A software evolution analysis model

This section will describe what kind of model is needed to work with evolution aspects, what requirements it will have to fulfil and what issues it should address. The last three sub-paragraphs will describe the approach that will be used for the software evolution analysis model.

3.3.1. The use of an analysis model

The problems that come with evolving software have been discussed, and popular methods are not able to provide methodologies to cope with these problems. This leaves customers and software engineers with the fact that maintenance of software systems can be responsible for up to 70 % of the total costs ([Pressman1992],[Aksit1994]). And with intense competition on the market nowadays, it is always profitable when costs can be reduced.

An analysis model for software evolution can reduce cost, when it is able to predict future requirements and provide advice on the best ways to prepare the system for these changes. When probabilistic models are chosen as the way to represent the evolution process, there is no certainty about what is going to happen, but it is possible to prepare for scenarios that are more likely than others. And when it is related to different implementation options it also becomes possible to determine how efforts are influenced by this work (and with that the costs).

This would lead to an extra investment during the production of the base system, but when the analysis model's predictions turn out to be correct, money will be saved. When the predictions turn out to be false, the invested money will not have a payback. Because of this the software evolution analysis model will be used as a validation to determine whether or not money should be invested to support uncertain scenarios.

The analysis model would lead to the following results:

- Software systems can be used for a longer period of time;
- Maintenance costs would decrease because of a more balanced design and implementation;
- Risk of unexpected costs for customers is decreased since software systems will be better prepared for future requirement-changes;
- A better understanding of the system structure and critical points of the system;
- Justifiable investments can be done for uncertain scenarios;
- A better insight is given into the relation between the occurrence of requirements, their causes and effects.

3.3.2. Requirements for the analysis model

To be able to define a proper software evolution analysis model, requirements have to be made on what the analysis model should do. Furthermore a number of assumptions will be made for the analysis model to be able to work. The requirements and assumptions will be covered in this part.

Assumptions

To be able to use the software evolution analysis model the following assumptions are done:

1. The software system will be used in a dynamic environment and will be subject of changing requirements;
2. It is possible to a certain extend to determine what new requirements are possible to occur for the system;
3. It is possible to define a probability model for each possible new requirement that represents the chance of occurrence for this requirement;
4. During the life-cycle of the software product, no techniques and methods will be used other than the techniques and methods currently available;

Requirements

The analysis model should meet the following requirements:

1. The model will be able to relate customer behaviour (the possible introduction of new requirements) to system complexity in order to be able to determine how system complexity will increase.
2. The model will be able to determine which (groups of) requirements and additional system complexity is probable to occur based on probability models for individual requirements;
3. The model will be able to determine what alternatives for system implementation can be used best in order to minimise cost and risk when certain evolution scenarios occur;
4. Based on the likeliness of evolution scenarios and implementation alternatives the system will be able to provide advice on how to prepare the system for the possible future requirements on the software system.

3.3.3. Extensive requirements modelling and analysis

The first step of the software evolution analysis model will be to analyse the way in which initial and future customer requirements relate to each other. The possible future changes need to be identified by using problem domain analysis, variability analysis etc.. This will lead to a collection of customer requirements that can be divided into known requirements and possible requirements.

The new requirements cause extra system complexity and this system complexity in turn can make it difficult to reuse an existing system. By modelling the relations between existing requirements, and known possible requirements-changes it is possible to determine the impact of evolution scenarios. This way of modelling can also help out when evolution scenarios occur that were not foreseen. The existing requirements relation model can be used to insert the new requirements and the consequences of the upgrade can be assessed.

When it is possible to determine what impact a new requirement will have and what its probability will be, it is possible to relate the problems (and for instance costs) to customer behaviour. When this is possible, a valuable insight can be given into the causes and consequences of system upgrades.

3.3.4. Preparations for software evolution

The next step of the evolution analysis should be to determine how the system could be prepared for the possible new requirements. What different implementation options are available? How can these options be combined? How do certain options influence the use of others? Also a method is needed to determine how cost estimations can be made based on the use of these techniques and options.

This part essentially covers identifying the evolution capabilities for a system, and should result in a pool of possibilities for adding support for possible new requirements. Each of these possibilities will have its own cost, pros and cons. The final step of the model will have to determine whether or not certain possibilities will be used to support future requirements.

3.3.5. Probabilistic reasoning

The final part of the analysis will have to result in conclusions on which evolution scenarios should be supported and what would be the best way of doing this. To be able to do this, both the results from the first step as well as the second are needed.

Besides using the probability models for the determination of which scenarios to support, risk should also be considered. It might be possible that a requirement might occur with a low probability, but is a high risk. This might imply that it would be better to support it anyway.

It is almost impossible to provide a black-and-white answer about whether or not scenarios should be supported. For some scenarios little preparation suffices while others require much more preparation before money can be saved. The analysis model should also provide the user with feedback on this issue.

3.4. Summary

In this chapter a closer look was taken at the specific properties of software evolution and how the problems and their causes can be analysed. By modelling software evolution in a formal way it will be possible to analyse the problems, the causes and consequences and ways of preparing for it.

Evolution of software systems will be seen as a probabilistic process that is time dependent. This probabilistic process represents the fact that it is not certain whether or not a certain requirement occurs for a software system. When this approach is chosen, all techniques available for analysis of probabilistic and stochastic processes become available. This way a better insight can be gained into the difficulties of software evolution and analysis is possible by using the available tools and techniques.

An analysis model is proposed that analyses the possible evolution scenarios, finds the most likely ones and proposes the best way to prepare for these changes in order to minimise impact. This is done by identifying possible future changes to customer requirements and relating them to the existing system. By assigning a probability to each change, evolution scenarios can be identified along with a probability of occurring. Possible ways of preparing the system can be assessed by these evolution scenarios in a probabilistic reasoning model.

4. A method for analysing the causes of evolution

Under domestication we see much variability. This seems to be mainly due to the reproductive system being eminently susceptible to changes in the conditions of life so that this system, when not rendered impotent, fails to reproduce offspring exactly like the parent-form.

Charles Darwin, the origin of species

4 A method for analysing the causes of evolution

The causes and consequences of evolution in software

In this chapter the first stage of the model will be defined. This is aimed at defining a method with which it is possible to relate the occurrence of new customer requirements to the impact the addition of these requirements has on the existing system.

4.1. Introduction

As was mentioned before, the problem of software evolution starts at the requirements modelling level. New requirements occur that impact existing requirements. The domain analysis methods that were mentioned try to address these problems by analysing a problem domain, which means that all relevant features are identified, and incorporated into reusable assets.

To make a sound decision on how a system should be prepared for software evolution problems the first thing that needs to be known is what can happen. Identifying the possible changes to the requirements is the first step, but other aspects of evolution should also be modelled. For instance the relations to existing requirements and the probability of occurrence.

In this chapter two models will be presented with which it is possible to represent the different aspects of software evolution that have been described above. By using the FODA feature model the relations between different functional blocks of the system can be modelled. The second model is a state-transition approach to representing software evolution scenarios. With these two models combined it is possible to represent the impact of known requirements changes and the possible ways in which they can occur.

4.2. Customer requirements and relations

To be able to assess the impact of new requirements on existing software systems, it should be known how these requirements are related to each other. Some requirements might be totally independent, while others not only need specific functionality's to be present but also force constraints on other requirements and functionality's. This paragraph will present a method on how to model the requirements and their relations.

4.2.1. Feature based customer requirements analysis

Requirements modelling requires skill and experience since engineers and customers tend to think in different concepts. While a customer may think about the entire system when requirements are discussed, a software engineer may think in many small parts that together

form the system. Normally this doesn't have to be a problem, since the engineer can examine the desired system requirements and extract the needed requirements for each part.

Customers typically think about a system in desired functionality. And because software evolution is caused by changes to the desired functionality, software evolution can be analysed best in terms of this functionality. The functional blocks defined by a customer can be referred to as *features*. A feature can be seen as a black-box that provides the functional behaviour that was asked for by the customer. When a system is defined in terms of features, it is comprehensible for the customer as well as the engineer. Also it is easier to define possible future changes because it can be expressed in the same functional black-boxes.
But features alone are not enough. As was described earlier, changes to existing requirements can affect other parts of the system as well. This is caused by the relations that exists between the different system parts. To gain a better insight into the problems these relations should be made explicit in the model. Therefore between features relations should be defined. Different types of relations are needed to accurately model the structure of the system.

On the subject of features and relationships-modelling FODA has defined a method that will be used throughout the thesis to represent the customer requirements. The notation and meaning of the FODA feature analysis will be covered in the next sub-paragraph. Furthermore an addition will be done to the notation of FODA to make the method more intuitive towards the problems caused by software evolution.

4.2.2. The FODA feature model

FODA is an analysis method that uses a feature based notation to represent the desired (software) system. For the thesis the feature model defined by FODA will be used and a short outline will be given below. The complete description of the FODA method can be found in [Kang1990].

Features in the FODA method are defined as *"the attributes of the system that directly affect end-users"*. Examples of features are for instance a manual or automatic transmission for a car. In FODA notation, features are represented by dots, and specialisations by lines.

According to FODA three different types of features can be identified:

- Mandatory features, features that form a necessary part of the system
- Alternative features, features that cannot occur in the system at the same time (for instance a manual and automatic transmission)
- Optional features, features that can occur in a system but are not mandatory

Furthermore FODA defines three different types of relations that are possible between features:

- *Depends-relation*, this relation means that for this feature to be implemented, the feature it is related to needs to be present. For instance for a manual transmission to be included in a car, it should first be possible to include a transmission at all.
- *Mutual exclusion-relation*, this relation occurs when a feature cannot coexist with two or more other features in the system. This is for instance the case for an automatic and manual transmission in a car. Only one of them can be present in a car.
- *Composition rules*, the third type of relation that covers the relations that cannot be addressed by the other two. In particular relations between features that do not seem to have an immediate relation towards each other. Composition rules can be defined on a per

relation basis. For example for an air conditioning system in a car the engine should have at least 100 HP. Such a relation can be modelled by using a composition rule.

A typical feature tree that is defined in [Kang1990] looks like this:

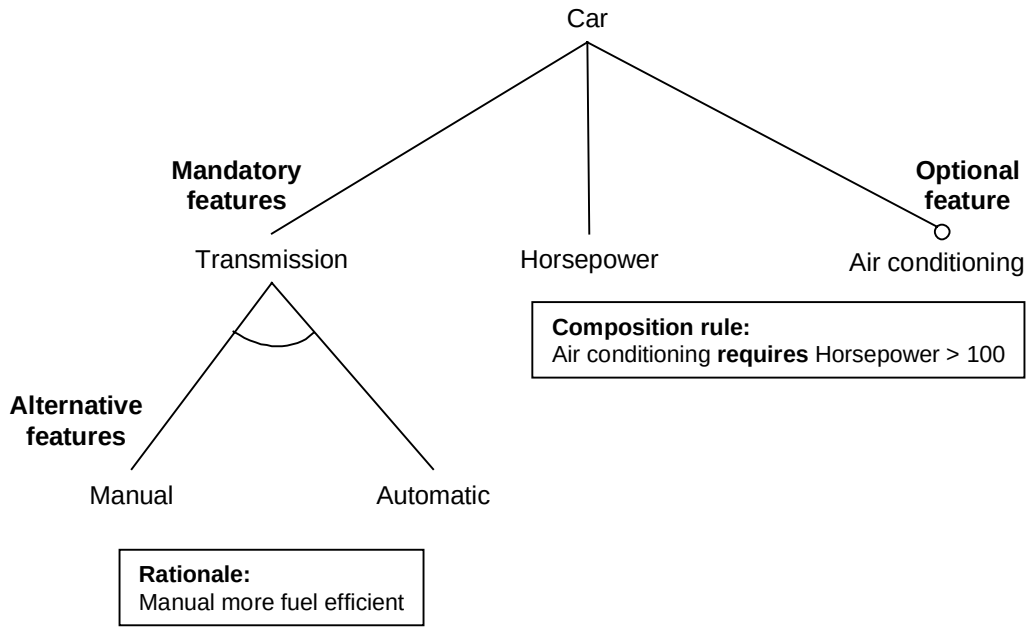


Figure 2: a typical feature tree in FODA

In this feature tree all three types of features and the possible relations are included. Optional features are depicted with a dot. A normal depends-relation is represented by a single line like for instance *car – transmission.* When two features mutually exclude, an arch is added like is done for manual and automatic transmission.

Another concept is the rationale. A rationale represents (intuitive) knowledge about the system and can help the customer in determining what features to choose. In the tree this is the fact that a manual transmission is more fuel efficient, which should lead to lower costs when the car is used.

A feature tree in FODA represents the possible systems of an entire problem domain since FODA is a domain engineering method. For instance the tree above represents several different cars like a car with an automatic transmission, or one with a manual transmission and air conditioning. This makes the modelling method very suitable for software evolution analysis. The possible systems the initial system can evolve into, can be represented in one feature tree (when the problem domain has been scoped correctly).

A composition rule restricts the freedom of other features and are because of this important for evolution analysis. The aim of using the feature model for evolution analysis is to attain a better insight into system complexity and dependency, in order to understand the impact of adding new features to an existing system. For this purpose the composition rule will be made visual in the feature tree. When a composition rule exists between two features, this will be marked by a dotted line. The picture would then look like this:

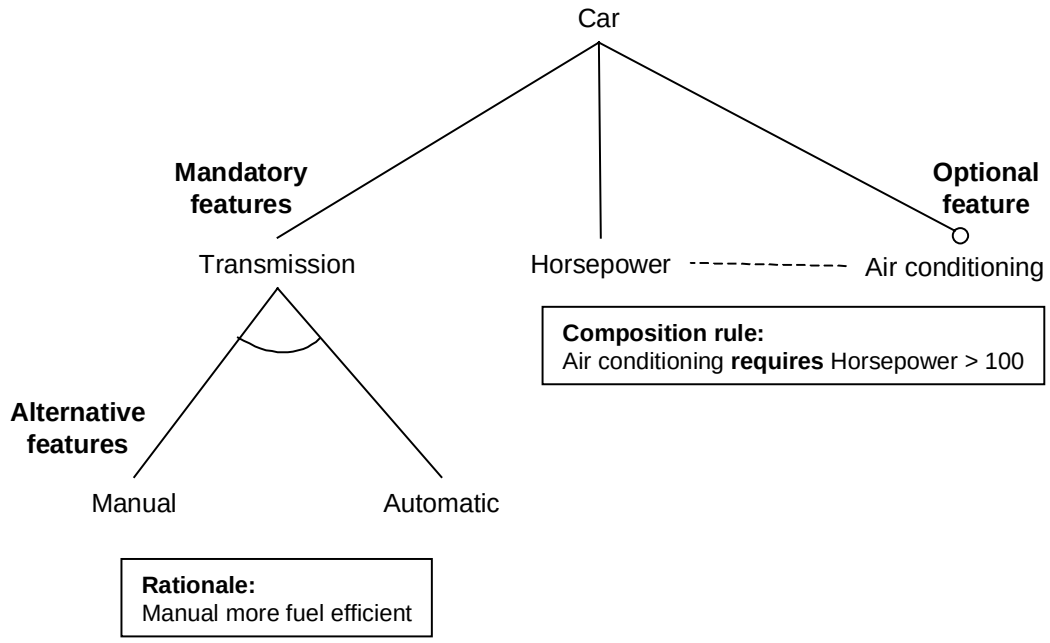


Figure 3: A feature tree with visible Composition rules

It should be noted that these composition rules are inherited for specialisations. When it is possible to have a super-airco, which is a specialisation of airco, this means that this is also true for this feature. However, it is also possible to overwrite the existing composition rule (for instance more than 120 HP is needed for super-airco).

4.2.3. Assumptions and modelling heuristics

The feature tree model that is proposed by FODA can lead to very large and complex feature trees. When these should analysed with regard to software evolution aspects, the amount of work can become overwhelming. Because of this a number of modelling heuristic has been defined below.

Because of the possible complexity of the evolution analysis, the main concern is to limit the system that is to be analysed. For this purpose it is reasonable to search for parts that can operate independent of each other. A part in a feature tree that is independent, can be identified in the following way: whenever a subtree that starts at the top-node has no composition rule-relations to features outside the subtree, this subtree can be considered as an independent subsystem. Such subsystems can be analysed individually. In the picture subtree I represents an individual subsystem, while subtree II is not an independent subsystem.

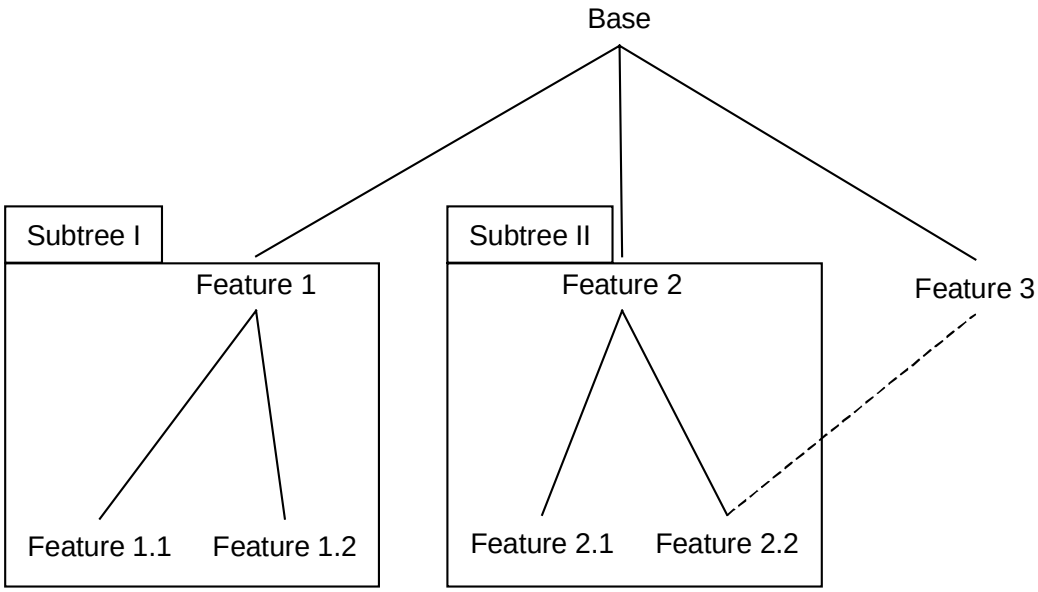


Figure 4: Dependent and independent subsystems

O
Modelling heuristics specific for feature tree modelling can be found in the FODA documentation.

4.3. Defining the possible evolution scenarios

Besides knowing what changes can be done to the existing system it is also important in what possible ways these changes can occur. The order might be relevant to the complexity that can be expected and it might for instance be possible that certain changes cannot be done anymore from a certain point. What is needed is a way of modelling the possible evolution scenarios. In this section a model will be proposed for this purpose

4.3.1. A model for possible evolution scenarios

The ways in which the features that have been identified can be demanded by the customer can affect the impact the request has on the software system. By modelling the scenarios that are possible with the known possible changes this can be analysed.

An evolution scenario consists of several steps, when the changes are serialised. At a certain point in time a software system exists and an evolution request occurs. The system is changed accordingly which leads to a new software system on which new evolution request can be done. Such an evolution scenario can be modelled as a state-transition diagram. A state in the diagram represents the current system and a transition represents the event that the customer changes the requirements of the current system.

Several evolution scenarios can share one or more states with other evolution scenarios for a software system. And since the start-state will be the same for all of them they can be modelled in one state-transition diagram which includes all possible evolution scenarios for known changes. This state-transition diagram will be called a *requirements evolution diagram*.

By defining the states of the requirements evolution diagram in terms of features, the graph can be related to the feature tree. This way it is possible to determine the complexity of going from one state to another based on the type and amount of relations that need to be changed in the feature tree.

A state in the requirements evolution diagram represents a software system made up from a set of features. It is possible to determine the difference between two systems by analysing these feature sets. For instance for a system *X* two possible additions (features) have been defined: *a* and *b*. This would mean that *{ X }* is a state. Now the evolution scenarios can be defined. From state *{ X }* either feature *a* or feature *b* can be asked for by the customer. This means from state *{ X }* two different states can be reached: *{ X, a }* and *{ X, b }*. And from both these states the state *{ X, a, b }* can be reached. This requirements evolution diagram is depicted on the right.

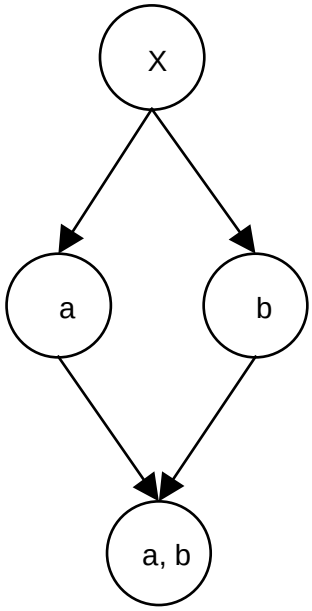


Figure 5: requirements evolution diagram

Depending of the nature of software evolution and the system that needs to be analysed it might be necessary to include more states in the requirements evolution diagram. For instance this requirements evolution diagram always ends up in a state where the customer has demanded all possible extensions. This might not necessarily be the case, and then more end-states will be needed. Also the order in which the features have been demanded can be relevant. This would mean for the graph on the right that not only *{ X, a, b }* but also *{ X, b, a }* should be included.

For this model to capture all the possible evolution scenarios, based on the relations defined for any feature tree in the FODA model, a number of basic requirements evolution diagrams can be defined that will be described in the next subparagraph. The resulting requirements evolution diagram will contain all possible evolution scenarios; each unique path through the requirements evolution diagram will be a possible evolution scenario.

4.3.2. Extracting the evolution diagram from the feature tree

The different types of relations that can exist between features in a feature affect the way in which a evolution scenario can unfold. By extracting the evolution diagram from the feature tree these limitations can be included in the requirements evolution diagram and it will represent the proper ways in which the software system might evolve.

For each relation that can exist between two or more features the mapping to a basic evolution diagram structure will be explained below. It should be noted that this is done assuming the order in which the features are demanded is irrelevant. The mapping does not become much more difficult when this is not the case, since only equivalent states need to be added because less states can be shared by evolution scenarios.

The evolution diagram should be extracted started at the top of any feature tree. Any feature that can be reached within one step (or relation) from the current feature set should define a new state in the requirements evolution diagram that can be reached from the current state. By combining the mapping described below the proper customer behaviour can be modelled.

Independent features

Suppose there are two features *a* and *b*. These features have no relation to each other, besides that they belong to the same system. This would lead to the following feature tree:

This means that there is no restriction to the order in which feature can be asked for. There is also no need for specific feature to be present, when another feature is to be added to the system. In the feature tree on the right two independent features are present; *a* and *b*. The evolution diagram for this feature tree can be calculated in the following way.

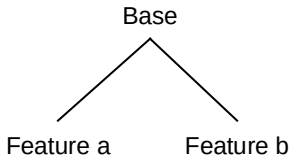


Figure 6: Relation tree for *a* and *b*

The states that are possible in the evolution diagram can be divided into three levels or stages: no features have been asked for, one feature has been asked for or both features have been asked for. This leads to the following evolution diagram:

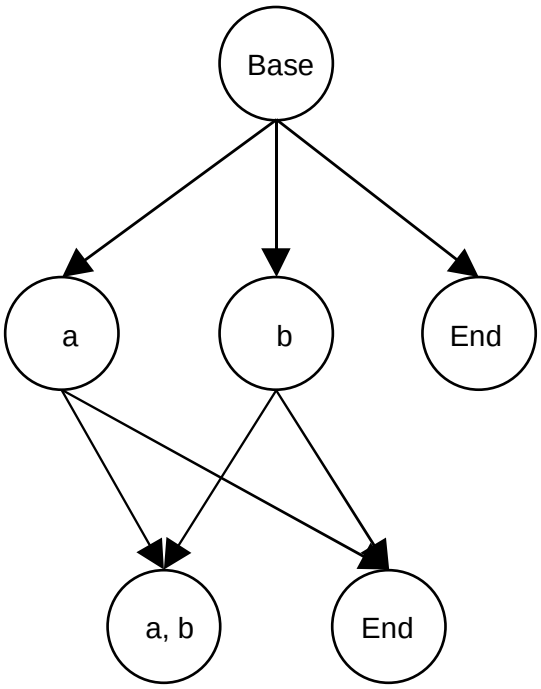


Figure 7: Evolution diagram for independent addition

At the first stage no features are known and any of the possible features can be asked for. This means there are three possible states at the second level (a was asked for, or b was asked for, no feature was asked for). These states can be reached from the first level. When no feature is asked for, this means that now and in the future the customer will not asks for new functionality. This should be included to complete the customer behaviour. This state is an end-state which means that it is not possible to do a state-transition.

At the second level each state has two possible actions (or events); either the feature that has not occurred will be asked for, or the feature will not be asked for. In the first case, the next state will be that all features are asked for which is an end-state. In the latter, the graph will also reach an end-state but now only one of the two features has been asked for.

Dependency

When the relation between *a* and *b* is changed to a specialisation-relation, for instance when *b* is a specialisation of *a*, this restricts the way in which the customer can behave Whenever a customer asks for feature *b* directly, this means that *a* needs to be implemented first. However, when a customer asks for feature *a* this doesn't necessarily mean that feature *b* will be asked for. The feature tree will now look like this:

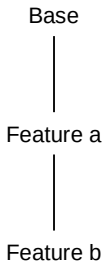


Figure 8: Specialization feature tree

The requirements evolution diagram now becomes:

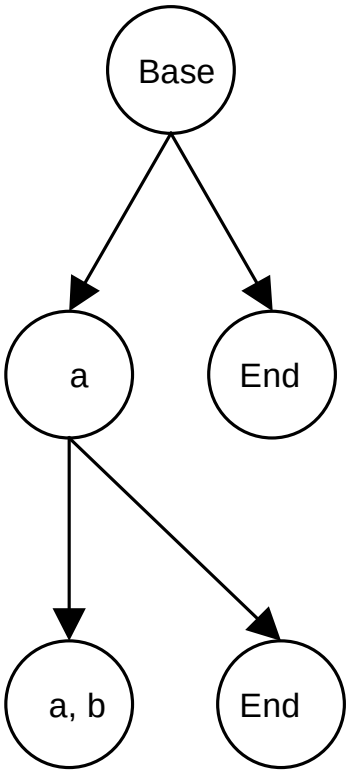


Figure 9: Evolution diagram for dependency

At the first level no features have been asked for yet, and there are two possible states that can be reached. Either *a* is demanded by the customer, or no feature will be demanded at all. Although this looks like the customer can not asks for b at all, this is solved when the probabilities are defined for the features. When feature *a* will be asked for with probability P(a) and *b* with probability P(b), this actually means that the probability of *a* being asked is raised because *b* requires *a* to be present.

The second level is fairly straight forward. Once *a* has been asked for (either directly or indirectly) *b* is the only feature that still can be demanded. Thus two states can be reached: a state that contains all features and an end-state when *b* is not demanded by the customer.

Mutual exclusion

The third possible relation between features is mutual exclusion. This means that two features can not coexist within one system. Suppose *a* and *b* share a mutual exclusion relation, which would lead to the following feature tree:

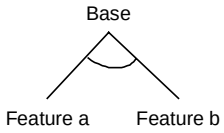


Figure 10: Mutual exclusion feature tree

From this feature tree the following evolution diagram can be made:

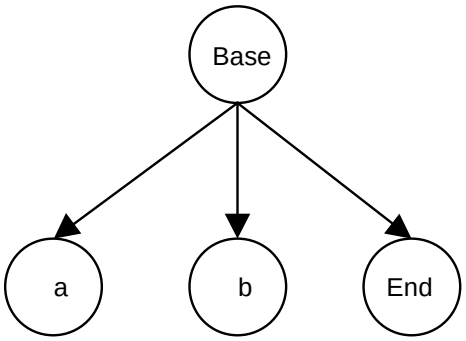


Figure 11: Evolution diagram for mutual exclusion

At the first level no features are asked for, and once again there are three possibilities: either *a* or *b* is asked for, or none at all.

The second level in such a case is the end level. Because when one feature is asked, the other can not be required. And when no feature is asked for this ends the process.

Composition rules

The last relation that is possible between features are composition rules. For this type of relation no specific translation can be given, but the relation can be expressed by the constructions defined above. For each specific rule, a proper construction has to be defined.

For instance the example of airco and horse power can be solved by mapping it into a evolution diagram in the following way: the airco puts a constraint on the amount of horsepower that is needed. This constraint actually means that the feature horsepower is divided into two different features: the feature horsepower that contains an amount of horsepower smaller or equal to 100, and a feature that contains an amount of horsepower greater than 100. These two features obviously mutually exclude each other. The airco feature now relates to the second horsepower feature in much the same way as a specialisation relation, meaning that when airco is asked for the horsepower feature needs to be present where horsepower exceeds 100.

It should be noted however that this is not a proper dependency relation so the way in which the costs are calculated for such a feature addition might greatly differ for these types of relations.

Composition rules affect the system severely because the impact can involve large parts or the entire system, which rarely is the case with the other types of relations. For this reason the

composition rules are used to identify system blocks that can be identified for individual software evolution analysis.

4.3.3. Evolution diagram extraction algorithm

Computing a evolution diagram is a straight-forward process, and can be easily be automated. There is an algorithmic approach to calculating the states of a requirements evolution diagram that will defined below in pseudo-code. The calculation is expressed in a depth-first, by following all unique path-parts of the evolution diagram:

```
Set the feature-set to all features;
Set the current state to the initial state;

For all features in the features-set
{
	Check whether this feature can be added to the
	Feature-set of the current state;

	If the feature can be added
	{
		Calculate the corresponding next state including next states;
		If the calculated state is unique
		{
			Add the calculated state to the set of possible
			next states of the current state;
		}
		else
		{
			Add the already defined state to the set of possible
			next states of the current state;
		}
	}
}
```

When the corresponding next state is calculated including next states a recursive call is done until a state is reached for which no possible next states exist. The feature-set for subsequent states is the feature-set that contains all features without the features that are already present in the current state (these cannot be asked for anymore). To optimise computations it is best to check whether a calculated next state already exists, before it is added to the possible next state-list of the current state. When a duplicate state exists this state can be added instead and the requirements evolution diagram will be minimised in terms of needed states and possible paths through the graph.

4.3.4. Adding probability models to the evolution diagram

Probabilities of evolution scenarios

After the entire requirements evolution diagram has been defined, the probability models need to be added. As described before, every state represents a certain set of features the customer has asked for and in every state there are transitions that can be done. Each transition stands for the event that the customer asks for a feature from the set of features that has not yet been asked for.

Because the customer chooses the next feature from the set of available features, this means a probability can be assigned to every feature that is left. This probability represents the chance that the customer will ask for this feature as the next. A direct consequence of this approach is that the sum of the probabilities for choosing a specific feature should always be smaller than or equal to one.

In a requirements evolution diagram it is possible to reach states that are end-states but do not contain the maximal amount of possible features. This is because when a customer can choose his next feature from a set of all possible remaining features, this doesn't always mean that customer will choose a feature. It is also possible that the customer will not ask for a feature anymore (no matter how long is waited by the system engineer). The states in the evolution diagram the represent such events are named premature end-states.

The probability of ending the customer feature requirement prematurely is defined by the probability of the features that can be asked for at a certain stage. The probability will be one minus the sum of all other probabilities, because it represents the event that the customer will not ask for any of the features.

The probabilities for the features that can still be asked is fairly unrestricted and can different for every single state in the requirements evolution diagram even though for several states they can be equal. As long as the sum of probabilities of reaching the next states of a certain state equals one, the distribution is valid.

The reason for this is that when it is possible to define new probabilities for every single state in the evolution diagram, it is possible to adjust the probabilities for the features that have occurred. For instance, suppose in the initial state the customer can choose from three features: *a*, *b* and *c*. And each of these features can be asked for with a probability *P(a)*, *P(b)* and *P(c)*, respectively. This also means that the customer will not ask any feature with probability:

$$P(\text{no features will be demanded}) = 1 - P(a) - P(b) - P(c)$$

But when the customer asks for feature *a* first, the evolution diagram will go to another state, where the customer can only ask for *b* and *c*. It is possible to redefine *P(b)* and *P(c)* because it might be the case that the customer will ask for *b* or *c* with a higher (or lower) probability when *a* has already been asked for.

Correcting probabilities for dependent features

As was described before, features that share dependencies should be treated somewhat differently. When a feature can only occur when another feature is present in the system, this means that the probability for the needed feature is influenced. This probability is composed of two parts: the probability that the specific feature is asked for directly and a correction because the feature can also be asked for indirectly.

For instance, when feature *b* needs feature *a* in order to be implemented the probabilities can be calculated in the following manner:

The probability that *a* will be asked directly: $P(a)$
The probability that *b* will be asked: $P(b)$
The probability that *a* will be asked (total): $P(a) + P(b)$

And because *b* can not be asked for at the first level, its probability needs to be corrected for every state from which it can occur:

The probability that *b* will be asked for when *a* has been asked for: $P(b) / (P(a) + P(b))$

This probability should only be applied to the next state from which feature *b* can now be asked for. This way it is possible to correct the probabilities for each of the dependent features.

For example the feature tree and evolution diagram with dependent features that were defined before:

Figure 12: Specialization feature tree

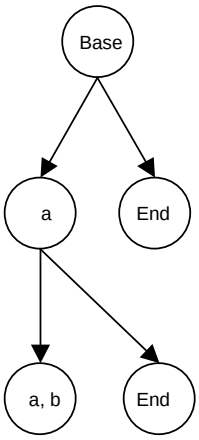


Figure 13: Evolution diagram for dependency

In this example feature *b* is dependent on feature *a*. When for feature *a* a probability of *P(a)* has been defined and for *P(b)* the probabilities of state transition in the evolution diagram can be defined.

For the transition from *Base* to state *a* the probability is made up from two parts: the chance that *a* is asked for directly and that *a* is asked for because *b* is asked for. This means to total probability of reaching this state from the base-state will be: P(a) + P(b).

The other state that can be reached from the base-state is a premature end-state. The probability for this will be: 1 – (P(a) + P(b)).

The probabilities of the two remaining transition now are defined by the previous level because of the dependency. For the transition from state *a* to *a, b* the probability will be: P(b) / (P(a) + P(b)). This is the corrected probability that was defined above. The other state that can be reached again is a premature end-state. The probability for this transition becomes: 1 – (P(b) / (P(a) + P(b))).

4.4. Summary

In this chapter a closer look was taken at the impact of evolution and ways of modelling customer behaviour and evolution scenarios .

By defining the requirements for a software systems using the FODA method, it is possible to model the relations that exist in the system in terms of customer concepts. Because FODA is a domain analysis method it is possible to describe all the systems that can be made within the scope that is defined. This makes it possible to define all possible system with the features that have been found as the possible extensions. The way these features relate to each other define the possible complexity of the software systems that can be the result of software evolution.

To model the possible evolution scenarios, the requirements evolution diagram model is defined. A requirements evolution diagram consists of states and actions. A state contains the features the customer has already asked for, and an action represents the event that the customer asks for a new feature. A requirements evolution diagram can be extracted uniquely from a FODA feature tree.

By defining probabilities for the actions, the uncertainty of customers asking for a specific feature can be represented. Thus it is possible to model the different ways in which a customer can behave with regard to new requirements on the system. The requirements evolution diagram can be defined in different terms of detail with respect to customer behaviour that has occurred. It is possible to include the order in which the features have been demanded varying from no order at all to a full history in every single state. Depending on the amount of detail that is needed for the analysis the model can be configured to supply the needed information.

5. Software techniques for coping with evolution

As natural selection acts by competition, it adapts the inhabitants of each country only in relation to the degree of perfection of their associates; so that we need feel no surprise at the inhabitants of any one country, although on the ordinary view supposed to have been specially created and adapted for that country, being beaten and supplanted by the naturalised productions from another land.

Charles Darwin, the origin of species

5 Software techniques for coping with evolution

Identification and analysis of possible evolution preparation

Problems that are caused by software evolution can be addressed by preparing the software system for known possible changes in the future. Whenever a feature is demanded, these preparations can have a positive effect on the effort that has to be done and thus minimise cost. In this chapter a closer look will be taken at these techniques and preparations, how they can be found and incorporated and how they influence effort and costs.

5.1. Introduction

In the previous chapters a methodology was defined with which it is possible to model the ways in which a software system can evolve. This makes it possible to determine the impact such changes will have. The next step is to determine in what ways the system can be prepared so it becomes easier and more cost-effective to reuse the system in case of software evolution.

Many techniques exist that make it easier the reuse software systems when it is known in which context this reuse will take place. For software systems that could be affected by evolution it is now possible to identify these techniques. By analysing the possible extensions and they in which they relate the existing system, the context of the reuse can be defined and applicable techniques can be selected.

Not every technique will be applicable in every possible situation and multiple techniques might even interfere with each other in providing the best solution for evolution preparation. And including preparations in the existing software systems is not without costs. This means the selected techniques need to be analysed on how they influence reusability and adaptability, costs and other techniques.

In this chapter a closer look will be taken at how software evolution preparation techniques can be found and which aspects of the techniques should be analysed before including it as a possible option in the optimisation formalism that will be presented in chapter 6.

5.2. Finding preparation techniques

Costs caused by extensions to existing systems relate directly to the way the extensions affect the structure of the existing system. The techniques that can prepare for evolution should therefore be selected based on the possible impact the identified changes can have. In this paragraph a description will be made of how to search for these techniques and how they should be selected.

5.2.1. Concerns for evolution preparation

When a system is built and prepared for future changes, this means that costs will be higher as compared to a system that will not be prepared. This means the ways of preparing a system should be assessed accordingly, because a wrong decision might lead to high costs that will never pay off, even when new requirements occur.

The use of a specific technique is dependent on the amount of problems caused by an evolutionary step. When a feature affects several existing features at the same time, and a certain technique is capable of simplifying this process, this technique can be a valid choice. But when the same technique is not as usable for other features, the choice might become a little less obvious.

Because of this, searching for techniques that can help with preparing for evolution should be based on how well the techniques are capable of solving the impact new features have on a software system. When a feature is added to a system, a feature tree depicts all relations its has to existing features. This way it is possible to determine the impact the addition of a feature has on a existing system. Whenever the amount of relations for a feature increases, this means that a feature has a bigger impact on an existing system.

When a feature has a small amount of relations to existing features, this means the impact will be relatively small although the feature should be able to function properly in the system. The relations that are defined for a feature in a feature tree all have their own contribution to the impact of the specific feature.

The specific type of relation also determines how well a technique is able to solve problems. In a feature tree a dependency might differ in character for different sets of features. A close look should be taken at the relations in the feature tree before selecting preparation techniques for the specific system.

Besides the feature trees and their relations that influence the applicability of preparation techniques, also additional customer requirements can be of influence. Some of these requirements might not be represented in the feature tree but should still be included in the analysis. For example requirements on performance of the system, cross-platform usability, etc..

5.2.2. Identifying available techniques and knowledge sources

From an engineering point-of-view the impact of software evolution depends on how the software system was implemented initially. This is because a feature in a feature tree is not mapped to a concept in the software system in the same way every time. It is possible that the functionality a feature represents in a feature tree, is provided by several modules in a software system that contain numerous objects and classes.

Most of the time a mapping of features directly to concepts in a software systems is very difficult because they represent complex functionality’s. Because features represent system requirements in a way that customers can relate to, they abstract underlying complexity. This means a single feature can represent a simple modular part of a software system, but it can also represent a complex database system that involves semaphoric shut-out, atomic transactions, etc.

Such a representation makes it very difficult to define a system that contains the features as identifiable building blocks, if not impossible. So if a system should be prepared for features

that might occur, the impact on the system will differ for each implementation. When a system is prepared for the impact of new requirements with certain techniques, the techniques should extend the implementation in such a way that it is easier to implement these new features.

Identifying applicable techniques therefore is dependent not only on the features that should be supported, but also on the way in which the initial system will be implemented. And because system implementation is based on experience combined with common methodologies, not every possible technique is applicable.

The software evolution analysis method analyses future extensions for which it is possible to determine how they will interfere with existing system functionality. Generally speaking for a certain project the engineer has knowledge on how to re-use system parts such as re-use techniques, inheritance structures, etc. This knowledge can serve as a starting point for identifying applicable techniques for the specific project.

Another source for techniques are new developments in computer science and software engineering. New technology may be able to address specific issues caused by a certain feature much better than existing (or known) techniques. This could mean, however, that the use of these techniques involve courses to understand them properly, and that the experience of using the new technology is missing. This could have its effect on the overall costs.

5.3. Identifying how techniques influence future efforts

After applicable preparation techniques have been identified the next step is to determine how these techniques can help out during software evolution scenarios. In this paragraph several points of attention will be mentioned for analysing preparation techniques.

5.3.1. Technique application

When techniques for software evolution preparation needs to be analysed it is not possible to define a standard way of doing this. The relevant properties of techniques depend on many different aspects of the analysis. When the analysis is performed on a low level it can be possible to analyse the techniques up the implementation level, while this is very difficult and labour intensive for a high-level analysis. The goals of the analysis of the techniques is the same for every analysis, which means a general approach can be defined.

The software evolution analysis model is aimed at finding the optimal development policy for evolving software systems with respect to the costs. The selected techniques for evolution preparation each modify the costs caused by features in a certain way. Depending on the technique the costs may decrease, stay level or even increase when a certain feature occurs. To be able to define a cost model for using these techniques, it is important to know how these techniques behave with respect to the possible evolution scenarios. Based on these characteristics it should be possible to define an accurate cost-model (see sub-paragraph 5.3.2).

This is a very important step of the software evolution model, because the way a technique influences costs of software evolution will be used to determine what the best way will be to prepare a software system. The costs that can be expected when a technique is included in the system will be compared to the costs when other techniques are included and to a worst-case scenario when the system is not prepared at all.

The worst-case scenario is evaluated as a reference for assessing techniques for preparation. The scenario is defined as what happens when a system is not prepared for any new

requirements. This scenario should be considered a valid option, however. When the probability of extension is very low, the probability of additional costs will also be very low. This means that the best solution should be to minimise the costs for the initial system which is done by not preparing a system for future changes. Because when a system is prepared, this means additional costs were raised because of this preparation. And these costs will most likely never have a pay-off.

Individual techniques can be assessed based on this reference scenario. The occurrence of each feature can be assumed, and the costs that are caused by this event can be calculated. When this is done for each feature a valid assessment can be made of the technique and whether it is useful for preparing this particular system for evolution scenarios.

When a technique is applied the way it behaves in different evolution scenarios might not be the same every time. The order in which the features have occurred can make it easier or more difficult for using a technique. This is why the requirements evolution diagram should contain all the relevant information for cost calculations.

This is called the *context* in which the technique will be applied. The different aspects that can influence the advantages of a technique in dealing with evolution problems should be analysed to assess the effects on costs.

5.3.2. The influence of combinations of techniques

Techniques that influence the costs of software evolution can be also be influenced themselves by other techniques. The specific advantages might be cancelled out, less effective or even more effective. Two or more techniques might be mutually exclusive but can also be able to exist in one system and not interfere with each other. The techniques would then be able to assist when necessary and the total reduction would be better than when one of the techniques would have been left out.

Most of the time it is possible to analyse the techniques and determine beforehand whether a technique is able to cooperate with another technique, or not. The groups of techniques that can be chosen by an engineer can be analysed in much the same way as the individual techniques. These combinations of several techniques can be seen as a new technique which has its effect on the costs that are associated with extending system functionality.

As well as features, techniques can also exclude each other. The structures defined by a technique can influence the use of other techniques or even exclude them. There are two types of mutual excluding techniques.

For independent techniques the mutual exclusion is a straight forward process. Because of the mutual exclusion the techniques are not able to be used in the same system at the same time. For the analysis of the evolution scenarios this does not mean that choosing a specific technique cancels the other techniques out permanently, but rather that all uses of the other techniques have to be replaced. When another excluding technique is chosen at a later stage, all excluding techniques have to be replaced once again.

The other type of exclusion is a stronger restriction. This type of exclusion between techniques restricts the engineer of choosing a certain technique whenever another technique has been chosen. This type of exclusion does not occur as often as the other type but serves a specific purpose. For instance, when a certain technique has an initial cost that needs to be paid (e.g. a course) the technique can be divided into two different techniques: a technique with the initial cost and the inited technique (where the course has already been done). These techniques share a strong mutual exclusion. Whenever the first technique (with initial costs) has been chosen, it

can not be selected again. From that point it is replaced by the inited technique as a valid decision.

The analysis of combinations of techniques should be an integral part of the analysis of techniques, because the influence on the costs can be quite relevant. The amount of work can become quite large but once this is done it is possible to reuse the acquired knowledge can be reused in future analyses.

5.3.3. Cost modelling

Costs for implementing software systems are caused by the time and effort that is needed to implement the desired functionality. And with software evolution the time that is needed to implement new features is greatly extended because of the necessary effort to make the new functionality work together with the existing system. The use of techniques influences this amount of time needed, making it easier to control costs when software evolution occurs.

The software evolution analysis model analyses problems that can occur with evolution for a software system by calculating the different configuration that can occur. Based on these configurations the selected techniques will be assessed based on the costs that can be expected when they are chosen. This makes it necessary to define a cost model for implementation of features and using preparation techniques.

The software evolution analysis model has no standard way of modelling these costs because the analysis can be performed on many different levels of abstraction. The properties on which the cost model will be based (lines of code, functional blocks, etc.) can be different for every analysis, and need to be defined for the specific project. This makes it possible to use any popular way of modelling costs, such as for instance COCOMO.

At the final stage the software evolution model will calculate the costs that can be expected from a certain point, based on the features that can occur, the techniques that were used and the cost model. To do this all the relevant information should be accessible for the model. This means that all the properties of features and techniques that are needed for cost calculations should be defined for use in cost computations. For instance when the cost model is based on the amount of method redefinitions that need to be done when features occur (and the techniques influence this amount and the type of redefinition), for the features the amount of methods need to be stored. For techniques the influence on method redefinitions needs to be defined, etc..

5.4. Summary

In this chapter a closer look was taken at the possibilities that are available to an engineer in preparing for evolution and likely evolution scenarios. These possibilities are defined in terms of techniques that make the process of adding features to the affected system easier and less costly.

The techniques influence the efforts that are needed to implement and incorporate new system functionality into existing systems. This means that these techniques and their associated costs should be defined by their specific structure and properties of features they influence. As a result, whenever a cost-calculation should be made the necessary properties for each identified feature and technique should be defined.

The experience an engineer has in this field can be a valuable source of identifying the techniques that might help in an evolution process for a specific software system. Another source for finding techniques is new knowledge. This source is somewhat more cumbersome to handle, because for new techniques valuable experience is missed. And costs are generally raised because courses or intensive study is needed to be able to use these techniques in an efficient way.

The techniques that have been identified define the way in which costs will be calculated. By preparing a system for software evolution, modifying at a later stage will become easier. The way in which it becomes easier depends on the techniques that are used. By defining a cost model based on the techniques, the software evolution model is capable of comparing the expected costs for different techniques in the possible evolution scenarios.

The software evolution analysis model has no pre-defined cost model, and therefore grants the possibility to use any way of modelling costs that is available. Models that have been used for a long time can be included in the calculations without difficult adaptations. The properties needed for the cost-calculations however, need to be done carefully.

6. Markov decision process formalism

Why, for instance, should the colour of a flower be more likely to vary in any one species of a genus, if the other species, supposed to have been created independently, have differently coloured flowers, than if all the species of the genus have the same coloured flowers?

Charles Darwin, the origin of species

6 Markov decision process formalism

Dynamic programs and software evolution optimisation

When analysing software evolution and ways of dealing with its difficulties, many parameters must be considered. Parameters that influence cost should be configured in such a way, that expected costs are minimised based on probabilities. This chapter will describe how to use Markov decision theory as a way to define the possible states in which an evolution process can be described and optimised.

6.1. Introduction

The final step of the software evolution analysis model will be to determine how the software system can be prepared best for the evolution scenarios that can occur. In the previous chapters all the relevant information has been acquired for reasoning about software evolution. What is needed is a formalism that contains all relevant aspects of software evolution and is capable of applying reasoning techniques to find the best development policy.

In chapter 4 a model has been defined where evolution scenarios are represented by state-transition diagrams. Software evolution is treated as a sequential process where requirements changes are done one after the other. From the engineer's point of view the decision on how to prepare the system for the possible requirements changes should be done from the current state in the evolution scenario. The decision should be made on which technique(s) should be used so costs can be minimised for the changes that can be expected. Once this decision has been made it is uncertain what will happen next in the software evolution process.

This type of problem resembles the theory of sequential decision problems closely. In this theory from a certain situation decisions are made that lead to new situations. Based on the characteristics of these situations and the goals that are chosen, it is possible to optimise the decisions that should be made from a certain situation. In this chapter a class of sequential decision problems, dynamic program, will be used to model the decision process that occurs for a system engineer when a software system should be prepared for the consequences of software evolution.

6.2. Dynamic programs: a short outline

The software evolution analysis model reasons about software evolution using a specific class of Markov decision problems: dynamic programs. This paragraph will give a brief description of the theory of Markov decision theory and dynamic programs. For a more complete description of Markov decision theory and dynamic programming a number of good text books can be found. For this thesis the following books were used: [Puter1994], [Gluss1972].

6.2.1. Sequential decision problems in general

Many processes involve making decisions at a certain point in time, and these decisions influence the situations that can occur in the future. Whenever a series of decisions in sequence is needed, this is called a sequential decision model or problem.

In [Puter1994] a sequential decision process is represented in the following way:

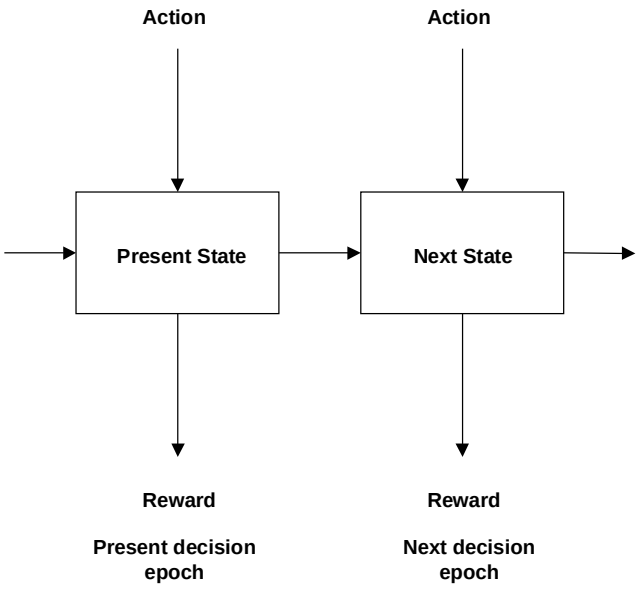


Figure 14: A symbolic sequential decision problem

The decision problem is in the present state and a certain action has to be chosen. This means the decision problem will not reach a new state until an action has been chosen. This is represented by the *present decision epoch*. When any of the possible actions is chosen, a reward-function will return a certain reward. By choosing an action a next state will be reached and a new decision epoch commences.

When the decision process is modelled in such a way that it is possible to determine the rewards by the information of the current state and the action that is chosen, it is called a Markov decision problem or a dynamic program. In these type of problems no information is needed from previous states or actions chosen in the past.

The solution for dynamic program is called a policy and consists of the optimal decisions for every state in the dynamic program. When the evolution process is modelled by means of dynamic programs, an optimal policy would represent the feedback that is desired by the engineer: which technique(s) to choose in every possible situation that are expected to minimise costs in the best way.

6.2.2. Dynamic programs

The Markovian state property

For a sequential decision problem to be a dynamic program it has to fulfil the Markovian state property. This means that every state in a dynamic program is completely defined by the

previous state and its actions. No other states or actions are needed to be able to determine the next state the dynamic program will reach.

This means every current state should contain all relevant information needed to make a valid decision about the available actions. When all relevant information for determination of the next state is available in the current state it is possible to determine the optimal decision for this state. The states should be defined in such a way that there is clear separation between a state and the possible history.

The Markovian state property includes the history relevant to the reward-function into every state of a dynamic program. This generally causes the state-space to become quite large, because the states can differ in many different properties. The complexity of solving a dynamic program directly relates to the amount of states that are present in the dynamic program. For this reason the states must be defined in such a way that they fulfil the Markovian state property but no more states than necessary exist.

Decision epochs and time-horizons

Decisions that are made are referred to as *decision epochs* in the theory of dynamic programming. Dynamic programming distinguishes between different types of decision epochs.

In a discrete decision epochs the decision epochs that occur follow each other directly. Every time a decision is made, it is directly followed by the next decision epoch. When the decision epochs form are a continuum the decisions do not occur at a specific time. They can occur at all decision epochs, as a result of random events or moments that can be determined by the decision maker.

Both of these decision epoch types can be finite as well as infinite. These are called time horizons. A finite time horizon means that the amount of decision epochs is limited. Infinite time horizons supply an infinite amount of decision epochs to the decision maker. Decision epoch sets that have a infinite time horizon supply the decision maker with knowledge about what can happen. Even for finite time horizons this can be true, because to amount of decisions that need to be made can limit the size of dynamic programming problems.

6.2.3. Cost- and reward-functions

Dynamic programs optimise decisions for a certain criterion such as costs or profits. The values are determined by a *criterion-* or *cost-function*. When decision are made in a dynamic program from a starting state x_0 the sequence of decisions leading up to an end-state are called a *policy*. Such policies are the input of a cost-function together with the start-state and the cost-function returns the costs for using the policy from that state. This implies that choosing a different start state would lead to a different result from the cost-function. The best policy can be determined by minimising or maximising the result of the cost-function for the different possible policies.

When deterministic dynamic programs are analysed, the cost-function will return a precise value for a specific policy, but this is not possible for stochastic dynamic programs. Because it is not possible to uniquely determine the next state for choosing a specific action it is not possible to calculate the exact value for the cost-function. For stochastic dynamic programs the cost-function will return an expected value for applying a certain policy. Suppose the following two dynamic programs:

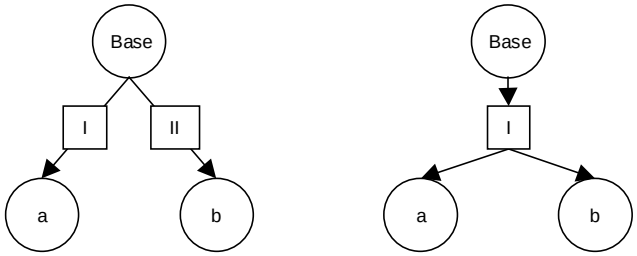


Figure 15: A deterministic and a stochastic dynamic program

The dynamic program on the left is a deterministic dynamic program because for every action that can be chosen (either I or II) it is known which state will be the next. The results from the reward-function can be compared for both actions and the one with the best result identified. This is not possible for the dynamic program on the left, which is a stochastic dynamic program. When action I is chosen, the next state can either be *a* but also *b*. And because different costs can occur from state *a* forward than state *b*, it is not possible to calculate the exact cost of choosing action I. When several actions exist for such a dynamic program they are compared based on expected costs for these actions.

While a cost-function is able to calculate values for entire policies from a certain start-state, the *reward-function* returns the contribution of choosing an action in a certain. A reward-function returns the value for choosing an action which represents the contribution for choosing this action to the total costs. Reward-functions can always be declared in the following way (here the function will return the reward for choosing action a_z in state x_y):

$$Reward(x_y, a_z)$$

When in a certain state x_y a certain action a_z is chosen the total contribution to the cost-function can be divided into a contribution for choosing that action in the specific state, and a contribution which is caused by applying the remaining policy to the next-state. This way cost-functions most of the time are defined in terms of a reward-function. For a deterministic dynamic program this can be exemplified in the following way:

Suppose for the start-state x_0 and policy $\{ a_1, a_2, \ldots, a_n \}$ the cost-function should return the value, and when choosing action a_x in state x_y you would reach state x_{y+1}. Then the cost function can be defined as:

$$Cost(x_0, \{ a_1, a_2, \ldots, a_n \}) = Reward(x_0, a_1) + Reward(x_1, a_2) + \ldots + Reward(x_{n-1}, a_n)$$

How this is done for stochastic dynamic programs becomes clear from the backward inductive algorithm.

6.2.4. The backward inductive algorithm

When a solution for a dynamic program is calculated, this is the determination of an optimal path (either maximum or minimum) through the dynamic program. One way of finding such an optimal path is calculating the value of every possible path in the dynamic program. But because of the Markovian state property the amount of states in a dynamic program tends to grow extremely fast. Even though the process is highly automatable, large problems can make the computations too complex for even very powerful computers.

The backward inductive algorithm is way of optimising the computation for finding optimal policies in dynamic programs. Most of the time paths through a dynamic program are not entirely unique, but parts of a path can also be used by other paths. For instance in the (stochastic) dynamic program on the right. In this dynamic program the path *(Base, a, c, e)* is not unique because for instance the path *(Base, a, d, e)* shares a part with it (*Base - a*). When the costs for every single path is calculated and at the end the optimal path is selected, this would mean calculations have to be performed for state *a* twice.

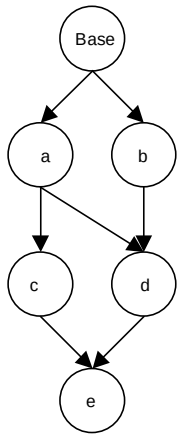


Figure 16: Paths in a dynamic program

The backward inductive algorithm divides the dynamic program in levels of equivalent states. Two states are equivalent in terms of level they belong to, when they can be reached in the same amount of decisions. For instance, when a system has two possible features that can be asked for, *a* and *b*, and there is a choice between two possible techniques, *I* and *II*, the following levels can be identified (ordered feature sets are not considered):

Level 0:	The level that contains the initial state of the dynamic program
Level 1:	This level contains every state that can be reached when one decision is made. In this case they would be *{a,I}, {a,II}, {b,I}, {b,II}* and the end-state after one decision when the customer doesn't ask for a new feature.
Level 2:	In this level are all states that can be reached when two decision have been made: *{a,b,I}, {a,b,II}*, and the end-state after two decisions when the customer doesn't ask for a new feature.
Level 3:	This level contains the total end-state.

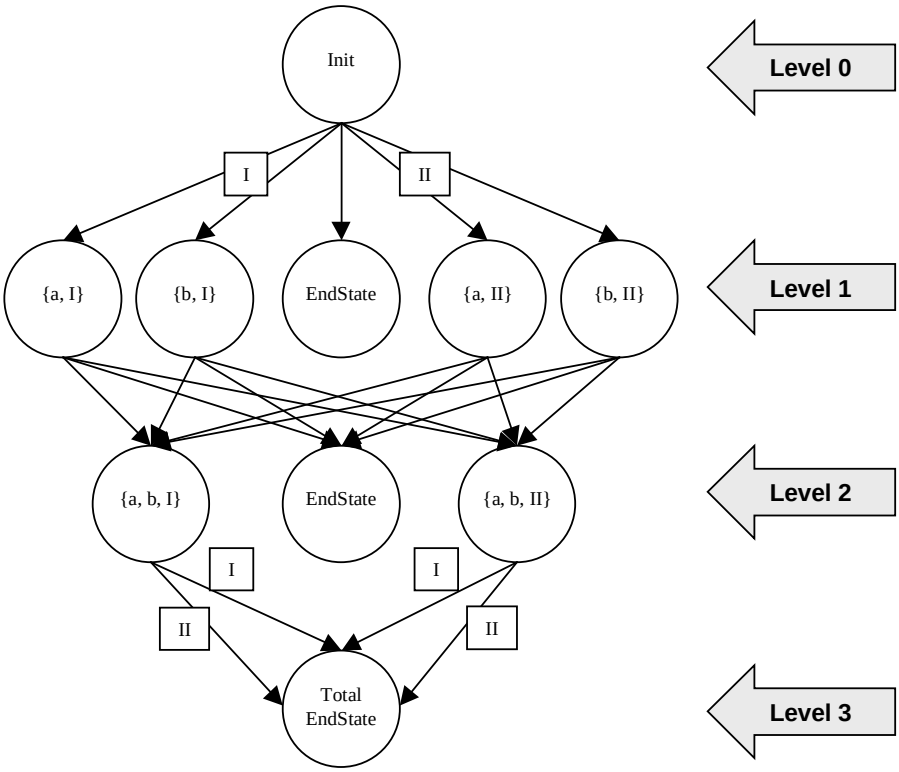


Figure 17: Simple dynamic program

The backward inductive algorithm starts at the highest level, and determines the optimal decision for every state in this level. The backward inductive algorithm now goes up one level and determines the best action for every state in this level. Because in the previous level all optimal decisions (and their costs) have been calculated for the states that can be reached from the current level, this computation is possible. This process is repeated until the chosen start-state has been reached.

In [Puter1994] the backward inductive algorithm is defined in the following manner:

1. Set $t = N$ and

$$u^*_N(s_N) = r_N(s_N) \quad \text{for all } s_N \in S,$$

2. Substitute $t - 1$ for t and compute $u^*_t(s_t)$ for each $s_t \in S$ by

$$u^*_t(s_t) = \max_{a \in A_{s_t}} \{ r_t(s_t, a) + \sum_{j \in S} p_t(j \mid s_t, a) u^*_{t+1}(j) \}$$

Set

$$A^*_{S_t, t} = \arg\max_{a \in A_{s_t}} \{ r_t(s_t, a) + \sum_{j \in S} p_t(j \mid s_t, a) u^*_{t+1}(j) \}$$

3. If t = 1 stop, otherwise return to step 2.

The first step of the backward inductive algorithm starts at the highest level and defines the function $u^*_N(S_N)$. Throughout the algorithm this will be the function that returns the value for choosing the optimal action in state S_N. At the highest level this value will be equal to the

result of the reward-function applied to this state: $r_N(S_N)$. Normally the reward-function also needs an action to calculate the reward, but since in the highest level there are only states without actions the reward-function can perform the calculation.

The second step goes up one level and calculates the value of the optimal decision for every single state in the level. The function $u^*_t(S_t)$ (where t represents the level of the dynamic program) is now defined as a Max-function. The value of every single action in the action-set A_{St} of state S_t is calculated and afterwards the maximal (or minimal) value is selected. The computation now consists of two parts: first the reward-function returns the value of choosing action a at state S_t. The other part of the costs of choosing this action is made up by states that can be reached when this action is chosen. This is done by a summation of every state that can be reached with action a, multiplied by the probability of reaching this state (expectance value). In this summation $p_t(j \mid S_t, a)$ is the probability of reaching state j when choosing action a at state S_t. $u^*_{t+1}(j)$ is value of the optimal decision of state j. This value has been calculated in the previous step of the algorithm. The function $A^*_{St,\,t}$ is identical to $u^*_t(S_t)$ but instead of storing the value it stores the optimal decision itself.

The third step checks if the algorithm has handled the last level. When this is the case the algorithm stops, and when not the algorithm continues with the next level. Because of this level approach the backward inductive algorithm calculates every transition once. This optimises the computation of solving the dynamic program, although in specific case the optimisation can be quite small.

6.3. Defining evolution with dynamic programs

This paragraph will describe how the process of software evolution and preparation can be modelled using dynamic programs. Based on the states and transitions of the requirements evolution diagram the states of the dynamic programs will be defined.

6.3.1. Identifying the states and state-space

In software evolution two events are important for the analysis: a requirements change by a customer and the choice of a preparation technique by the system engineer. When dynamic programs are used for the analysis of software evolution these two events together define the transition behaviour of the dynamic program. The states that are reached in the mean time must represent the relevant history up to this point. Furthermore must a state contain the possible actions that can be chosen. A state in the dynamic program can now be defined in the following manner.

{a, b, c, d, …}	*Set of features*
{I, II, …}	*Set of techniques used to implement the system up to now*
{1,2,3, …}	*Set of possible actions that can be chosen*

What does a state in the dynamic program actually represent? At a certain point during the evolution process the system engineer has to make a decision on what to do with the system (whether or not the system should be prepared). Such decisions are normally done when new functionality needs to be implemented, which can be at the initial level but also at a later time. A state will therefore be interpreted as a stage where a new requirement has occurred and still has to be implemented, while the other features already have been implemented. This divides the feature set into two parts: the features that have been implemented and the one that has not.

This makes it possible to model the event that the customer does not ask for any changes anymore, even though not all identified features have been demanded. This is called a *premature end* of the evolution process. In the dynamic program such a state have no actions and therefore can not be exited. These states will be referred to as *premature-endstates*. From the levels in figure 16 it can be seen that only one premature end-state is needed per level, since the costs that can be expected from a premature end-state are always zero: all features that will be demanded have been implemented.

6.3.2. Identifying the actions and action-space

As described above, an action in the dynamic program of an evolution analysis consists of two parts: the engineer who makes a decision and the customer asking for a new feature. Whenever a certain technique is chosen to implement the system with the requirements change, the current state is no longer valid and therefore is left. The next state depends on the feature that will be demanded by the customer next. This means a stochastic dynamic program is needed to represent the decision process of software evolution, where an action can have several different outcomes or next states.

An action can be defined in the following way:

Technique Y

Possible nextstate 1	*probability P(1)*
Possible nextstate 2	*probability P(2)*
.	
.	
.	
Possible nextstate y	*probability P(y)*
Premature enstate level x	*probability 1 – P(1) – P(2) – … – P(y)*

Each action represents a technique that is used to prepare for future requirements changes, and the possible states that can be next when this action is chosen together with their probability. These probabilities are the same as the probabilities that were mentioned for the occurrence of features and the transitions in the requirements evolution diagram. Because the dynamic program will be defined based on the requirements evolution diagram (see the next paragraph) the probabilities that have been found can be used directly.

As mentioned earlier it is possible that techniques can mutually exclude. For features this would mean it is not possible to include one feature as long as the other exists in the system. For techniques this is somewhat different. When two techniques are mutually exclusive this means they can not coexist in one system, but it is possible to switch between techniques that are used to implement the system. This is called *technology switching*. This is possible at any point of the evolution process, but the costs will become higher at the later stages. This consequence should be handled by the reward-function, however.

6.3.3. Definition of the dynamic program

The decisions that are made in an evolution process are done at states that occur in evolution scenarios. This makes it possible to use the requirements evolution diagram as the base for the dynamic program. In essence the requirements evolution diagram also is a dynamic program, but only one choice can be made from each state. It is a dynamic program for one technique.

When several preparation techniques have been identified, from every state (except end-states) in the requirements evolution diagram any of these techniques can be chosen. For every single technique the same features can be demanded by the customer:

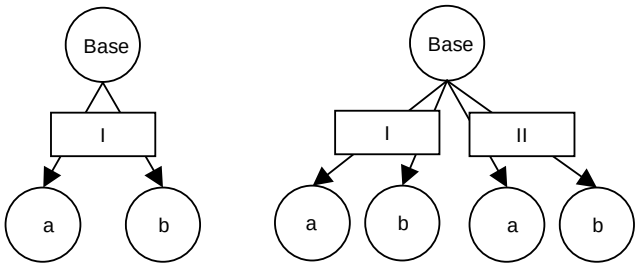


Figure 18: Dynamic program with one technique and one with two

Note that the states *a* and *b* in the second dynamic program for technique *I* are different than the ones for technique *II*. As described above these states represent the situation that a feature has been asked for once the previous requirements have been implemented with the chosen technique. For the states *a* and *b* of technique *I* this means the base has been implemented using technique *I* and *a* or *b* has occurred respectively. For the other two the base has been implemented using technique *II* which leads to a different situation in the evolution process.

When the requirements evolution diagram is taken as the base for the dynamic program a Cartesian product needs to be performed to represent all the possible configurations that can occur. The following methodological approach can be applied for defining the dynamic program:

1. Set the start-state of the requirements evolution diagram as the start-state of the dynamic program;
2. The actions that can be chosen from this state are the techniques that have been identified;
3. For every state that can be reached from this state in the evolution diagram, an equivalent state is made for every technique (except for end-states);
4. The next-states are added to the possible next-state set of the actions of this state;
5. Step 2, 3, 4 and 5 are repeated for the states that have been generated in the dynamic program until no more new states occur;
6. Add the total end-state to the dynamic program.

When this approach is applied is applied to the following evolution diagram this would leave the following result (when two mutually exclusive techniques have been identified):

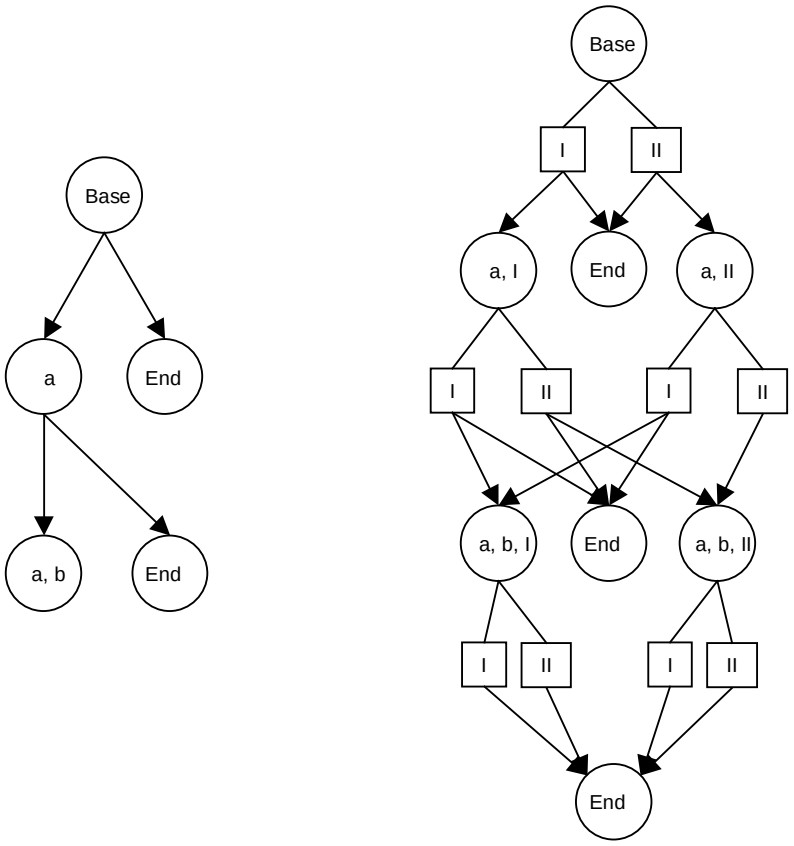


Figure 19: An evolution diagram and its dynamic program

First the start-state of the evolution diagram (base) is identified as the start-state of the dynamic program. Because two techniques have been identified they will be the possible actions from the start-state. The third step defines the states that can be reached from this state. In the evolution diagram two different states can be reached: *a* and *End*. First *a* needs to be defined for the different techniques: the states *a, I* and *a, II* are defined and assigned to the possible next states set of *I* and *II* respectively. The second state that can be reached, *End*, is and end-state. This means it is not necessary to define it for every technique. It can included directly in the possible next-states set of both techniques.

This process now is repeated for the states that have been defined in the first iteration: *a,I* , *a,II* and *End*. The techniques once again are *I* and *II*, which means the actions are the same as the previous state. The equivalent state (the state without the technique) in the evolution diagram for state *a,I* is *a*. The possible next states for this state therefore should be based on *a,b* and *End*. Once again states need to be added for every technique: *a,b,I* and *a,b,II*. These states are added to their respective techniques, and the end-state is done in the same way as the previous iteration. The state *a,II* can use the states that have been defined *a,b,I* in this case.

When the iteration is done again, no new states are added to the dynamic program. At this point the dynamic program needs to be added. The total end-state can be reached from any state where all possible features have been demanded, in this case *a,b,I* and *a,b,II*. As was mentioned before a state in the dynamic program represents a system where one feature still has to be implemented, which in this case means feature *b*. This can be done using either technique *I* or *II*, but the result will always be the same from a software evolution perspective: the evolution is completed. For the calculation of costs this last state is necessary to include, however.

One aspect that deserves special attention is the initial costs of a technique. Because the reward-function has to be able to calculate the costs of an action based only on the current state and the chosen action additional information is needed. The information that needs to be included in a state is whether the engineer has chosen the specific action before (which would mean the initial costs have not to be paid again). This can be done by dividing the action into two different actions: one with initial costs and one without initial costs. When the possible actions for a state are defined only one of these two can be present. The amount of states will increase because of this step, because states can only be reused by different paths when they are equal in features, techniques and possible actions. This is also logical because the amount of possible configurations increases because of techniques with initial costs.

In a more algorithmic approach the methodology of defining the dynamic program can be described as follows:

```
CalculateNextStates(Dynamic Program State, level)
{
	Identify the applicable technique set;
	Identify the possible next states from the evolution diagram;
	For every possible next state from the evolution diagram
	{
		If the next state is not an end-state
		{
			For every technique in the technique set
			{
				Define a Dynamic Program state;
				Set the feature-set of the DP State to the feature set of
				the current state of the evolution diagram;

				Set the technique of the DP State to the current
				technique;
			}
		}
		else
		{
			Define an End-state for the level;
		}
	}
	Add the unique LevelEndState to each action;
	If the defined dynamic program state is unique, add it to the level + 1;
}

Main

For all techniques
{
	If technique has an initial cost
	{
		Add a technique with the same properties but without the initial cost;
	}
}


Retrieve the start-state from the customer evolution diagram;

Define the Start-state of the Dynamic Program;

CalculateNextStates(Dynamic Program Start-state, 0);

While a next level with states is generated
{
	For each state in the new level
	{
		CalculateNextStates(Current Dynamic Program State, new level);
	}
}
```

This algorithm declares a function that is able to calculate next states for any state in the dynamic program that already has been defined except for its actions and next states. The

Main part first defines the start-state and then kick-starts the process for every next level that is calculated.

The process of calculating the dynamic program is very straight forward and can easily be automated. Normally calculating a requirements evolution diagram or a dynamic program will be done by a computer. Included with this thesis is a prototype that is able to calculate evolution diagrams and dynamic programs from feature definitions and techniques. It will be described in more detail in Appendix A.

6.3.4. Probability-functions

The probabilities that have been defined for the requirements evolution diagram can be reused for the dynamic program. Every transition in the evolution diagram has one or more equivalent transitions in the dynamic program, for which the probabilities can be transferred.

It is possible to define these probabilities using a probability-function. Probability-functions resemble reward-functions but return a set of states and their probability when in a state S_x action A_y is chosen. A probability-function will always have the following declaration:

Probability(S_x, A_y)

This function will return the probabilities for reaching the states that are possible for action A_y.

As with cost- and reward-functions these probability-functions should be tailor-made for every single project that needs to be analysed. The inputs that influence the probabilities of a customer choosing a specific feature are from outside the system. The relevance of the probability-functions determines the accuracy of the model in terms of costs that can be expected. It is possible to have a very accurate cost-model for analysing software evolution, but when the probability-model is inaccurate, the results will also be inaccurate.

6.4. Modelling cost aspects

The software evolution analysis model is aimed at finding the best possible way of preparing a system for evolution impact with the respect the total costs that can be expected. Costs are therefore the single most important criterion that need to be identified. This paragraph will describe how costs and cost-functions can be defined for the dynamic program model and how decisions affect costs for implementing new customer requirements.

6.4.1. Definition of the reward-function

The final part of the dynamic program that needs to be defined is the reward-function. Earlier for the use of techniques a cost model has been proposed, on which the reward-function can be based. The contribution the use of a certain technique has to the total costs can be divided into two parts: the initial costs and the application costs.

The initial costs of a technique (a course, investing in development software, etc.) normally only have to be paid once and do not depend on the complexity of the software project. Initial technique costs therefore can be treated as a constant that is added to the result once, in an evolution scenario where this technique is chosen.

The application of a technique when a feature needs to be implemented is not a constant, but varies depending on the complexity of the change. The cost-model that has been chosen for the calculation is based in the properties that have been stored for features and techniques. This makes it possible to calculate the reward of choosing a certain action from a specific state, because all the relevant information is available. The feature that was asked for, the features that already have been implemented and the techniques that were used are all present in the current state. When a technique is chosen the new feature is implemented using the this technique. The reward-function is capable of determining whether a technology switch occurs (see next sub-paragraph) and which complexity occurs based on the new and existing features. And by applying the chosen cost-model with these parameters the reward for this action can be determined.

It is not possible to define a methodological approach on how this reward-function should be defined since this depends on the cost-model that has been chosen for cost-calculations. The reward-function, like the probability-function, should be tailor-made for each analysis. For small problems it is possible to define a lookup-table containing the value for every possible combination of states and actions, but for large problems it might be better to use a proper reward-function that can calculate rewards based on the available information.

6.4.2. Technology switching

In a state of a dynamic program it is possible to choose from a set of actions. The techniques that are associated with these actions can be compatible or might even require each other. But it is also possible that the selected technique can not coexist with another technique the system already includes at that point. This makes it necessary to reimplement the parts that incorporate the excluded technique. This concept is named *technology switching*. For the selected technique this means that the costs are raised because of the extra effort of reimplementing system parts that already had been implemented.

Although it seems illogical to consider these kinds of technology switches, they should be considered carefully in specific situations. For instance, when a technique is selected that is not available at the moment (e.g. because it is still in a testing phase) it is not possible to choose the technique until some later state. With the dynamic program approach it is possible to determine whether the technology switch is still feasible from state, and when it is not it can determined from which state the switch would be within the project’s budget.

Technology switches can be quite attractive at early stages when little redefinition is needed when a switch is selected. When a technique offers very good performance with features that can be expected from a state, the reward for this technique might even be lower than when a technique is chosen that is capable of co-operating with the existing system implementation. When the initial costs for a technique are raised this means that the reward for choosing this technique also increase. But when the same technique is also capable of reducing implementation costs better than other techniques from states in a lower level, the increase will also apply to the expected minimal rewards of these states.

When the initial costs are raised to a level where the rewards of choosing this technique in states of lower levels will be higher than the rewards of other techniques, the rewards will become constant from this point on.

6.4.3. The reference scenario: total reimplementation

For evolution processes a reference scenario exists that represents the scenario when no attention is paid to evolution aspects. This is the scenario where every new feature needs to be "hacked" into the system because it has been optimised for the requirements known at the start of the scenario. Every time a new feature is asked for, the effort of implementation increases because a larger system needs to be reimplemented.

This scenario will undoubtedly lead to the highest possible cost for an evolution scenario where it is likely that new requirements occur. However, when the probability for requirements changes is very small it is likely that this scenario offers the best solution from a cost perspective. No attention has to be paid to difficult preparation techniques, only the initial system is implemented.

This scenario will be referred to as total reimplementation and can be calculated as the highest-risk technique that can be chosen in an evolution process. Because risk in evolution is caused by not preparing a system for evolution when it might occur this scenario can be used as a tress-hold for the determination if a system should be prepared at all.

6.5. Using the dynamic program for evolution optimisation

6.5.1. Choosing the start-state(s)

Solutions of dynamic programs can be calculated on a per state basis. For each state in a dynamic program the optimal decision can be determined, even if the specific state might never be reached when higher level optimal solutions are considered.

For evolution analysis this property of dynamic programming makes it possible to analyse the process of software evolution scenarios to a great extend. The main goal of the analysis is to determine which decisions will provide the most cost-effective approach towards preparing a system for evolution. But when the modelling is accurate enough it is possible to simulate evolution scenarios to gain a better insight in the process that is modelled.

It is possible to choose any state and determine what the best choice would be from that state. The computations that are needed to determine the optimal decision for this state can be limited to solving only the subtree that is relevant for this state (only the states that can be reached from this state). This subtree will be a complete dynamic program on which for instance the backward inductive algorithm can be applied.

Choosing a start state in this respect is a matter of which scenarios and what kinds of feedback are interesting for the engineer. When an advice is needed which decision to make, the computations can be optimised for speed. For such a computation it is best to identify the state of interest before the computation, because then only the subtree that can be reached from that state needs to be solved.

When an extensive analysis is desired, it is best to solve the entire dynamic program and store the results. These results can be used to simulate evolution scenarios and inspection of specific states that can be reached in the dynamic program. The computations that are involved in this will be more complex since the entire dynamic program needs to be calculated and solved. And since most of the time there is a state-space explosion for typical projects that need to be

analysed these computations might go beyond the capabilities of the person or computer that should perform the analysis.

6.5.2. Finding the optimal development policy

Once the entire dynamic program has been defined and a start-state has been chosen, the dynamic program can be solved. By applying a technique such as the backward inductive algorithm the best choice and its related cost are determined and stored. The result of this computation is that for every single state in the (relevant) dynamic program the best action has been determined and the costs that can be expected for this action.

For every step in the possible evolution scenarios that have been defined by the requirements evolution diagram (on which the dynamic program is based) the optimal decision has been identified. These results can now be used as an advice in determining the optimal software development policy. The results can serve as a validation for deciding whether or not to support evolution with selected techniques.

A generalised result can be calculated for initial costs of techniques. The software evolution analysis model can be applied to determine if the project will have a pay-off for investing in courses, etc. For these kinds of techniques a break-even point occurs after which the initial costs of a technique exceed the expected reduction caused by this technique.

This can be determined by making the initial costs of a technique variable. The expected costs for an optimal decision can now be calculated as a function of the initial costs. This function will have a linear character, because initial costs are added once to the total reward of an action. The function that is determined can be plotted into a graph from which the break-even point can be determined. Raising the initial costs might also raise the expected values for other actions in a state. This is because of the point at which a technology switch still is feasible to the project budget.

Solving a dynamic program is a very labour-intensive practice that normally will be automated. Because of the size of such computations an example will not be given here. In chapter 7 an example case will be defined for which the results and a generalised solution will be computed.

6.6. Summary

In this chapter a methodological approach is based on dynamic programming was defined to determine the expected costs for evolution scenarios and optimal decisions for software evolution scenarios.

Dynamic programming is based on Markov-decision processes and forces the decision to be made based on knowledge that is present in the current state. This chapter has presented a way of modelling software evolution as a dynamic program based on the states that have been identified for the requirements evolution diagram. The choices an engineer has during a project can be mapped to techniques that are available to the engineer to prepare for evolution impact.

By assigning cost models to the usage of preparation techniques it becomes clear which technique suits a certain situation best. The software evolution analysis model can calculate any configuration (a set of features that have occurred, and that have been implemented with a set of techniques) that can occur within an evolution process from the engineer's point of view. Based on the probabilities of these features occurring, the software evolution analysis model is capable of determining which technique(s) will have minimal expected costs.

This way the software evolution analysis model is capable of identifying the best decision based on the assumptions that were done for the analysis. Because of the apparent state-space explosion also the backward-inductive algorithm was mentioned. This algorithm makes it possible to reduce computing effort for solving dynamic programs that contains paths that visit a limited amount of states.

7. Case study: The Email system

On this view, the capacity of enduring the most different climates by man himself and by his domestic animals, and such facts as that former species of the elephant and rhinoceros were capable of enduring a glacial climate, whereas the living species are now all tropical or sub-tropical in their habits, ought not to be looked at as anomalies, but merely as examples of a very common flexibility of constitution, brought, under peculiar circumstances, into play.

Charles Darwin, the origin of species

7 Case study: the Email system

Application of the model to an example case

To illustrate the use of the software evolution analysis model it will be applied to an example case in this chapter. A number of assumptions and restrictions is done to simplify the analysis with respect to the possible state-space explosions that can occur.

7.1. Introduction

In this chapter a description of an example case will be given that will be used with the software evolution analysis method, and is based on the e-tutorials "*Software evolution problems in case of inheritance and aggregation based reuse*" and "*Solving the evolution problems using Composition Filters*" on software evolution problems, by Mehmet Aksit and Lodewijk Bergmans. These and other e-tutorials can be found at: http://trese.cs.utwente.nl.

The case that is described in the e-tutorials exemplifies the problems that can be encountered when new requirements occur. A start-system is defined and several possible extensions are identified. Based on the amount of methods that need to be reimplemented, the system impact is determined. Several different approaches are analysed (inheritance, aggregation, composition filters) and conclusions are drawn from the results.

The case for the software evolution analysis method should be more refined, to better illustrate the use. From the e-tutorials it becomes clear that for the specific case it is always best to use composition filters for implementing possible new requirements. This means the software evolution analysis method is not necessary for project configuration and reusability trade-off.

Therefore in the case a description will be given of an implementation group that has a certain amount of knowledge on implementation techniques. In order to work with for instance composition filters, the group must acquire the relevant knowledge, which will take a certain amount of time, and thus costs will be raised. This is a reasonable assumption, since not always all relevant knowledge is present and the main concern is whether or not to invest in a course on a certain technology.

7.2. The Email system

7.2.1. Case situation description

A software development company *ImpGroup* acquires a contract on the implementation of an email system for the company *Customer*. ImpGroup is fairly experienced in object-oriented programming and the requirements that are posed by Customer do not seem to pose any significant problems.

The email-system will be part of a bigger system that consists of the email-system, browsing functionality and a programming environment. However, these parts do not interfere with each other, they are independent.

ImpGroup suspects, however, that the email-system might need to be extended in the near future. Because of this a domain analysis is done to identify the possible future extensions. This leads to the following results on possible system extensions:

E-mail system

The e-mail system is the base system, for which ImpGroup has acquired the contract. This system can be characterised as a simple e-mail system. The system should be able to define, send and receive e-mails. Furthermore should it be possible to determine the route, approve e-mails and reply to messages. The following interface is defined for the e-mail system:

```
Class Email interface
putOriginator(anOriginator);
getOriginator returns anOriginator;
putReceiver(aReceiver);
getReceiver returns aReceiver;
putContent(aContent);
getContent returns aContent;
send;
reply;
approve;
isApproved returns Boolean;
putRoute(aRoute);
getRoute returns aRoute;
deliver;
isDelivered returns Boolean;
```

Figure 20: Interface specification for the Email class

USViewmail

The first possible extension that was identified, is USViewmail. USViewmail restricts the methods of the interface that can be used based on the entity-identity, being either a User or the System. For instance a user cannot approve an email, and the system cannot read the content of an email. The methods of Email can be divided into three different groups: methods not available for the user, methods not available for the system and methods that can be accessed by both. The following division is made:

Not available when in Userview:
- approve
- putRoute
- deliver

Not available when in Systemview:
- putOriginator
- putReceiver
- putContent
- getContent
- send
- reply

All remaining methods are available in both views.

ORViewmail

ORViewmail extends the system with distinguishing between the Originator of a message and the receiver of a message. For instance it should not be possible for a receiver to determine the content of the email that is send to him. Note however, that this extension is independent of the USViewmail extension. The email system can exist with the ORView extension but without the USView extension. The ORView system divides the methods in the following manner:

Not available when in Originatorview:
- reply

Not available when in Receiverview:
- putOriginator
- putReceiver
- putContent
- send

All remaining methods are available in both views.

Warning2Mail

Warning2Mail is a proper extension of USViewmail and therefore requires USViewmail to be present in the system. Warning2Mail raises a warning whenever in the userview the same method is invoked for a single mailobject two times or more subsequently. For instance it should not be possible to invoke the method *send* twice in a row, because a message should only be sent once.

LockingMail

LockingMail extends the system independently, so no other extensions need to be present in order to implement LockingMail. LockingMail introduces a locking mechanism to the mail-object. When the email-object is locked, all method-invocations on the object are queued until the object is unlocked. When the object is unlocked, all methods in the queue are executed, and new invocations will be executed instantly.

DynamicMail

DynamicMail once again is an independent extension of the base system. DynamicMail adapts the implementation of the method *send* based on the type of data that is to be sent. For instance DynamicMail will change the protocol of sending an email to a dedicated video-protocol when a video is being sent.

These are the identified possible extensions for the email system, but it is not sure if and when any of the extensions will be asked for. It is possible that none of the extensions are asked for, all of them or any combination. The only thing that is certain, is that when Warning2Mail is asked for USViewMail also needs to be implemented. It is possible however, to define probabilities for the occurrence of the extensions.

7.2.2. Case problem description

ImpGroup was somewhat worried by the results of the domain analysis. The five possible extensions all have their own specific impact on the system, and it might be difficult to solve the problems that occur with the present knowledge about inheritance and aggregation. Recent articles on Composition Filters have sparked the attention of the management and it is possible to follow a course on Composition Filters for the implementation team. Following the course takes time and costs money that will need to pay of during the project.

Another problem is, that ImpGroup is low on money. Investing in a course on Composition filters when it does not pay off, might seriously damage the company. Because of this, the objective in this project is to maximise reuse with the lowest possible cost.

Management now wants to know what the best action would be, based on the probability of the extensions occurring and the possible costs. What is the chance that the investment of a course on composition filters will pay of? Should the management rely on the knowledge of the implementation group? Should the implementation group prepare for implementation based on inheritance, aggregation or not prepare at all?

7.2.3. Assumptions on the problem domain

In order to limit the eventual state- and action-space somewhat, the following assumptions are made on the problem space:

Extensions and serializability

Although the extensions can occur at any moment in time and in any order, for the case it will be assumed that this happens in a serial way, which means that only one extension will be asked for at a time. This can be explained in the following way: when a set of extensions is asked for, this can always be mapped to a sequence where one extension is demanded at a time. The order in which this happens is not important, because the Software Evolution analysis method analyses all possible orders.

Techniques

In the case, four techniques of implementation will be evaluated:

- Reimplementation when necessary, which means the implementation group will optimise the implementation for the known requirements but will pay no attention to possible future extensions. Intuitively this would be the best solution, when no extensions are asked for because no effort is done to make the system reusable. All effort is put into the relevant system functionality.
- Inheritance structures, which means that the object-structure will be defined in such a way that is possible to reuse system functionality as much as possible with inheritance relations.
- Aggregation structures, which means that the object-structure will be defined in such a way that is possible to reuse system functionality as much as possible with aggregation relations.
- Composition filters, which means that a reusable base is defined and composition filters are used to handle the functional requirements of the extensions that occur.

It is assumed that these techniques cannot coexist in the system, although this might be possible in practice. The only reason why this is done, is to reduce the state-space of the software evolution analysis method. For the model to be more accurate, this assumption can be dropped, and the model can be adapted. The computations will become larger in size, however. A direct consequence of this assumption is, that after a certain technique is chosen, choosing another technique means reimplementing the entire system. This concept will be referred to as a technology-switch.

Costs

Reusing system parts means that as little as possible is to be changed in the existing system before it can be used with the new part that fulfils the new requirements. For the case a cost model will be assumed that is based on the methods in the email system. The amount of methods that need to be reimplemented because of an extensions and the new methods that need to be implemented will be used to calculate the cost of extending the system. Furthermore following the composition filter course can raise costs. It is possible to distinguish between different types of methods based on the complexity (forwarding, new code, etc.), to make the cost model more realistic.

Probabilities

It is assumed it is possible that the probability of extensions occurring can be adequately described with constant probabilities, for instance P(USViewMail) = x, where $0 \leq x \leq 1$. A consequence of these assumptions is that when there is an infinite time frame in which the system will be used, all extensions will be asked for. This would justify implementing all extensions from the start, since they all will occur. Because of this it is also assumed the system will be used for a finite time.

7.3. Customer requirements analysis

7.3.1. Definition of the feature tree

The E-Mail system is defined as a part of a larger system, that also contains browsing functionality and a programming environment. These parts do not interact with each other and therefore are very independent towards each other. They might share an interface with which it is possible to open the desired functionality, and switching between applications can be more efficient. Extensions to any part of the system will not affect the other parts, which means they can be analysed for evolution independently. The main system can be represented in a feature tree in the following way:

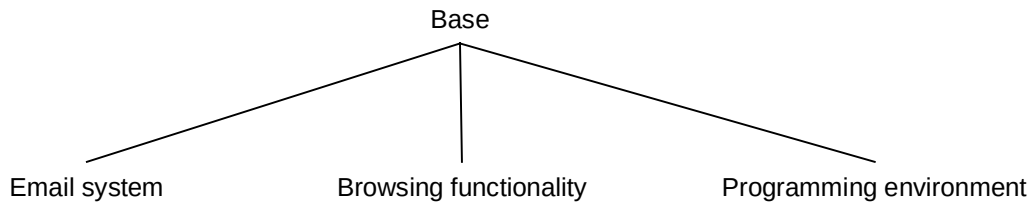


Figure 21: Feature tree for the overall system

Because ImpGroup should implement the E-Mail system this is the part that is of particular interest. In the case description five possible extensions were identified for the E-Mail system: USViewMail, ORViewMail, LockingMail, DynamicMail and Warning2Mail. Most of these extensions are independent towards each other except for Warning2Mail, which requires a user-system view (USViewMail) to be present when it is implemented. The feature-tree that represents the relations between to possible extensions and the existing E-Mail system will look like this:

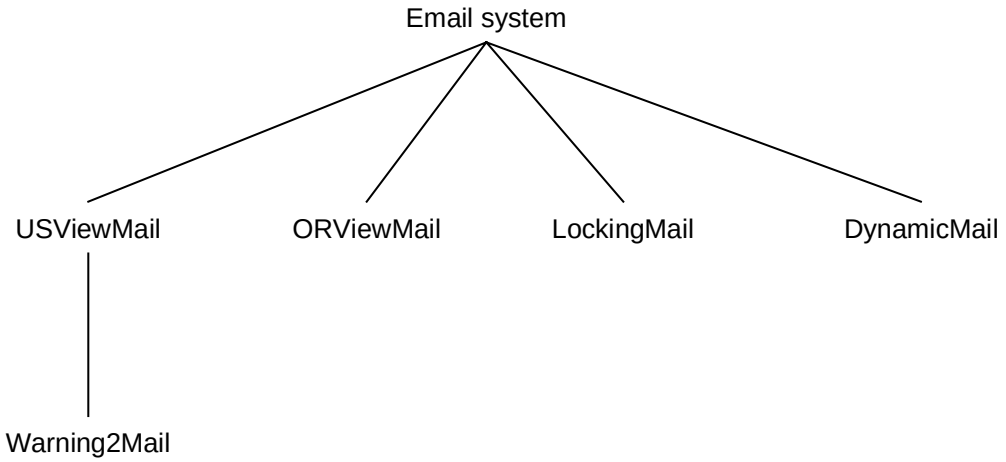


Figure 22: Feature tree for the Email system

In the feature-tree the independency of most of the features can be seen, because they only require the E-Mail system to be present. Only Warning2Mail requires USViewMail to be present which can be seen by the specialisation-relation. Each of the features can asked for as an extension and because the entire E-Mail system is needed by every extension the E-Mail system can be represented as a single feature. The example case offers no complex relations between several features, but there is sufficient complexity to demonstrate the analysis model.

7.3.2. Requirements evolution diagram extraction

From the feature tree the requirements evolution diagram is extracted. Because the features are relatively independent towards each other, it is only necessary from a state to determine which feature was asked for last.

The initial state for the requirements evolution diagram is the E-Mail system. The algorithmic approach is used to determine the possible next states of the initial state. The set of possible features is set to:

{ USViewMail, ORViewMail, LockingMail, DynamicMail, Warning2Mail }

For each feature in the featureset it is checked whether a next state can be made. The first feature is *USViewMail*. Since USViewMail has no requirements of other features being present, it is possible to make a next state that contains *{ E-Mail, USViewMail }*. This next state also might have possible next states, but these will be calculated at the second pass.

This process can be repeated for *ORViewMail, LockingMail* and *DynamicMail*. The next states that come from these additions are *{ E-Mail, ORViewMail }, { E-Mail, LockingMail }* and *{ E-Mail, DynamicMail },* and they are all unique.

The final feature that needs to be tested is *Warning2Mail*. When Warning2Mail is added USViewMail needs to be present. But in the current state (the initial state) only the E-Mail system is present. This means that it is not possible to choose Warning2Mail from this state. The final state that can be reached from the initial state is the premature end-state. This state has no possible next states and will contain zero features for this level. This state will be represented as EndStateLevel0.

The states that can be reached from the initial state have now all been calculated, and the requirements evolution diagram up to now looks like this:

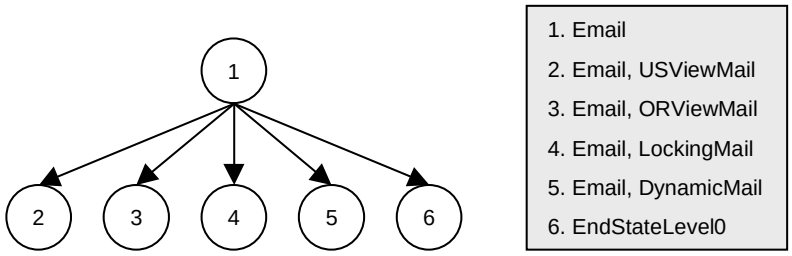


Figure 23: Evolution diagram of the initial level

The second pass will be to complete the states that have been defined for Level 1. For each state the features that it contains already have been defined, only the possible next states need to be calculated.

The first state is *{ E-Mail, USViewMail }*. The set of possible features becomes:

{ ORViewMail, LockingMail, DynamicMail, Warning2Mail }

The following states can be reached:

{ E-Mail, USViewMail, ORViewMail }
{ E-Mail, USViewMail, LockingMail }
{ E-Mail, USViewMail, DynamicMail }
{ E-Mail, USViewMail, Warning2Mail }

For this state it is possible to add *Warning2Mail* to the set of features because USViewMail is present in the current state. The final state that needs to be added is the premature end-state for this level, which contains one feature and from which it is not possible to ask for any of the remaining features. This state will be referred as EndStateLevel1.

The second state is *{ E-Mail, ORViewMail }*. The set of possible features becomes:

{ USViewMail, LockingMail, DynamicMail, Warning2Mail }

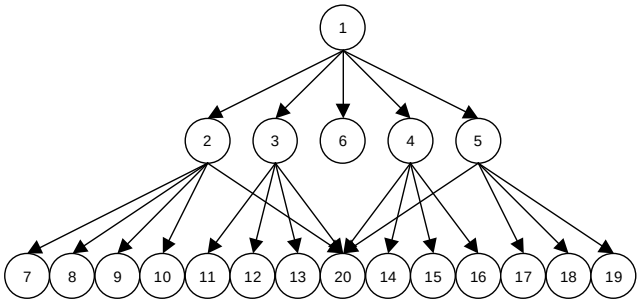
The process can be repeated and results in the following states:

{ E-Mail, ORViewMail, USViewMail }
{ E-Mail, ORViewMail, LockingMail }
{ E-Mail, ORViewMail, DynamicMail }

From the states that have been calculated *{ E-Mail, ORViewMail, USViewMail }* is unequal to *{ E-Mail, USViewMail, ORViewMail }* because order is important for the last feature. This

means that *{ E-Mail, ORViewMail, USViewMail }* will needs to be included as a unique state. The premature end-state for the current state is the same as the one that was used for *{ E-Mail, USViewMail }* and can be reused.

When this process has been repeated for every state in Level 1, the following evolution diagram can be drawn:



1. Email
2. Email, USViewMail
3. Email, ORViewMail
4. Email, LockingMail
5. Email, DynamicMail
6. EndStateLevel0
7. Email, USViewMail, ORViewMail
8. Email, USViewMail, LockingMail
9. Email, USViewMail, DynamicMail
10. Email, USViewMail, Warning2Mail
11. Email, ORViewMail, USViewMail
12. Email, ORViewMail, LockingMail
13. Email, ORViewMail, DynamicMail
14. Email, LockingMail, USViewMail
15. Email, LockingMail, ORViewMail
16. Email, LockingMail, DynamicMail
17. Email, DynamicMail, USViewMail
18. Email, DynamicMail, ORViewMail
19. Email, DynamicMail, LockingMail
20. EndStateLevel1

Figure 24: Evolution diagram of the second level

Each level can be calculated this way and eventually no new levels will arise since it is no longer possible to add a remaining feature. When all the levels have been calculated the following evolution diagram has been computed:

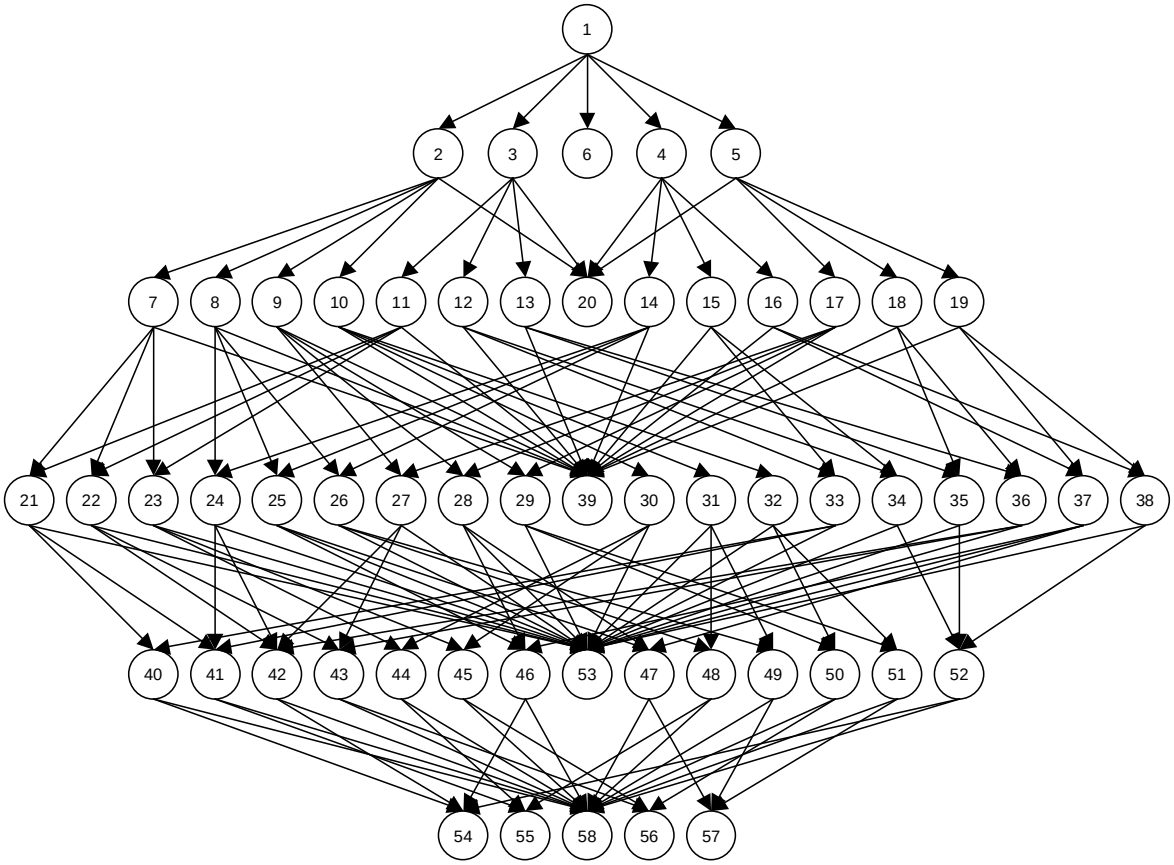


Figure 25: 1-ordered evolution diagram

1. Email
2. Email, USViewMail
3. Email, ORViewMail
4. Email, LockingMail
5. Email, DynamicMail
6. EndStateLevel0
7. Email, USViewMail, ORViewMail
8. Email, USViewMail, LockingMail
9. Email, USViewMail, DynamicMail
10. Email, USViewMail, Warning2Mail
11. Email, ORViewMail, USViewMail
12. Email, ORViewMail, LockingMail
13. Email, ORViewMail, DynamicMail
14. Email, LockingMail, USViewMail
15. Email, LockingMail, ORViewMail
16. Email, LockingMail, DynamicMail
17. Email, DynamicMail, USViewMail
18. Email, DynamicMail, ORViewMail
19. Email, DynamicMail, LockingMail
20. EndStateLevel1
21. Email, USViewMail, ORViewMail, LockingMail
22. Email, USViewMail, ORViewMail, DynamicMail
23. Email, USViewMail, ORViewMail, Warning2Mail
24. Email, USViewMail, LockingMail, ORViewMail
25. Email, USViewMail, LockingMail, DynamicMail
26. Email, USViewMail, LockingMail, Warning2Mail
27. Email, USViewMail, DynamicMail, ORViewMail
28. Email, USViewMail, DynamicMail, LockingMail
29. Email, USViewMail, DynamicMail, Warning2Mail
30. Email, USViewMail, Warning2Mail, ORViewMail
31. Email, USViewMail, Warning2Mail, LockingMail
32. Email, USViewMail, Warning2Mail, DynamicMail
33. Email, ORViewMail, LockingMail, USViewMail
34. Email, ORViewMail, LockingMail, DynamicMail
35. Email, ORViewMail, DynamicMail, LockingMail
36. Email, ORViewMail, DynamicMail, USViewMail
37. Email, LockingMail, DynamicMail, USViewMail
38. Email, LockingMail, DynamicMail, ORViewMail
39. EndStateLevel2
40. Email, USViewMail, ORViewMail, LockingMail, DynamicMail
41. Email, USViewMail, ORViewMail, LockingMail, Warning2Mail
42. Email, USViewMail, ORViewMail, DynamicMail, LockingMail
43. Email, USViewMail, ORViewMail, DynamicMail, Warning2Mail
44. Email, USViewMail, ORViewMail, Warning2Mail, LockingMail
45. Email, USViewMail, ORViewMail, Warning2Mail, DynamicMail
46. Email, USViewMail, LockingMail, DynamicMail, ORViewMail
47. Email, USViewMail, LockingMail, DynamicMail, Warning2Mail
48. Email, USViewMail, LockingMail, Warning2Mail, ORViewMail
49. Email, USViewMail, LockingMail, Warning2Mail, DynamicMail
50. Email, USViewMail, DynamicMail, Warning2Mail, ORViewMail
51. Email, USViewMail, DynamicMail, Warning2Mail, LockingMail
52. Email, ORViewMail, LockingMail, DynamicMail, USViewMail
53. EndStateLevel3
54. Email, USViewMail, ORViewMail, LockingMail, DynamicMail, Warning2Mail
55. Email, USViewMail, ORViewMail, LockingMail, Warning2Mail, DynamicMail
56. Email, USViewMail, ORViewMail, DynamicMail, Warning2Mail, LockingMail
57. Email, USViewMail, LockingMail, DynamicMail, Warning2Mail, ORViewMail
58. EndStateLevel4

Figure 26: Legend for the evolution diagram

7.3.3. Probability modelling

The probability for reaching next states from a certain state can be defined on a per state base. For instance the following probabilities can be defined for the initial state:

Possible next state	Probability
{ E-Mail, USViewMail }	P(USViewMail)
{ E-Mail, ORViewMail }	P(ORViewMail)
{ E-Mail, LockingMail }	P(LockingMail)
{ E-Mail, DynamicMail }	P(DynamicMail)
EndStateLevel0	1 - P(USViewMail) - P(ORViewMail) - P(LockingMail) - P(DynamicMail)

These probabilities can be changed for every state, but it is also possible to have a probability model that uses constant values. When the probabilities can change over time a probability-function can be written that returns the probability for the next state from a specific state in the requirements evolution diagram. The way the probabilities are defined determines the accuracy of the customer behaviour representation.

For this example case a very simple probability model will be used where the probabilities will remain constant. These probabilities can be applied for every single state in the evolution diagram:

Feature	Probability
USViewMail	0.2
ORViewMail	0.2
LockingMail	0.2
DynamicMail	0.2
Warning2Mail	0.2
None	0

The table above defines the probability of Warning2Mail because from a customer point of view it can be asked directly. However, when Warning2Mail is asked for USViewMail might not be present. When this occurs the probability of USViewMail has a corrected value. This leads to the following corrected probability values:

Feature	Probability
USViewMail	0.4
ORViewMail	0.2
LockingMail	0.2
DynamicMail	0.2
None	0

7.4. Possible evolution preparations

Four different techniques were identified in the case description. In this paragraph a closer look will be taken at these techniques and a cost-model will be defined for the application of these techniques in the Email system.

7.4.1. Inheritance analysis

A lot of features extend the functionality of the base E-Mail system which makes to use of Inheritance logical. The technique can redefine functions that need to be redefined and others can be handled by super calls. For instance for USViewMail this will look like this:

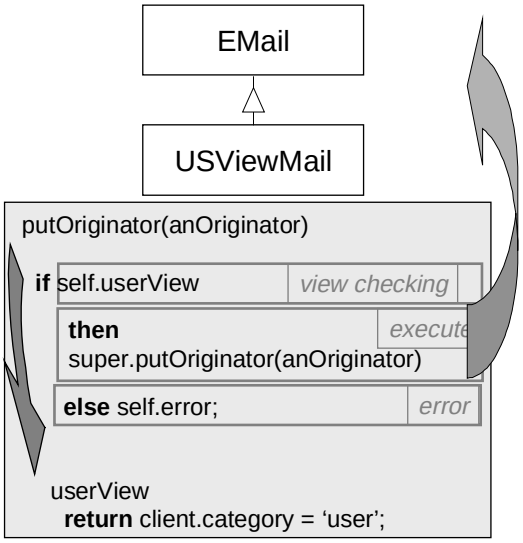


Figure 27: Supercall-structure for inheritance

But there are aspects of inheritance structures that might make the technique less applicable. Using this technique, extensions can only be done compile-time. Upgrading the system would mean stopping the existing system. This could cause costs when the system should be running all the time.

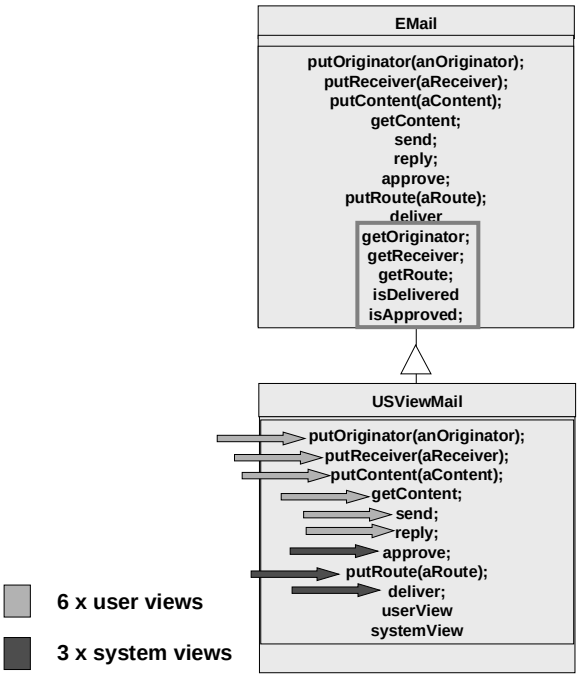
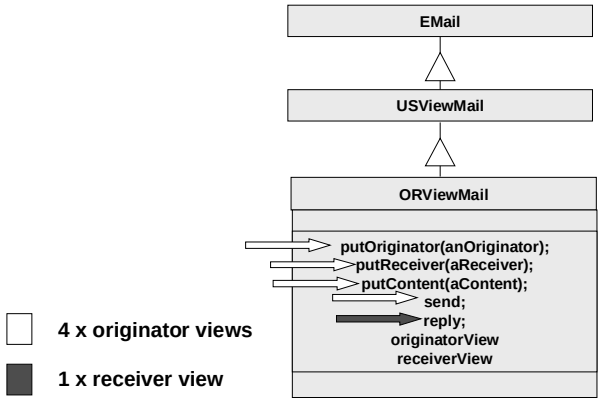


Figure 28: Redefinitions for inheritance

The amount of redefinitions that are needed for this extension is 9, but also two methods are needed to support both the views. This means that a total number of 11 method-(re)definitions are needed to implement the USViewMail-feature for the E-Mail system. These methods will be considered equivalent in complexity which means that the effort needed to implement such a method will be almost the same. These types of effort will be called *method redefinitions*.

When a second feature, for instance ORViewMail is asked for by the customer it is possible to inherit from the USViewMail-class and redefine the methods for the ORViewMail-feature. This would result in the following picture:



The amount of redefinitions that is needed now is only 5, and two methods are needed for the views. The inheritance structure provides transitive reuse. The order in which the views are enforced are now fixed to first checking the ORView and then the USView. When the customer wants this order to be different, a total reimplementation is needed.

Figure 29: Redefinitions for inheritance ORViewMail

When the other features are analysed in the same way, the amount of method redefinitions can be identified. The order in which the features are added is of no significant influence on the efforts needed so the following table can be made which includes the efforts needed to implement the new features with inheritance structures (the figures were taken from the e-tutorial [Aksit2001a]):

Feature	No. of method redefinitions	No. of class definitions
USViewMail	11	1
ORViewMail	7	1
LockingMail	16	1
DynamicMail	1 + no. of protocols	1
Warning2Mail	15	1

For Dynamic Mail no specific figure can be given since this depends on the amount of protocols need to be supported. In the e-tutorial is four different protocols are identified, which means that a total number of 5 redefinitions is needed for the implementation of DynamicMail. Although trivial for the example case, it is also necessary to implement the new class for each feature. This is also included in the table

Once again, the order in which the views are enforced, the locking is applied, etc., depends on the order in which the features have been demanded. Changing this order could lead to a total reimplementation of the system.

7.4.2. Aggregation analysis

To address the compile-time adaptability, aggregation can be considered as an alternative. The aggregation structure for USViewMail could look like this:

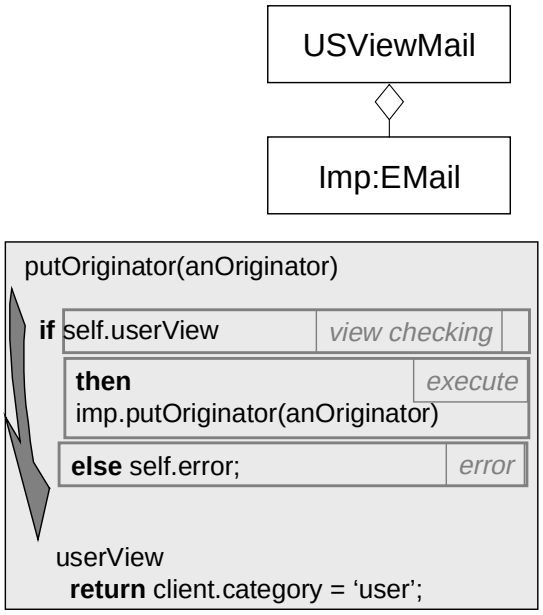


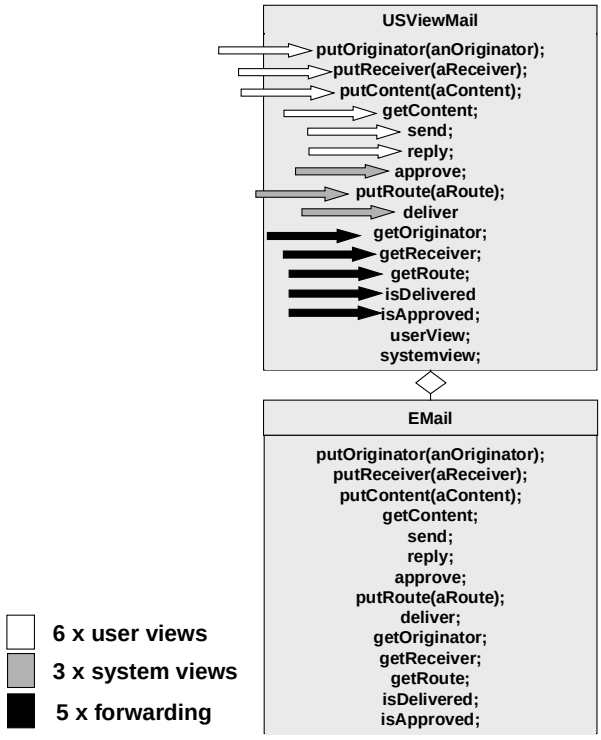
Figure 30: Call-structure for aggregation

When new features are added by using aggregation techniques it is possible to provide run-time adaptability. Aggregation could provide a better solution when the existing system should be running as much as possible. The performance of the application could be affected however.

View checking is provided by a USViewMail-class. The USViewMail-class has an aggregation class E-Mail that provides the needed functionality. The effort that is needed compared to total reimplementation is reduced because the functionality of the E-Mail-class can be used by forwarding after the view has been checked. Methods that are not affected by the new feature however, also need to be forwarded. This is because aggregation can not do any automatic super calls.

The effort that is needed to implement forwarding methods will be called *method forwarding*.

For for instance USViewMail the following redefinitions need to be done:



For this extension 9 redefinitions need to be done and 2 methods need to be implemented to supply the views. But as opposed to inheritance, all remaining features need to be forwarded to the aggregation object. This means that 5 forwarding methods need to be defined. Also one extra class needs to be defined.

Figure 31: Redefinitions for USViewMail and aggregation

When multiple features are analysed it becomes clear that because of the method forwarding, aggregation can not provide transitive reuse. The table of the efforts for the features for aggregation structures will look like this (figures taken from e-tutorial [Aksit2001a]):

Feature	No. of method redefinitions	No. of method forwardings	No. of class definitions
USViewMail	11	5	1
ORViewMail	7	9	1
LockingMail	16	-	1
DynamicMail	5	14	1
Warning2Mail	15	-	1

As with inheritance, switching the order in which the extensions are applied can cause a total system reimplementation. For DynamicMail support for four protocols was assumed.

7.4.3. Composition Filters analysis

ImpGroup has noticed that Composition Filters might be a good alternative for inheritance and aggregation. Composition Filters use a different approach that makes it more indifferent tot the amount of redefinitions that are needed. For USViewMail the Composition Filters approach will look like this:

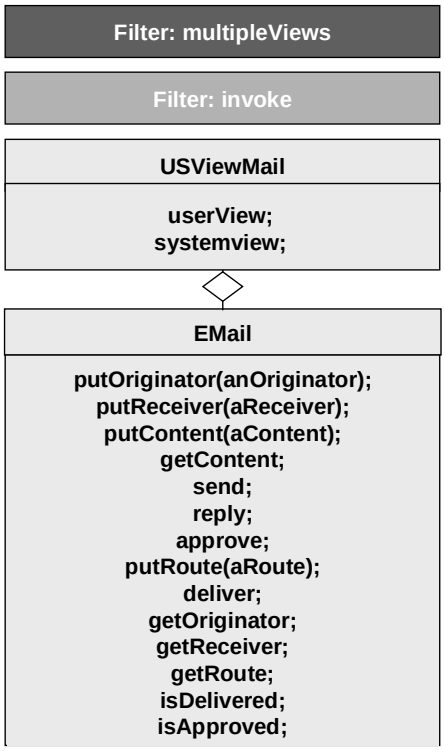


Figure 32: Composition Filters implementation

Composition Filters reduce the efforts that are needed for implementing system extensions by catching messages to objects and applying a filter to them. For USViewMail for instance a Composition Filter checks whether the view is applicable for the method that is called. When the view is correct the message can pass through the filter and call the method. When this is not the case the filter raises an error and the message can not pass through the filter. The amount of methods that need to be (re)implemented is very small compared to the reference scenario, only the methods to enforce the view and an implementation of the filter is needed. Methods that are affected do not need to be redefined because the message that calls these methods will never pass the filter when the proper view was not used.

Composition Filters can be implemented in different ways that makes it possible to support both compile-time adaptability (which increases performance) or run-time adaptability. For USViewMail for instance, a Composition Filter definition could look like this:

The aggregation-structure that is used here is not the same as the aggregation-structure that was discussed earlier since the aggregation is not used to supply the new functionality but the filters. The aggregation-structure makes it possible to provide run-time adaptability, however. For implementing USViewMail a new class needs to be implemented, two methods for providing the views and two filters: *Multiple views* and *Invoke*. The effort that is needed for implementing the filters is different from method redefinition and method forwarding. This type of effort will be called *Filter definition*. A detailed explanation of how Composition Filters provide the desired functionality can be found in [Aksit2001b].

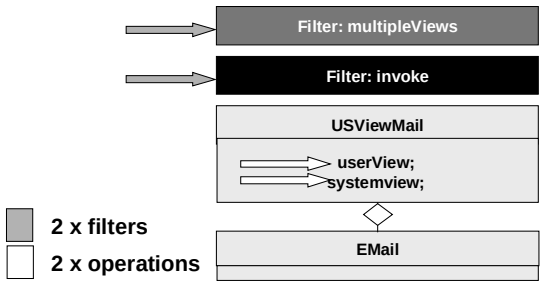


Figure 33: Using Composition Filters for USViewMail

Adding ORViewMail to the system can be done in the following way:

For implementing ORViewMail with Composition Filters one new class is needed, two methods for the views and three filters: *origView*, *recView* and *Invoke*. Based on the e-tutorial the following figures can be identified for implementing the features with composition filters (based on [Aksit2001b]):

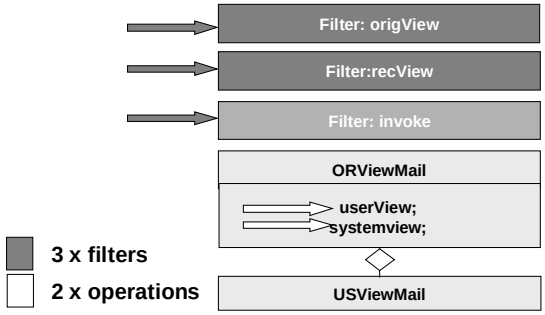


Figure 34: Using Composition Filters for ORViewMail

Feature	No. of method redefinitions	No. of class definitions	No. of filter definitions
USViewMail	2	1	2
ORViewMail	2	1	3
LockingMail	2	1	2
DynamicMail	5	2	2
Warning2Mail	1	2	2

7.4.4. Total reimplementation analysis

This “technique” is included as the reference scenario for the analysis. Total Reimplementation might be necessary in situations that can occur during an evolution process. In the e-tutorials a certain customer demand is mentioned for inverse checking of features that have been added (ordered feature sets). This is not possible for aggregation structures, which means whenever this feature is asked for and the system has been implemented with aggregation structures, a total reimplementation is needed. These kinds of technology switches can be analysed in the software evolution analysis model by choosing a new best solution that involves total reimplementation.

The concept of total reimplementation can be a valid choice in specific situations. When the probability is very low that the customer asks for any feature from a state early in the evolution

diagram, the most cost effective solution might be total reimplementation. Preparing a system for future extensions that most likely will not occur, will only raise costs and never pay off.

The reference technique is simple in terms of method redefinitions. This technique assumes a complete system reimplementation for every feature that occurs. This means that for every evolution step the methods of the initial system need to be reimplemented. Methods that belong to a specific view can be reused. This leads to the following table:

Feature	No. of method redefinitions
USViewMail	16
ORViewMail	16
LockingMail	15
DynamicMail	16
Warning2Mail	19

Although these figures seem identical to the figures that were presented for aggregation, all these efforts are redefinitions. Aggregation has the same amount of methods that need to be implemented but most of the time several are forwarding methods which take less effort.

7.4.5. Overall cost model

The analysis of the techniques and how they influence implementation of features has identified four different types of efforts:

Method redefinition
This type of effort is needed when an existing needs to be modified, or a new method should be implemented. Because of the complexity of the example case these two activities will be modelled as an equivalent amount of effort.

Method forwarding
Forwarding is needed for aggregation structures and is easy to implement. The effort needed for this will be lower than a method redefinition. However, the methods still need to be written and therefore contribute to the total effort that is needed to implement a feature.

Class definition
Class definitions are a commodity when software is implemented and for inheritance and aggregation there is no real difference in the class definitions that need to be done. However, for Composition Filters sometimes it is necessary to implement additional classes, which means that the amount of class definitions can influence the total effort. Class definitions can be compared to method forwarding in terms of effort that is needed.

Filter definitions
When Composition Filters are used it is necessary to define filters and super-imposition. This type of work is called filter definition. The effort that is needed to implement a filter will be considered equivalent to a method redefinition, because of the low complexity of the example case.

To define a cost-model for the techniques based on the effort-types described above, contributions can be used for each of the effort-types. For instance it would be possible to assign an amount of time which is needed to perform a certain effort-type. When

one method redefinition needs to be done this will take 2 hours for a typical employee. For several method redefinitions the hours are multiplied.

For ImpGroup the following cost model is defined:

Effort-type	Manhours
Method redefinition	2
Method forwarding	0.5
Class definition	0.6
Filter definition	2.2

With this cost model it is possible to calculate the costs (in terms of effort) for adding a specific feature to the existing system.

7.5. Calculating the optimal policy

7.5.1. Defining the dynamic program

The dynamic program that represents the evolution process can now be calculated from the states of the requirements evolution diagram. First the state representation for the dynamic program needs to be defined.

Extra information in each state of the dynamic program is needed when techniques exist have initial costs. This is because in every state of the dynamic program it is necessary to know whether the initial costs for a certain technique have been paid. To include this information in the dynamic program extra states need to be generated based on the two techniques that represent the technique with the initial costs. For the example case first a dynamic program calculation will be described with techniques that have no initial costs. Secondly the calculation of the dynamic program will be described where Composition Filters have an initial cost.

An evolution process without initial technique costs

The calculation can be started with the start-state of the requirements evolution diagram: E-Mail. This state represents the stage at which the engineer has to make the first decision: with which technique to implement the initial requirement of the customer.

The next step is to determine what techniques can be chosen from this state. For the start-state all techniques can be chosen so there are four different actions the engineer can decide on:

Inheritance, Aggregation, Composition Filters, Total Reimplementation

Based on the possible next states of the requirements evolution diagram the following states can be defined for Inheritance:

{E-Mail, USViewMail, Inheritance}
{E-Mail, ORViewMail, Inheritance}
{E-Mail, LockingMail, Inheritance}
{E-Mail, DynamicMail, Inheritance}
EndStateLevel0

For aggregation:

{E-Mail, USViewMail, Aggregation}
{E-Mail, ORViewMail, Aggregation}
{E-Mail, LockingMail, Aggregation}
{E-Mail, DynamicMail, Aggregation}
EndStateLevel0

For Composition Filters:

{E-Mail, USViewMail, Composition Filters}
{E-Mail, ORViewMail, Composition Filters}
{E-Mail, LockingMail, Composition Filters}
{E-Mail, DynamicMail, Composition Filters}
EndStateLevel0

And finally for Total Reimplementation:

{E-Mail, USViewMail, Total Reimplementation}
{E-Mail, ORViewMail, Total Reimplementation}
{E-Mail, LockingMail, Total Reimplementation}
{E-Mail, DynamicMail, Total Reimplementation}
EndStateLevel0

For all techniques the premature end-state *EndStateLevel0* is the same state and can be shared. The picture below illustrates the state-space explosion dynamic programs generally face: for each state in the evolution diagram that is not an end-state a state is added for every technique. All these states can be reached after one decision, which means they all belong to Level 1. The dynamic program up to now looks like this

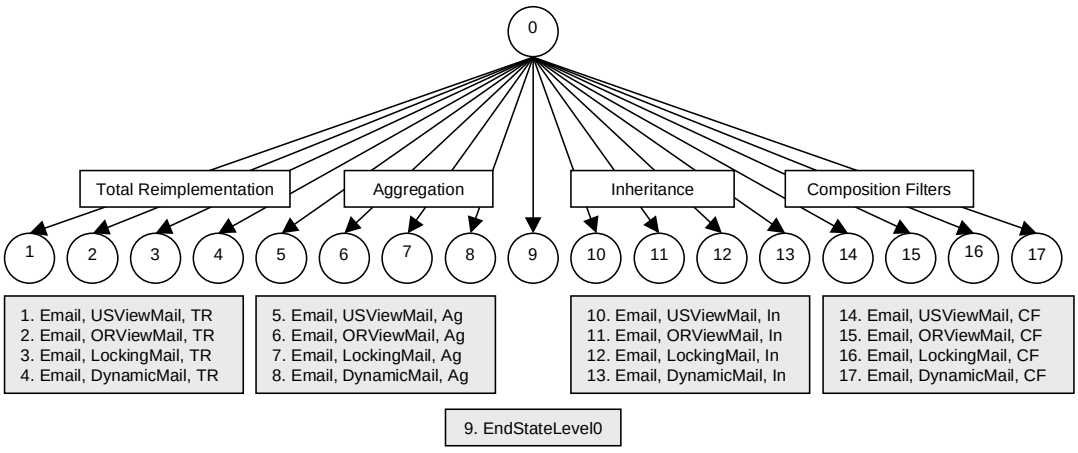


Figure 35: Dynamic program of the first level

Now the next states can be calculated for all states of Level 1. For each state the calculation is the same as the previous step. The first state:

{E-Mail, USViewMail, Inheritance}

The techniques that can be chosen from this state is the same as for the initial state: all techniques. Because no restrictions exist it is possible to choose either of the available techniques. The state from the requirements evolution diagram that corresponds with this state is:

{E-Mail, USViewMail}

When Inheritance is the action that is chosen the next possible states for this action will be:

{E-Mail, USViewMail, ORViewMail, Inheritance}
{E-Mail, USViewMail, LockingMail, Inheritance}
{E-Mail, USViewMail, DynamicMail, Inheritance}
{E-Mail, USViewMail, Warning2Mail, Inheritance}
EndStateLevel1

When this process is completed for every state in Level 1, all the states from Level 2 are known and the algorithm can be repeated for this level. Whenever a state is generated during the process that is equal to a state that already exists, this state should only be included once. The state already present can be used by the state from which it could be reached.

Because of the size of the result the entire dynamic program will be included on the CD in text-format.

An evolution process with initial technique costs

When techniques exist that have an initial cost the dynamic program has to store in every state whether the decisionmaker has already paid the initial costs or not. For the example case Composition Filters have been defined with initial costs. To calculate the dynamic program another technique needs to be defined: *Composition Filters inited*. This technique has exactly the same properties as Composition Filters but has no initial costs. In a state where the decisionmaker already has paid the initial cost Composition Filters Inited can be chosen, but not Composition Filters. From a state where the costs are not yet paid, Composition Filters is the only technique with these properties that can be chosen. This way the reward-function is able to determine the proper costs for choosing a certain action at a certain state.

The process of generating the dynamic program closely resembles that without initial technique costs, only for each state now the applicable set of actions need to be identified:

Starting at the initial state the following actions can be chosen:

Inheritance, Aggregation, Composition Filters, Total Reimplementation

In the initial state the first decision has to be made so the initial costs for Composition Filters have not been paid yet. The following next states can be identified:

For Inheritance:

{E-Mail, USViewMail, Inheritance}
{E-Mail, ORViewMail, Inheritance}
{E-Mail, LockingMail, Inheritance}
{E-Mail, DynamicMail, Inheritance}
EndStateLevel0

For aggregation:

{E-Mail, USViewMail, Aggregation}
{E-Mail, ORViewMail, Aggregation}
{E-Mail, LockingMail, Aggregation}
{E-Mail, DynamicMail, Aggregation}
EndStateLevel0

For Composition Filters:

{E-Mail, USViewMail, Composition Filters}

{E-Mail, ORViewMail, Composition Filters}
{E-Mail, LockingMail, Composition Filters}
{E-Mail, DynamicMail, Composition Filters}
EndStateLevel0

And finally for Total Reimplementation the possible next states will be:

{E-Mail, USViewMail, Total Reimplementation}
{E-Mail, ORViewMail, Total Reimplementation}
{E-Mail, LockingMail, Total Reimplementation}
{E-Mail, DynamicMail, Total Reimplementation}
EndStateLevel0

The next step again is the calculation of the next states of every state in Level 1. This is the same as for techniques without initial costs except for Composition Filters. When the next states for for instance *{E-Mail, ORViewMail, Composition Filters}* should be determined the set of techniques that can be chosen becomes:

Inheritance, Aggregation, Composition Filters inited, Total Reimplementation

Here Composition Filters inited can be chosen because the initial costs have been paid by choosing Composition Filters in the initial state. The possible next states now can be calculated in same way as other states.

By restricting the action set for every possible next state of a state where the initial costs for Composition Filters have been paid, it is possible to know this relevant information. For instance the state *{E-Mail, ORViewMail, USViewMail, Inheritance}*. When it is not necessary to know whether initial costs have been paid this state can be reached from for instance *{E-Mail, ORViewMail, Aggregation}* and from *{E-Mail, ORViewMail, Composition Filters}* by choosing Inheritance. But it wouldn't possible to tell whether Composition Filters has been chosen from state *{E-Mail, ORViewMail, USViewMail, Inheritance}*.

To solve this, the action set for every state that can be reached from a state, at which the initial costs for Composition Filters have been paid, can only contain *Composition Filters inited* (next to the other possible techniques) and not Composition Filters. Now when Inheritance is chosen at state *{E-Mail, ORViewMail, Aggregation}* this would lead to (when USViewMail occurs next):

{E-Mail, ORViewMail, USViewMail, Inheritance}

with the following possible actions:
Inheritance
Aggregation
Composition Filters
Total Reimplementation

But when Inheritance is chosen at the state *{E-Mail, ORViewMail, Composition Filters}* this would lead to (when USViewMail occurs next):

{E-Mail, ORViewMail, USViewMail, Inheritance}

with the following possible actions:
Inheritance
Aggregation
Composition Filters inited
Total Reimplementation

These two states are not identical because the actions that can be chosen are different. The second state is also a state where it is known that the initial costs for Composition Filters have

been paid (this can be seen because the action Composition Filters inited can be chosen). So every state that can be reached from this state also can not have Composition Filters as a valid action but will have Composition Filters inited instead.

This causes additional states to be generated, which is logical because it is necessary to store additional information. Compared to the process with no initial costs, the process with initial costs will have more states in most levels but the process of computation is the same.

7.5.2. Defining the reward-function

The costs of implementing a feature with a specific technique can be calculated based on the cost model defined earlier. For instance when USViewMail needs to be implemented (and no technology switch is applicable) the effort that is needed is:

11 method redefinitions + 1 class definition

According to the cost-model that has been defined for the example case a method redefinition requires 2 manhours and a class definition 0.6 manhours. The total effort that is needed to implement USViewMail then becomes:

*(11 * 2) + (1 * 0.6) = 22.6 manhours*

The table below contains all costs of implementing a features for a specific technique without technology switching (four protocols are assumed for DynamicMail). The costs of implementing the base system is equal for each technique because the same amount of work needs to be done (1 class definition and 14 method redefinitions).

Feature	Inheritance	Aggregation	Composition Filters	Total Reimp.
E-Mail	28.6	28.6	28.6	28.6
USViewMail	22.6	25.1	9	32
ORViewMail	14.6	19.1	11.2	32
DynamicMail	10.6	32.6	9	30
LockingMail	32.6	12.6	15.6	32
Warning2Mail	30.6	30.6	7.6	38

This table will be used as the definition of the reward-function for the example case. This means that in any state where for instance Inheritance is used as the implementation technique, and LockingMail is asked for, the implementation with Inheritance would cost 32.6 manhours. When for instance Aggregation is chosen this would be 12.6 and the costs of reimplementing the existing system parts, because of the technology switch and the mutual exclusion of the techniques. When a technique is chosen for the first time and this technique has an initial cost (the reward-function can derive this from the state and its actions), the reward-function will add the initial costs to the total reward.

7.5.3. Solving the dynamic program

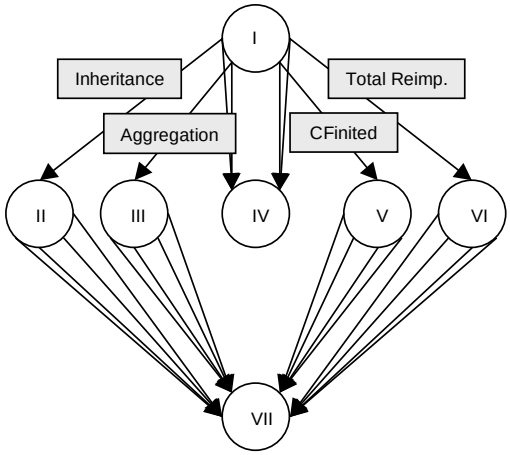
To demonstrate the way in which the dynamic program is solved by the backward inductive algorithm, solving it for the initial state and describing it would lead to a very large computation. But because the dynamic program can be solved for any state in the dynamic program, two subtrees will be solved. The path that has been followed to reach this state is unimportant for the computation, because the costs have already been paid and therefore can

not be minimised. Only the possible costs in the future can be minimised. For the computation an initial cost of 5 manhours is assumed for Composition Filters.

Suppose the following state needs to be analysed:

{E-Mail, USViewMail, Warning2Mail, DynamicMail, LockingMail, Inheritance }
with actions:
Inheritance
Aggregation
Composition Filters
Total Reimplementation

The subtree that belongs to this these state is made up by all states that can be reached from them. The following subtree is identified for the first state:



I. Email, USViewMail, Warning2Mail, DynamicMail, Inheritance
II. Email, USViewMail, Warning2Mail, DynamicMail, LockingMail, ORViewMail, Inheritance
III. Email, USViewMail, Warning2Mail, DynamicMail, LockingMail, ORViewMail, Aggregation
IV. Premature endState
V. Email, USViewMail, Warning2Mail, DynamicMail, LockingMail, ORViewMail, Composition Filters
VI. Email, USViewMail, Warning2Mail, DynamicMail, LockingMail, ORViewMail, Total Reimplementation
VII. Total end-state

Figure 36: Subtree of the dynamic program

When state I is designated as the start-state the backward inductive algorithm can now solve this dynamic program. Three levels can be identified in the dynamic program:

Level 0: I
Level 1: II, III, IV, V, VI
Level 2: VII

The backward inductive algorithm starts with the highest level (2), and calculates the contribution of these states to the cost-function. Because VII is the total end-state no actions can be chosen. All features that can be asked for have been implemented so the reward for this state will be zero:

u^*_2 (VII) = Reward(VII) = 0

Now the backward inductive algorithm goes up one level and determines the best action for each state in this level:

State II:

$u^*_1(II) = \min \{ reward(II, a) + \Sigma P(j \mid II, a)\, u^*_2(j) \}$

Four different actions can be chosen from state II (inheritance, aggregation, composition filters inited and total reimplementation). For inheritance the computation would be:

*Reward(II, Inheritance) + P(VII | II, Inheritance) * u^*_2(VII)*

The value of *P(VII | II, Inheritance)* evaluates to 1 because only one next state can be reached for Inheritance. And because only one possible next state exists the summation is restricted to one term. u^*_{t+1}*(VII)* was calculated the previous level and equals 0. *Reward(II, Inheritance)* finally is calculated by the reward-function. The feature that needs to be implemented is ORViewMail and no technology switch occurs because in state II inheritance is the technique that is used. *Reward(II, Inheritance)* therefore equals 14.6. The total computation becomes:

*14.6 + 1 * 0 = 14.6*

An equivalent computation can be done for Composition Filters:

*Reward(II, CF) + P(VII | II, CF) * u^*_{t+1}(VII)*

The expression *P(VII | II, CF inited) * u^*_{t+1}(VII)* again evaluates to zero for the same reasons as inheritance. The value of *Reward(II, CF)* will be 5 + 69.8 + 11.2. Choosing Composition Filters causes initial costs that have to be paid (5). State I has Composition Filters as a possible action. This means that the initial costs for Composition Filters have not been paid earlier. And because in state II the initial costs of Composition Filters still have not been paid, this has to be done now. The second part, 69.8, is caused because choosing this action from this state causes a technology switch: every feature needs to be reimplemented. The total computation becomes:

*5 + 69.8 + 11.2 + 1 * 0 = 86*

For Aggregation the computation becomes:

*129.5 + 19.1 + 1 * 0 = 148.6*

And for Total Reimplementation:

*160.6 + 32 + 1 * 0 = 192.6*

This means that the best action for state II would be Inheritance that would lead to an expected cost of 14.6.

When this calculation is repeated for every state in Level 1 the following values can be found (optimal decisions are bold):

State of Level 1	Actions	Expected cost
II	**Inheritance**	**14.6**
	Aggregation	149
	Composition Filters	86
	Total Reimplementation	192.6
III	Inheritance	139.6
	Aggregation	**19.1**
	Composition Filters	86
	Total Reimplementation	192.6
V	Inheritance	139.6
	Aggregation	148.6
	CF Inited	**11.2**
	Total Reimplementation	192.6
VI	Inheritance	139.6
	Aggregation	148.6
	Composition Filters	**86**
	Total Reimplementation	192.6
IV	No possible actions	0

From the table it becomes clear which actions are best and for each state the best action and the expected cost for choosing this action is stored. Note that for state VI *Composition Filters* is selected as the best action. This is because in state VI the system has been optimised for the known requirements before this state (the definition of total reimplementation). Whenever a new feature is asked (in this case ORViewMail) the entire system needs to be reimplemented which is the same as a technology switch.

At state V it is not possible to choose *Composition Filters* but rather *Composition Filters inited*. This is because the state can only be reached by choosing *Composition Filters* at state I. This means the initial costs will be paid by this action, and for that reason don't need to be paid for this state.

The backward inductive algorithm goes up one more step. Now the best action needs to be calculated for state I.

> In state I the features E-Mail, USViewMail, Warning2Mail and DynamicMail have been implemented with inheritance structures and LockingMail still needs to be implemented. The computation is:
>
> $u^*_0 (I) = \min \{ \text{reward}(I, a) + \Sigma P(j \mid I, a)\, u^*_1 (j) \}$
>
> From state I four actions can be chosen (inheritance, aggregation, Composition Filters and total reimplementation). The values for each of these actions will be calculated and from these results the best action will be chosen.
>
> **Inheritance**
>
> $reward(I, Inheritance) + \Sigma P(j \mid I, Inheritance)\, u^*_1 (j)$
>
> Reward(I, Inheritance) is evaluated to 32.6 (LockingMail should be implemented, no technology switch). From state I two states can be reached when Inheritance is chosen: II and IV. Because of the static probabilities ORViewMail will be asked for with a probability of 0.2 (which would mean the next state is II). The probability of

ORViewMail not occurring would be 0.8 and this will be the probability of reaching IV. The computation will look like this:

$32.6 + 0.2 * u^*_1(II) + 0.8 * u^*_1(IV) = 32.6 + 0.2 * 14.6 + 0 = 35.52$

The values of $u^*_1(II)$ and $u^*_1(IV)$ are the values of the best decisions for state II and IV.

Aggregation

$reward(I, Aggregation) + \Sigma P(j \mid I, Aggregation)\, u^*_1(j)$

Reward(I, Aggregation) is evaluated to 129.5 (LockingMail should be implemented, technology switching occurs). From state I two states can be reached when aggregation is chosen: III and IV. The same probabilities apply for ORViewMail being asked for: 0.2. The probability of reaching IV would again be 0.8. The computation will look like this:

$129.5 + 0.2 * u^*_1(III) + 0.8 * u^*_1(IV) = 129.5 + 0.2 * 19.1 + 0 = 133.32$

Composition Filters

$reward(I, Composition\ Filters) + \Sigma P(j \mid I, Composition\ Filters)\, u^*_1(j)$

Reward(I, Composition Filters) is evaluated to 5 + 69.8 (LockingMail should be implemented, technology switching occurs and initial costs have to be paid). From state I two states can be reached when Composition Filters is chosen: V and IV. The same probabilities apply for ORViewMail being asked for: 0.2. The probability of reaching IV would again be 0.8. The computation will look like this:

$5 + 69.8 + 0.2 * u^*_1(V) + 0.8 * u^*_1(IV) = 69.8 + 0.2 * 11.2 + 0 = 77.04$

Total Reimplementation

$reward(I, Total\ Reimplementation) + \Sigma P(j \mid I, Total\ Reimplementation)\, u^*_1(j)$

reward(I, Total Reimplementation) is evaluated to 160.6 (LockingMail should be implemented, technology switching always occurs for Total Reimplementation). From state I two states can be reached when Composition Filters inited is chosen: VI and IV. The same probabilities apply for ORViewMail being asked for: 0.2. The probability of reaching IV remains 0.8. The computation will look like this:

$160.6 + 0.2 * u^*_1(VI) + 0.8 * u^*_1(IV) = 160.6 + 0.2 * 86 + 0 = 177.8$

The final level has been calculated and the following results have been computed by the backward inductive algorithm for this level (best action in bold):

State	Action	Expected cost
I	**Inheritance**	**35.52**
	Aggregation	133.32
	Composition Filters inited	72.04
	Total Reimplementation	177.8

The analysis leads to the result that Inheritance would be the best decision with an expected cost of 35.52 manhours. This result of course is as accurate as the assumptions that were done and needs to be applied with this in mind.

7.5.4. Generalisation of the results

It is also possible with these results to generalise the values that have been found. When a closer look is taken at state VI, the best action for this state is Composition Filters. The expected cost for this action is 86. But for Composition Filters an initial cost should be paid in this state. When the initial costs had been higher, would Composition Filters still be the best solution? When will the break-even point be reached?

This can be analysed by defining the initial costs as a variable. The expected costs for choosing Composition Filters in state VI are calculated by redefining the system with Composition Filters (81 manhours) and the initial costs (5 manhours). When the initial costs are made variable the formula for the expected costs becomes:

Expected cost = x + 81

The costs for other techniques remain the same, because Composition Filters and its initial cost is no part of the computations of expected costs for these techniques. When these values are plotted in a graph it looks like this:

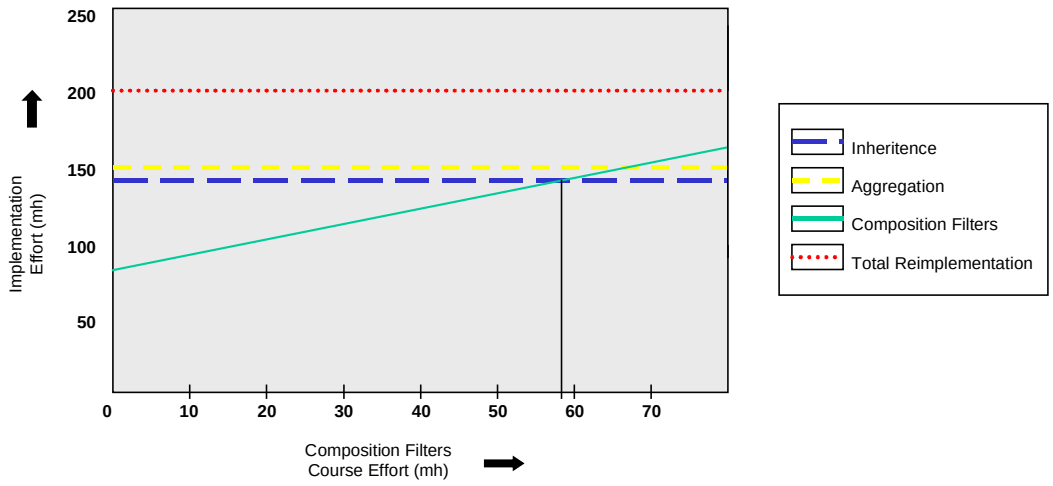


Figure 37: Generalized result graph

In the graph the expected costs for Composition Filters and Inheritance are equal when the initial costs for Composition Filters are 58.6 manhours, which can also be calculated from the function. From this point on, Inheritance will be the best action for this state. The software evolution analysis model can assist in various ways because of this type of calculation. For

ImpGroup it is obvious that when the evolution process ends up in this state the course should certainly not be taken when it takes more 58.6 manhours.

7.6. Summary

In this chapter the software evolution analysis model was applied to a small example case. A thorough analysis of the different aspects of software evolution for this software system was done and for a single state in the dynamic program the optimal decision was identified.

The size of the computations that need to be done makes it impossible to describe the entire process of software evolution analysis. From this it becomes clear this model should be automated as much as possible, also because the most labour-intensive parts (evolution diagram extraction, dynamic program definition) can be done algorithmically.

8. Conclusions

As this whole volume is one long argument, it may be convenient to the reader to have the leading facts and inferences briefly recapitulated.

Charles Darwin, the origin of species

8 Conclusions

Conclusions, recommendations and suggestions

In the previous chapters a model was presented for analysing impact of evolution scenarios and ways of preparing for this. In this chapter the results will be analysed and recommendations will be given towards future research.

8.1. Introduction

Software evolution is a serious problem in the field of software engineering. Customer have new or other requirements on the initial system. Although not always, most of the time it is very difficult to extend or modify the system to meet these requirement changes. The new requirements affect different parts of the system at the same time, which means in the worst case all these parts need to be replaced. These kinds of maintenance activities can cost up to 70% of the entire budget for a software project.

Over the years several techniques and structures have been identified that are able to make system reuse easier, such as inheritance, aggregation and composition filters. However, not in every situation these techniques provide easy reuse. Depending on the customer requirements and how they are interrelated, some techniques will perform better than others.

Current methods hardly address the issues of software evolution even when there is a strong focus on reusability and adaptability. Methods like the Unified process and product lines don't adequately identify the probabilistic character of evolution and the requirements that might occur. Although methods like organisational domain modelling are applied and problem domains are analysed thoroughly, software evolution might extend the problem domain at a later stage. This way a proper preparation might fall short in the future.

What is needed is a methodology which is capable of identifying future changes to software systems. It should be possible to relate the different features of a software system to each other. This makes it possible to analyse the impact of adding certain features to a system. Techniques can searched that can help in solving the types of impact that might occur. By relating the probabilistic behaviour of the evolution scenarios to the impact that might occur the different techniques can be assessed in terms of applicability for the specific software system and its future uses.

This model makes it easier for engineers and customer to design systems for dynamic environments. Software systems will become more long-lived and a better insight will be gained into systems complexity and the way customer requirements relate to each other.

8.2. Conclusions in the software evolution analysis model

The software evolution analysis model defines software evolution in terms of customer requirements and the way in which the customer can behave in asking for new/other

functionality. Customer behaviour is the driving force behind software evolution and therefore should be an integral point of attention when software evolution is analysed.

Software evolution analysis should be done parallel to requirements modelling and system architecture design. A prominent part of the software evolution analysis model is based on problem domain modelling methods like FODA. When problem domain modelling is done, these results can be used as input for evolution analysis. The results of software evolution analysis can be used in different parts of the cycle. Architecture decisions can be made based upon results from the software evolution analysis model, but also budget-decisions or configuration-decisions.

Because the occurrence of new requirements can have different impacts on the existing software system the different aspects the relations for these aspects should be modelled. The software evolution analysis model uses a FODA-based representation for this, where the different aspects of a software system are defined in terms of *features*. Between features different relations can be defined, that represent the dependency the different system parts have towards each other. By identifying different types of relations the impact of adding or removing features can be assessed.

By defining probabilities for the occurrence of features, it is possible to define customer behaviour by using non-deterministic state-transition diagrams. Each state in such a diagram represents a set of features that already have been demanded. From each state several other states can be reached with a specific probability. By designing representative probability models it is possible to define the possible customer behaviour accurately. The transitions that can be done are defined by the relations that exist between the features in the FODA feature tree.

Based on the impact preparation techniques can be identified. For the use of these techniques a cost model can be defined, based on existing cost modelling techniques or a complete new one. When, in the final step of the software evolution analysis model, the costmodel is related to the customer behaviour, predictions can be done on the costs that can be expected when these techniques will be applied. By using principles of probability theory and multi-stage decision problems the solution techniques and analysis techniques of this discipline can be used to provide justifiable results. The solving techniques that are available make it possible to analyse the different stages the evolution process can reach to a great extend.

The software evolution analysis model provides a formal way to analyse and assess evolution impact and comparing ways of preparing for future customer requirements. The normal way of handling these problems up to now has been based on the experience of the engineers assigned to the project. Whenever an engineer has worked in similar projects and has found certain solutions to work better than others, this knowledge is transferred. The engineer is forced to express the earlier experiences in terms of impact and applicable techniques. This leads to a better understanding of why the applied techniques have influenced earlier results positively earlier, and whether the current project can also be helped by such an approach.

The applicability of an analysis that is done by the software evolution analysis model depends on the way in which the costs and probabilities are modelled. Probabilities on customer behaviour can be defined to great extend, depending on market situations and even on events that occur within the evolution process itself. By defining probability functions these models can be changed on-the-fly for computations. For defining the cost-model the engineer can call upon any way of modelling costs that exists. As with the probability model it is possible to define costs in terms of cost-functions that can be changed on-the-fly in analysis computations. The accuracy limits the freedom of the engineer in defining this models however. The accuracy and relevance of the results of the software evolution analysis model depends on how

accurate the probability- and cost-models represent the real-world situation in which the software project will be implemented and used.

Because the software evolution analysis model uses state-transition diagrams for representing customer behaviour and system configurations, and because of the Markovian state-property that applies to these models, the software evolution analysis model faces to possibility of state-space explosions for the analyses of large software systems with many possible extensions. Even the use of optimised solving techniques such as the backward inductive algorithm do not provide sufficient ways of dealing with the amount states and paths that need to be calculated. This is an issue that needs attention, because the size of the analyses that will be performed will generally make the analysis a very labour-intensive task.

With respect to the requirements that were done earlier, the following conclusions can be drawn:

Relate customer behaviour to system complexity. By modelling requirements in terms of concepts that relate to the customer and defining relations between these features, the impact of changes can be identified and assessed. Customer behaviour is modelled with state-space models that represent the different scenarios that can occur. By applying probability models to the requirements evolution diagram the uncertainty of customer behaviour is represented, leaving the engineer free to define a probability model that is as accurate as is needed for the project and the analysis.

Determine which (groups of) requirements and additional system complexity is probable. By using the requirements evolution diagram graph and the probability models and tools that are available from probability theory, it is possible to assess the system complexity that can occur from changes to the set of requirements. By identifying likely changes and high-risk complexities it is possible to determine the parts of the system that might pose problems when the customer asks for an upgrade of the existing system.

Determine what alternatives for system implementation can be used best to minimise cost and risk. The third step of the software evolution analysis model is able to relate techniques and cost-models to customer behaviour and occurring complexities. By defining a cost-model in terms of complexity that can occur (features) and the techniques that affect the way in which the extensions can be implemented, the possible configurations can be computed. The integrated probability-models provide the risk-assessment for possible future costs.

Provide advice on how to prepare the system for the possible future requirements on the software system. The final step of the software evolution analysis model applies the defined cost- and probability-models to the possible configurations in which the evolution scenarios can end up. By determining the best decision for each of these configurations, the software evolution analysis model can provide feedback on the costs that can be expected with choosing specific actions at certain states. By using dynamic programs and the backward inductive algorithm, the model and its implementation can be optimised for the specific type of analysis that is required (maximal feedback on the analysis, minimal computation time for fast results).

8.3. Conclusions on the example case

When the model is applied to the example case it becomes clear that the state-space explosion is a problem that will occur soon when analysing software evolution. For a simple analysis with five possible extensions and four techniques, the amount of states grows rapidly when order is important and when initial costs for techniques have to be paid. Even though the process is highly automatable, the rate at which the state-space grows for more complex problems can pose serious problems for the analysis tools. And because projects that typically

will be analysed will include more features and techniques that need to be considered, minimisation of the state-space should be done as much as possible.

For the example case the best choice would have been obvious without any initial costs. Composition Filters are designed the address the problems that have been identified in [Aksit2001a]. The possibility of the software evolution analysis model to include initial costs in the computations allows for a validation of investing in new techniques when new projects are started. And by defining a generalised solution for any specific state in the dynamic program, the break-even point can be found for different techniques depending on initial costs.

8.4. Recommendations and suggestions

The software evolution analysis model uses discrete probability-models for modelling the customer behaviour. In real live discrete probability-models might not be able to model this customer behaviour accurately. Customer behaviour might be dependent on time, for instance because market conditions that have changed. Using continuous probability-models with the software evolution analysis model should be researched, and ways of modelling the customer behaviour with these models. By combining the continuous probability-models with the discrete cost-models it should be possible to model the customer behaviour more accurately.

The state- and action-space explosions have to be addresses to use the software evolution analysis model with the analysis of larger problems. A way of dealing with this is historical data. By comparing the current project to projects that have been done in the past, it should be possible to determine which configurations can occur and which will most likely not. By using these results the state-space can be minimised. The historical data can also be used to force the model to do lazy-evaluation of the possibilities. When a state is calculated that is very unlikely to occur (based on the historical data and some tress-hold), subsequent states might be affected by this. The lazy-evaluation can take advantage of this by short-circuiting the computation for these states.

Because of the amount of computations that need to be done, optimisation of these computations should be researched. At this point two different types of optimisations have been identified: analysis optimisation and computational optimisation. Analysis optimisation stores all the information that can be relevant to the engineer. For each state in the dynamic program the solution is calculated and all these solutions are stored. This makes it possible to review the entire analysis and run simulations to understand the evolution process to greater extend.

The computational optimisation is aimed at minimising the amount of computations that are needed to reach a desired result. For the the software evolution analysis model this means that when a solution is required from a certain state, the backward inductive algorithm will solve the subtree that has this state as the starting state. This way only the computations are done that are needed to reach the desired result. However, this approach makes it impossible to acquire information about the entire process of evolution. Only the calculated subtree can be analysed more thoroughly.

Research on evolving techniques should also be a part of extending the software evolution analysis model. During a project for which the software evolution analysis model is used, the techniques that have been analysed might evolve, which influences the costs that can be expected.

8.5. Reflection

During the research I have learned a lot about how to conduct a proper scientific approach to reach results that were desired. It was especially interesting to see how a mathematical theory like dynamic programming can be used to solve decision problems like evolution analysis. The nature of a scientific research requires a great amount of detail, and especially for this assignment since the research area was fairly new.

By examining and reusing existing methodologies I gained an insight in researches that already have been done, and how to search for these related topics. By adopting these researches to fit my own research the values of these researches are raised and my own research becomes more credible. But once again, a great amount of detail in applying these researches is needed.

The assignment itself was very interesting from my point of view, since I have also focused on management during my study. Designing methodologies that can help out in making decisions on for instance budgets, such as the software evolution analysis model, can provide valuable feedback to the engineers that apply it, but also to the management team that gains insight into the risks for a certain project. When a model can be derived from the base that is defined by the software evolution analysis model, it could prove valuable for companies that face evolution in software systems.

As a whole I feel that I have performed sufficiently during the research although some points can be improved. In the future it would be nice to have a more strict time-schedule. For the thesis generally 8 months are planned, but in my case I have used almost 11 months. Part of it can be explained from the fact that the research delivered interesting results and to raise the quality of the thesis, research was continued. But I still feel it would have been possible to finish the research earlier with the same results.

All in all the assignment has been a very interesting and educational experience which for me is a proper conclusion of my study. I have worked on it with much pleasure and I hope the committee feels the same way.

9. References

The other and more general departments of natural history will rise greatly in interest. The terms used by naturalists of affinity, relationship, community of type, paternity, morphology, adaptive characters, rudimentary and aborted organs, will cease to be metaphorical, and will have a plain signification.

Charles Darwin, the origin of species

9 References

Books, readers, websites and other sources that were used

[Aksit1994] Aksit, M., (ed.), *Reader Object-oriented systems*, Enschede University of Twente, 1994

[Aksit2001a] Aksit, Mehmet, Bergmans, Lodewijk, *Software evolution problems in case of inheritance and aggregation based reuse*, 2001, e-tutorial downloadable from: http://trese.cs.utwente.nl/

[Aksit2001b] Aksit, Mehmet, Bergmans, Lodewijk, *Solving the evolution problems using Composition Filters*, 2001, e-tutorial downloadable from: http://trese.cs.utwente.nl/

[Carnegie2001] Software Engineering Institute, *A framework for software product line practice - version 3.0*, Carnegie Mellon University, http://www.sei.cmu.edu/plp/frame_report/coreADA.htm http://www.sei.cmu.edu/plp/framework.html#framework_toc

[Czar2000] Czarnecki, K., Eisenecker, U. *Generative Programming: Methods, Techniques, and applications* Addison Wesley ISBN: 0-201-30977-7

[Gluss1972] Gluss, Brian, *An elementary introduction to dynamic programming, a state equation approach*, 1972, Allyn and Bacon, Inc. Library of Congress number: 70-150849

[Darwin] Darwin, Charles, *The origin of Species*, Bantam Classic and Loveswept; ISBN: 0553214632

[Jacobs1999] Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Development Process* Addisson Wesley ISBN: 0-201-57169-2

[Kang1990] Kang, Kyo e.a *Feature-oriented domain analysis (FODA) feasability study*, SEI Interactive, Carnegie Mellon University. PS-file: FODA.ps from http://interactive.sei.cmu.edu/

[Pressman1992] Pressman, R.S., *Software engineering – A practitioner's approach*. 3rd ed., McGraw-Hill, 1992

[Puter1994] Puterman, Martin, *Markov decision processes, discrete stochastic dynamic programming*, 1994, Wiley-Interscience, ISBN: 0-471-61977-9

[SCK1996] Simons, M., Creps, D., Klinger, C., Levine, L., Allemang, D. *Organization Domain modelling (ODM) Guidebook*, Version 2.0 Informal Technical report for STARS, STARS-VC-A025/001/00, 1996, http://www.organon.com

Appendices

At the commencement of my observations it seemed to me probable that a careful study of domesticated animals and of cultivated plants would offer the best chance of making out this obscure problem. Nor have I been disappointed; in this and in all other perplexing cases I have invariably found that our knowledge, imperfect though it be, of variation under domestication, afforded the best and safest clue. I may venture to express my conviction of the high value of such studies, although they have been very commonly neglected by naturalists.

Charles Darwin, the origin of species

Appendix A: SeamPrototype

A.1 Introduction

The SEAMPrototype is a program with which it is possible to apply the Software evolution analysis model to the cases that have been defined in the thesis and the presentation. It is possible to change several attributes of the features, the techniques and the evolution diagram and the influence on the end-result can be analysed.

The SEAMPrototype will be described in this appendix in a manual-like manner. The prototype itself is included on the CD.

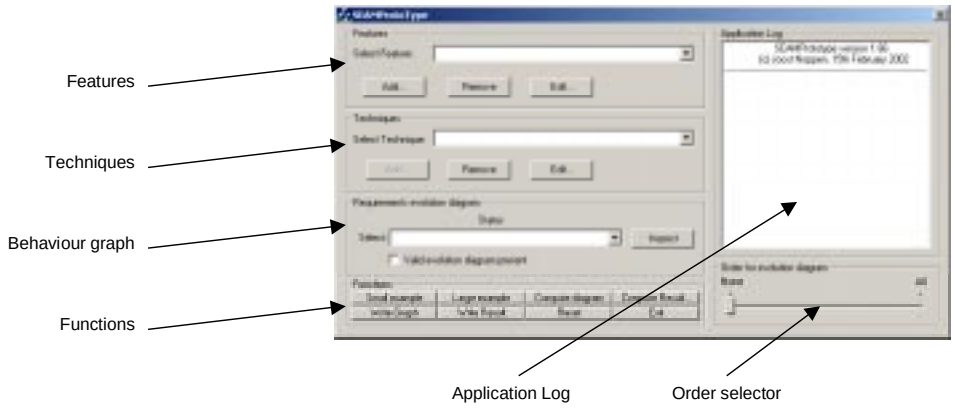
A.2 System requirements

The SEAMPrototype requires a Intel Pentium 100 MHz. or equivalent computer running Windows 95 or higher. For proper operation at least 16 MB of memory is needed. Several Microsoft foundation class libraries are needed to run the prototype, which are included in the zip-file on the CD. Because the program creates an eventlog on disk during program-execution it is not possible to run the program in a read-only environment.

The nature of the software evolution analysis model can cause state-space explosions when analysing software evolution. The computations done by the SEAMPrototype can take a long time because of this, and the memory allocation can become quite large. Because of this a computer is recommended with 256 MB of memory and a 500 MHz. or higher processor.

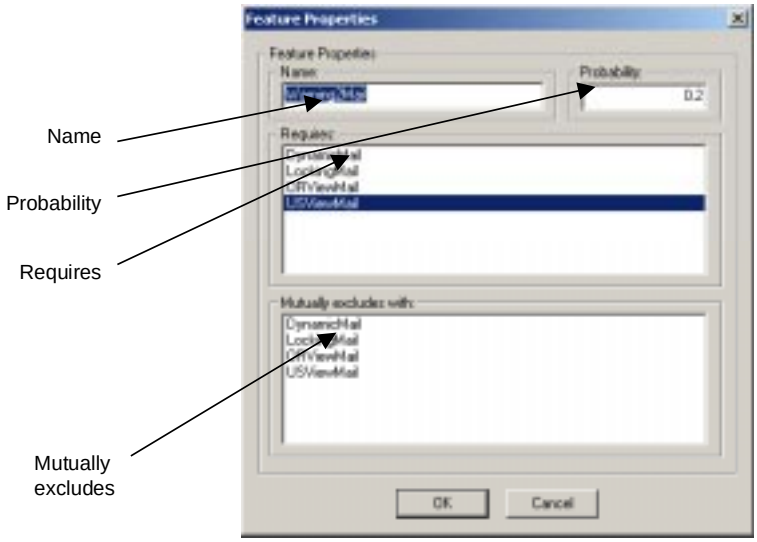
A.3 The main program

The main program consists of an interface where features and techniques can be defined, evolution diagrams can be calculated and the dynamic program can be solved. The interface looks like this:



A.4 Features

In the features-section the possible extensions on the software system can be defined. Features can be added, removed and edited. When one of these options is chosen a pop-up dialog appears with which the feature properties can be defined. This pop-up dialog will look like this:

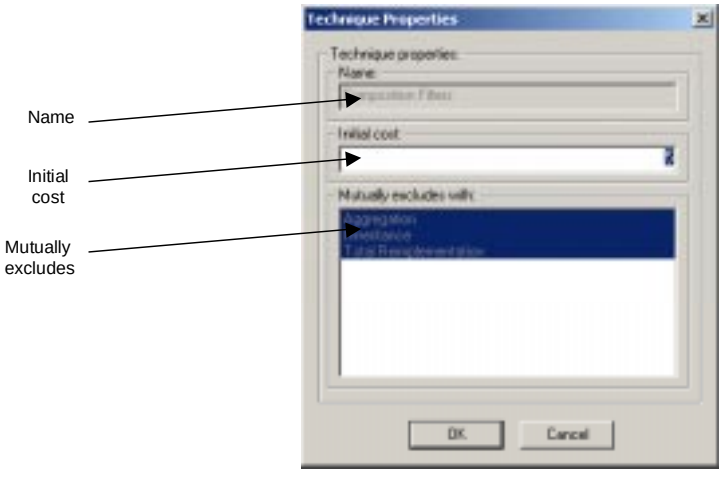


For a feature the name can be defined, which should be unique. The probability indicates the chance that a customer will ask for this specific feature. This means that the prototype uses a fixed probability model.

In the requires-section the requires-relations can be defined. This feature requires USViewMail to be present and therefore USViewMail is highlighted. When no feature is required none of the present features will be highlighted. The mutually excludes-section works in the same way as the requires-section. Features the current feature mutually excludes can be highlighted. The program does not allow features to be required and mutually excluded at the same time.

A.5 Techniques

Techniques can be defined in much the same way as features. For techniques also a pop-up dialog appears in which the several properties of techniques can be defined:

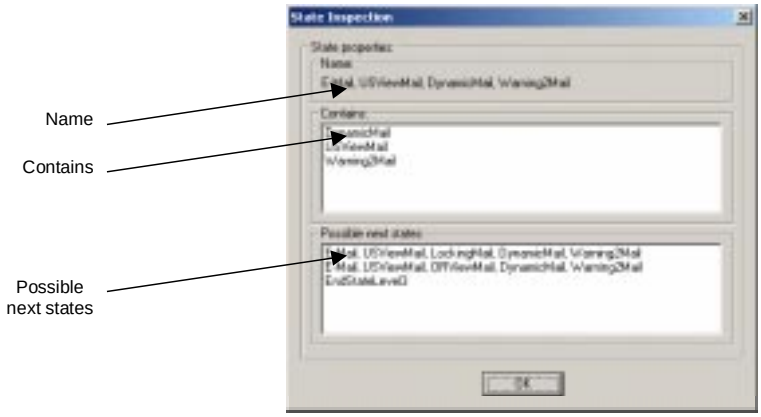


At this point it is only possible to change Initial costs for existing techniques. This is because the prototype only supports a reward-function for four different techniques at this point in time: Inheritance, Aggregation, Composition Filters and Total Reimplementation. These techniques can be loaded by using the *Large Example*-button. The *Small Example*-button loads Inheritance and Composition Filters as techniques.

A.6 Evolution diagrams

When all the features have been defined, it is possible to calculate the evolution diagram. To do this, the *Compute Graph*-button should be used. When this is done, the prototype will try to calculate the states of the requirements evolution diagram. These states will be added to the combo-box of the Evolution diagram-section. When a valid evolution diagram has been calculated the checkbox in this section will be marked.

By using the combo-box the several computed states of the evolution diagram can be selected and can be inspected by using the *Inspect*-button. The properties of the selected state will be displayed:

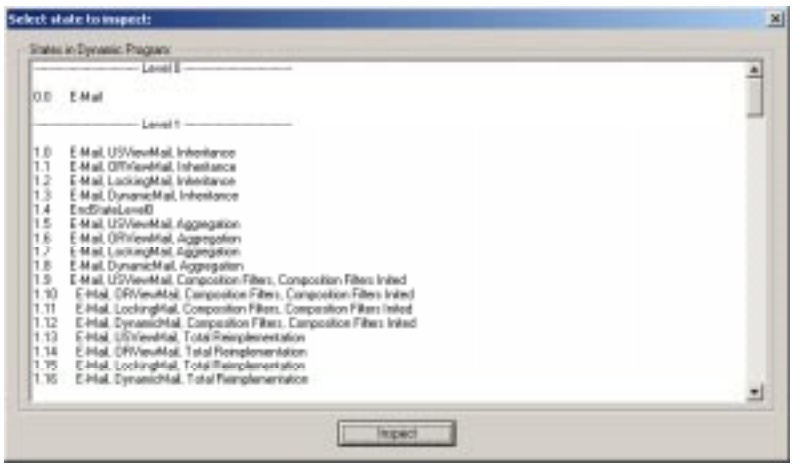


The name of the state is displayed, the features that already have been asked for by the customer, and the possible states that can be reached from this state in the evolution diagram.

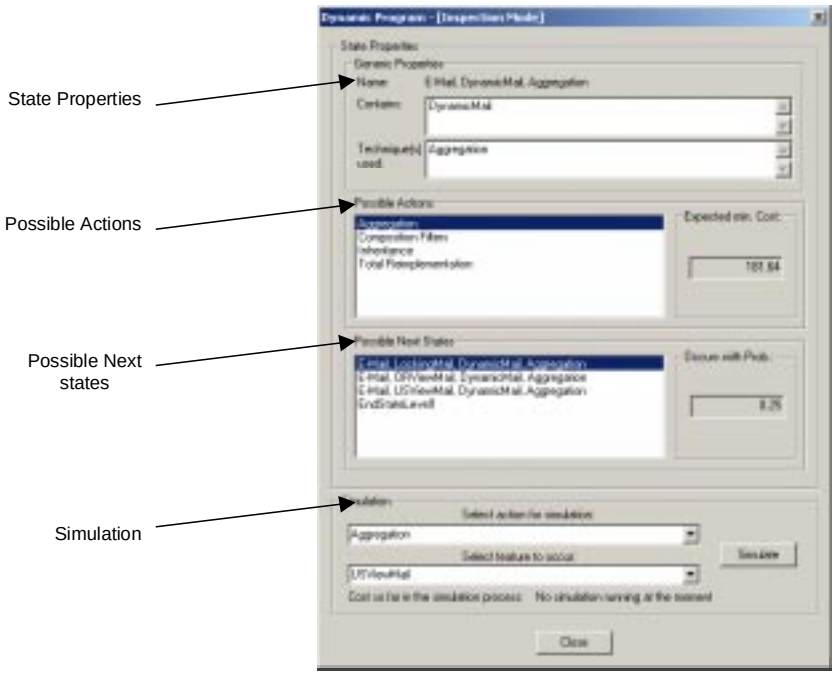
When a requirements evolution diagram is calculated the order in which the features are demanded might be important. To define the amount of order that is needed, the *Order selector* can be used. In the most left position a evolution diagram will be computed where order is not important. By sliding the selector to the right the amount of order increases. One step means a one-ordered evolution diagram where the feature that was asked for last can be retrieved from the state. Two steps means that the two features that have been asked for last can be retrieved from the state, etc. The amount of ordering should be set before the evolution diagram is calculated.

A.7 Computing the result

When the requirements evolution diagram has been calculated and all relevant techniques are present, it is possible to compute the dynamic program with which it is possible to analyse the possible evolution scenarios. To do this the *Compute Result…*-button should be used. When this is done the prototype computes a dynamic program with all the relevant states according to the software evolution analysis model. For each of these states the best action is determined by using the backward inductive algorithm. When this is done a list with all the states in the dynamic program is displayed:



Each of these states can now be inspected by using the *Inspect*-button or by double-clicking on one of them. The program will then display all the information of the specific state:



The state description is divided into three different parts: the generic state properties, the possible actions and the possible next states.

Generic state properties

The generic state properties describe the generic part of the dynamic program state: the name, the features that have been asked for by the customer and the technique(s) that ha(s)(ve) been used up to now to implement the system.

Actions and possible next states

The actions-section describes the possible actions that can be chosen by the engineer from this state. When an action is selected from the list, the prototype displays the minimal costs that can be expected from this state forward when this action is chosen.

The prototype also displays the possible states that can be reached with a certain action when it is chosen. On the right the prototype also displays the probability that this state will be reached when choosing this action.

Simulation

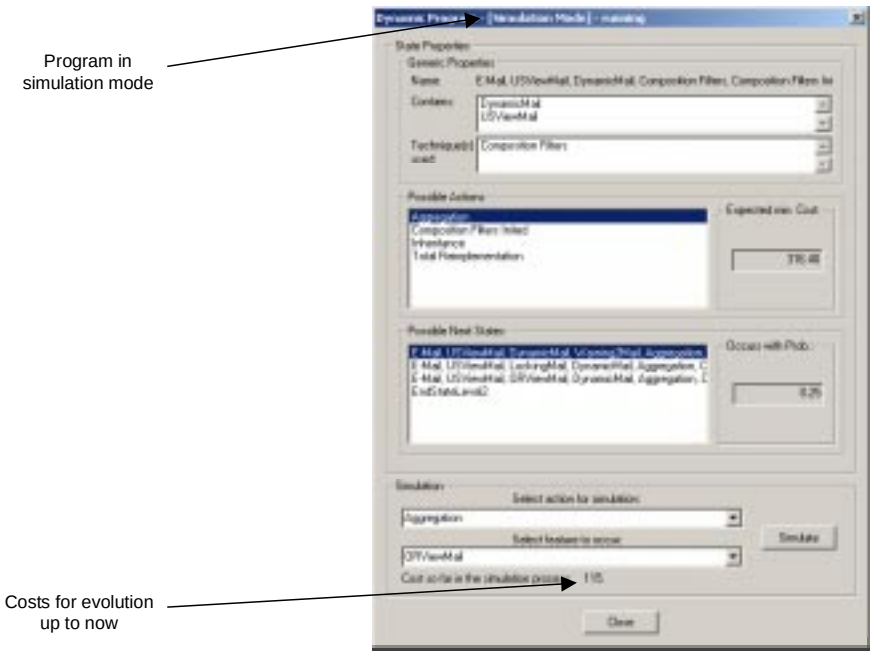
The last section is the simulation-section that can be used to run evolution scenario simulations and will be described to a greater detail in section A.8.

A.8 Simulating evolution scenarios

The SEAMPrototype offers facilities to run simulations of evolution scenarios. The simulation-section in the dynamic program state-inspector allows the user to specify the desired engineer- and customer-behaviour to see what happens. It is possible to select any of the actions that are possible from this state.

For the customer behaviour it is possible to define the feature that will be asked for next. Any of the features that are available from this state can be chosen. It is also possible to choose that none of the features will be asked for. Finally it is possible to have the customer behaviour simulated by the prototype. By choosing *randomise* the program will choose one of the possible features based on the probabilities.

When the *simulate*-button is used a new state from the dynamic program will be displayed but now the program will be in simulation mode. This can be seen by the caption of the dialog window. The window will now also display information on the simulation up to now:



The state now also displays the costs of the simulation up to now. These costs represent the costs that have been caused by the evolution process from the state from which the simulation was started. These kinds of simulations make it possible to asses the impact of features that occur with a low probability. When the costs for such a feature are very high they might not seam a high risk but the impact can be quite severe. Such scenarios can be simulated with the prototype.

From this state on, the simulation can be continued until a state is reached where no actions can be chosen (the simulation is completed). It is also possible to end the simulation and return to the state selector.

A.9 Additional Functions

The program has some additional functions that will be described here.

Application Log

In the application log the prototype displays relevant events that occur during run-time. Only information relevant to the user is displayed in the application log. A more complete description of the events that have occurred during runtime can be found in the eventlog that can be found in the execution directory of the SEAMPrototype.

Write Graph

With this function it is possible to write the last valid requirements evolution diagram to disk in a plain-text representation. In the execution directory the file *BehaviourGraph.txt* will be written. The size of this file depends on the amount of features that have been defined during runtime.

Write Result

This function writes the calculated dynamic program to a plain-text file in the execution directory. This will be done in the file *DynamicProgram.txt*. This file contains the dynamic program sorted per level. Each state is described in much the same way as during runtime, but only the best action and its expected cost is included. The size of the file depends on the complexity of the analysis but can become several megabytes.

Reset

The reset-button is included to make it possible to perform several analyses. When the reset-button is used all information will be deleted, which means all features, all techniques, the evolution diagram and the dynamic program. When this button is used to program can be used in the same way as when the program was first started.