A model and specification for development of enterprise application frameworks

A. van Oostrum

a.vanoostrum@student.utwente.nl

Department of Computer Science,

University of Twente,

The Netherlands.

August 27, 2009

version 1.0 (final)

“Maintaining large enterprise applications is hard, difficult and it can give you headaches. Having a good framework that is easy to use, intuitive to work with and easy to extend or to modify make s life a lot easier.”

ACKNOWLEDGMENT

The following people played a (significant) role in helping to finish this thesis:

University of Twente, The Netherlands:

dr. M.M. Fokkinga - first supervisor
dr.ir. W.K. Havinga
dr.ir. M. van Keulen - second supervisor

MASER Engineering, The Netherlands:

ing. W.J.K. Kemper - external supervisor
ing. C.T.A. Revenberg - external supervisor

NextSelect, The Netherlands:

L.A.J. Hellemons
T. Jansen
L.J. Koster
M. Niblett
D. Scheerens
S.A.A. Vercammen

PREFACE

Hereby I present my graduation thesis on the subject “A model and specification for development of enterprise application frameworks”. Finishing this thesis was probably one of the most challenging projects I have ever experienced. It required about one and a halve year for me to finish, were it should normally take about halve a year (one semester). Basically the delay was caused by the fact that I also had to keep my own company running while carrying out the research for this thesis. I could not step out of my company, not even for a short time, as this would have a negative impact on the continuity of the company.

When I started this research I had a rather naïve thought that with enough effort I would finish this thesis a lot sooner and at the same time run my company. Looking back now I realize that I underestimated the amount of time required to finish research subjects and to complete the thesis. Furthermore, when I was working for my company the thesis was always in the back of my mind and vice versa, which felt like a huge burden. I would not recommend this to anyone as it had a negative influence on the research and also on the company, for which I had to pay the price.

I have learned a lot and gained knowledge from this project. I am convinced that the results are going to help NextSelect build better software and as a result help to support the processes of MASER Engineering. The next step is to actually start building a new framework based on the finding in this report.

I would like to express my gratitude to Maarten Fokkinga. Not only for his wisdom and guidance, but also for his patience during the project. I would also like to thank Hans Kemper and Kees Revenberg for providing the research project and the means to perform the research. Furthermore I would like to thank the NextSelect staff for helping me with the designs and development of the prototypes, and making sure that the continuity of the company was maintained, it was not an easy period. At the end of the project Maurice van Keulen helped finishing the thesis, my gratitude goes out to him for the clean insight he provided. At last, I would like the thank my family and friends for their morale support, this gave me hope and courage to finish this thesis.

Alex van Oostrum
Enschede, The Netherlands, August 2009

ABSTRACT

Every software engineer knows that development of relatively large software packages without a good and preconceived plan to manage them is asking for a headache when these packages need further development, especially when the development is done by different software engineers. Frameworks are of key importance for developing large-scale object-oriented software systems. They promise higher productivity and shorter time-to-market through design and code reuse. However, many projects report that this promise is hard to fulfill: the design of object-oriented frameworks is all but well understood [1].

In this thesis a conceptual framework is represented that is intended to solve the problems of the software engineering company NextSelect in cooperation MASER Engineering. The framework is abstract by design and accompanied with a set of rules and specifications required to implement a concrete and deployable version of the framework. The design and specifications of the framework are driven by a unique philosophy which makes this conceptual framework also unique.

Not all aspects of the framework could be tested due to time limitations. Furthermore, the abstractness of the framework made it hard and even impossible to test all aspects. What has been used for validation is a proof of concept prototype implementation, scalability test of the design and a method for maximizing code reusability. The results thus far look promising, most of the objectives are achieved and could be tested with implemented prototypes. Based on this the framework concept appears to offer a solution for NextSelect and might offer a solution to other software development companies. Nevertheless, some aspects of the framework require further research and development before the framework concept can be fully used.

CONTENTS

1. INTRODUCTION

Every software engineer knows that development of relatively large software packages without a good and preconceived plan to manage them is asking for a headache when these packages need further development, especially when the development is done by different software engineers. Furthermore, the development of new code can take a lot of time that could be shortened if parts of the software code could be reused. However, creating reusable software is difficult, and there are many barriers that impede progress [16]. The use of a good and well constructed framework might make the development process shorter and easier and it is therefore appreciated. Frameworks are designed with the intent of facilitating software development and they have the goal to make the software development process simpler. This is achieved because a framework automates details and it offers common structure and better communication. Designers and programmers are therefore able to spend more time on meeting software requirements because they are dealing less with the more tedious low levels details of providing a working system.

The word 'framework' knows many definitions and it is used in many contexts. The term has been used more like a 'buzzword', especially in a software context. For example, the Java collections framework is not a software framework, but a library. For this research the following definition is used:

> ***"A software framework is a re-usable design for a software system (or subsystem). A software framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project."[2]***

NextSelect is a company with a core business of developing enterprise applications. Having built several large enterprise applications NextSelect experiences difficulties of managing and further developing them. The enterprise applications make use of a framework which should be common and reusable for all applications built by NextSelect. Unfortunately the opposite is true; business logic is located throughout the frameworks which resulted in different version of the framework for every application.

In our opinion other software development companies experience (partly) the same problems. Especially companies that also develop enterprise applications. With this in mind the research is broadened and not narrowed down by the boundaries of NextSelect. For the research NextSelect is used as a representative case.

In this thesis the terms ‘business logic’ and ‘application logic’ are both used. In many software engineering contexts they mean the same, in this thesis they are different and should not be mixed up. In subsection 2.2.1 this is elaborated.

1.1. BACKGROUND

It all started in 2002 when NextSelect had built it first enterprise application for a company called MASER Engineering. From that point on MASER and NextSelect collaborated in the software development of this enterprise application. The application, called MIDS (**M**ASER **I**ntranet **D**atabase **S**ystem), grew and evolved to a huge and complex application managing a lot of the business processes of MASER. Some example modules: human resource management (HRM), customer relation management (CRM), project management with detailed planning tool, quotation and order management, shipping and package traceability.

Why put effort in designing and building a system where other (big) companies, like SAP and Oracle, already have solutions available? One of the reasons is that these (large) software packages probably fit perfectly within a large number of companies using a common business process. But there are companies that have a (slightly) different business process or their business process often changes caused, for example, by the growth of the company. At this point costs could raise significantly to adapt the software, which could make the required adjustments unfeasible. As a result the company has to adapt its processes to the software instead of the software being adjusted to the processes. NextSelect uses a different approach; in this approach the business process has the main focus and the software is built around the business process.

In the first stages, of the development of MIDS, some design and implementation choices have been made that nowadays can be seen as 'ugly' and 'not done'. For example: no separation of business logic and framework code, no separation of application logic, model and presentation and the lack of a good object oriented design. As time passed MIDS evolved, however parts of the implementation of MIDS did not evolve and remained the same. So the customer (MASER) received a good working application, but underwater some parts were still a mess. There are mainly two reasons why certain development choices were made that leaded to the problems; first: the incompetence of NextSelect or lack of knowledge in the early stages of development; and secondly: tight budget and time limitations, which resulted that the bad code for the most part remained within the application.

As MIDS evolved so did NextSelect, it became competent and the development process of MIDS evolved. This resulted in nicer, more structured and better reusable code. However, code still remained and this complicated the development of MIDS.

In 2006 a new customer gave NextSelect the order to build a new business application to support parts of their business process. The difference between MIDS and the new application that had to be built was the business logic, the application logic for the most part is the same. Instead of rebuilding the whole application from scratch the existing framework code of MIDS, was reused. The idea of reusing the framework is great and should be encouraged. The problem, however, is the fact that this framework code contained a lot of MASER business logic, the

framework was custom built for MASER. As a result the MASER code was stripped out and replaced by business logic of the new customer. For a software developer two things are horribly wrong:

1. The existing framework contained business logic resulting in a customer specific framework that is not reusable.
2. By putting customer specific code in the two existing frameworks they automatically became two different versions, two versions that needed to be maintained.

The problem aggravated when more business applications had to be built for different customers. All the business applications share almost same framework but these framework contain business logic which make them different. Due to the number of frameworks the complexity of maintaining them increases, also the implementation time required to add a new feature or to resolve a problem increases. Rapidly building applications is becoming harder and NextSelect could lose its competitive advantage to the big companies like SAP.

A new framework with development guidelines is required which can be deployed for different applications and different customers. MASER and NextSelect started a collaboration to build a new framework with the ultimate goal to build a new version of MIDS using the new framework.

1.2. HYPOTHESIS

In anticipation of the results described chapter 2, this is the main research hypothesis:

> ***Does the framework design, sketched in Figure 1.1, offer a solution for the problems section in section 2.1 ?***

Figure 1.1 shows a representation of the architecture design of the framework, which is elaborated in section 3.2.

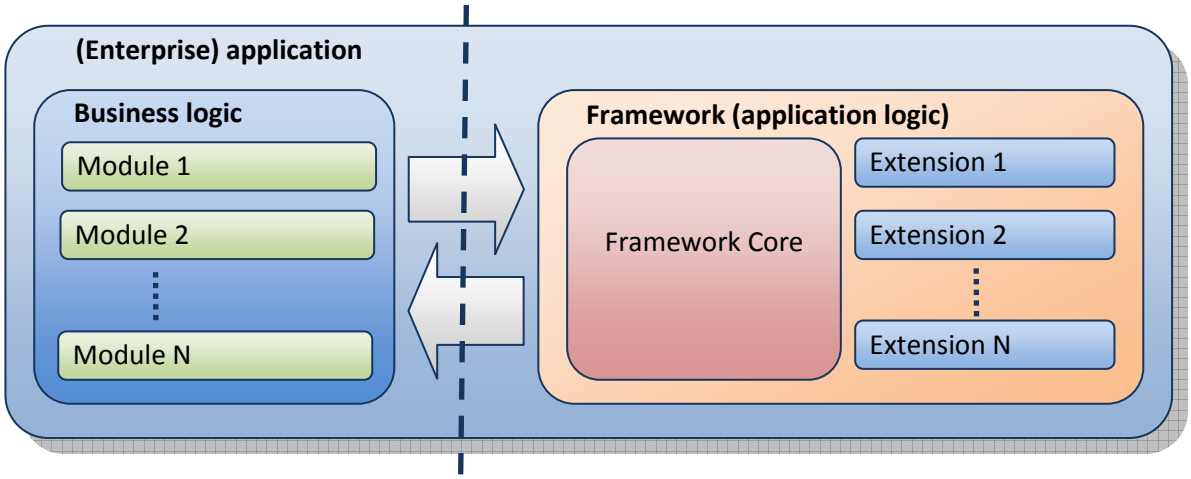


Figure 1.1: Framework architectural design. It separates business logic and framework (dashed line). Business logic is represented in the modules. A developer only has to script or configure a module, hence making software development easier. A module tells the framework what to do and when to do it. Features and functionality are put into extensions, which have the responsibility to facilitate the framework. The Framework Core manages the extensions and modules and takes care of the communications.

Separating business logic from the framework is one of the most important objectives of this research project, see section 2.2 for all the objectives. In Figure 1.1 this is depicted by the vertical dashed line. Within the modules requirements and specifications of a customer are translated into business logic. A module contains no functionality other than telling the framework 'what to do'. It is the responsibility of the *Framework Core* to locate the appropriate extensions and execute the requests issued by the modules. Extensions are tools that have the responsibility to facilitate the *Framework Core* and other extensions.

1.3. APPROACH

The research starts with the analysis of NextSelect business processes and a detailed problem analysis of the existing enterprise applications at a programming level. The option of using existing available frameworks has been exploited but rejected as it seemed that no existing framework met the research objectives. Furthermore, MASER and NextSelect prefer not to use third party software, due to the fact that they want to have full control over the software and not be dependent of a third party.

A framework philosophy is derived from the problem analysis and the wishes of NextSelect. This philosophy is translated into an the architecture shown in Figure 1.1. Details about this derivation, the philosophy and the architecture itself can be found in subsection 2.3.4.

Prototypes are implemented to refine the design of the framework. During the development of these prototypes two important aspects emerge; the design of the framework is abstract and it does not state anything about the type of application it is going to be deployed for. Because of the abstract design the framework is deployable for a wide range of applications. A secondary design is required that specifies the concrete implementation of a specific framework. This

secondary design reduces the range of applications the framework can be deployed for, due to the fact that the concrete design of the framework is made for a specific type of application.

Among other objectives, that can be found in section 2.2, three are used for validation to demonstrate the usefulness and feasibility of the framework. The design of the framework is validated by a prototype that implemented a small part of MIDS (section 4.1). It is reasonable to assume that the framework will grow with functionality in time, how this affects the scalability is tested with a scalability prototype (section 4.2). Another important issue with an growing framework is code reuse. Implementations should not be (partly) duplicated and code reuse has to be maximized. With a growing framework it becomes harder and harder to maintain the reuse of code due to the fact that a programmer has to know that a piece of code exist and, more importantly, the programmer has to be able to find this piece of code. A method for maximizing code reusability with the use of code retrieval is presented (section 4.3).

In Figure 1.2 the research model, followed during this research project, is illustrated. It gives an overview of the main activities during the project and where to locate the results within this document.

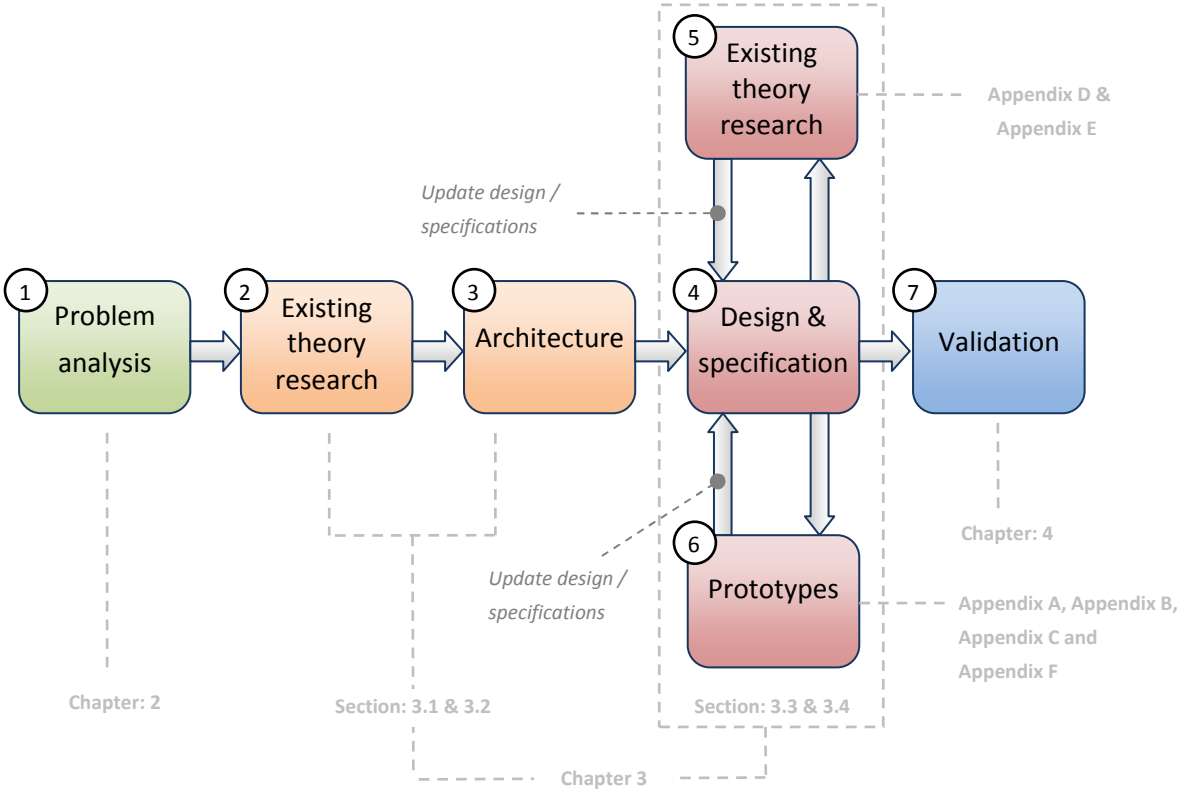


Figure 1.2: Research model used: Start with problem analysis using NextSelect case (1). Acquire knowledge from existing research theories (2). Design the framework architecture using the obtained knowledge (3). Produce a detailed design of the framework based on the architecture (4). Acquire additional knowledge (repeated) about isolated framework issues and update the framework design and specification if required (5). Build prototypes (repeated) that test isolated framework issues and framework design and update the framework design and specification if required (6). Validation of the framework design and specification conform the research objectives (7).

1.4. DOCUMENT STRUCTURE

This thesis report holds a typical research structure. Chapter 2 it starts with a problem analysis of NextSelect and states the objectives and research questions that lead the research. For this research the NextSelect case is used as a basis.

In chapter 3 the results of the research is presented, a model and specification for development of enterprise application frameworks. This chapter starts with the philosophy which forms the basis of the research and makes, to the best of our knowledge, this framework unique. Derived from the philosophy, problems, and research questions, an architecture and a detailed design of the framework are presented. However, the architecture and the detailed design of the framework do not completely cover the philosophy, therefore a set of rules and specifications are also provided.

In chapter 4 the results are validated. Validation is required to ensure that the objectives from section 2.2 are guaranteed. However, due to time limitations not all objectives could be validated. Furthermore, some objectives are hard, if not impossible, to validate.

During the development of the framework design tests and research was performed on certain aspects of the framework. For some prototypes had to be built to test a certain requirement of the framework design. In other cases some research had to be performed.

Appendix A: In the early stages of the framework design, a design pattern called Model View Controller (MVC), was assumed to be necessary in the basic abstract design of the framework. In this appendix this is tested with an implemented prototype.

Appendix B: The framework design needs to be platform independent. In this appendix the design with MVC from appendix A is used for a prototype developed a Java. The test showed that the design can be used in other object oriented programming languages.

Appendix C: The design of the framework needed a ‘real’ application to show its usefulness. A prototype framework has been developed to make Tic Tac Toe games. The design of the framework is further developed.

Appendix D: An important design issue about multiple inheritance, discussed in subsection 3.3.1, has been tested this appendix. The issue is that not all programming languages support multiple inheritance, a solution from Tempero et al. [13]. is used to resolve the issue.

Appendix E: The *Framework Core* needed a more facilitation role for the modules, like a façade. The behavior had resemblance to CORBA and Web Services that have broker architecture. In this appendix research is performed to see what the resemblances are and if the framework concept is not reinventing the wheel.

Appendix F: With the design of the framework finished, a prototype is built to test the framework for the NextSelect case. This prototype provides the thesis with a proof of concept and it provides the framework rules and specifications (section 3.4).

2. ANALYSIS & OBJECTIVES

Creating reusable software is difficult and there are a lot of barriers that need to be breached. To design a new framework, that will solve the problems of NextSelect, these barriers must first be identified. The enterprise application for MASER, called MIDS, and other enterprise applications built by NextSelect, are used as research cases for this thesis. Furthermore, the assumption is made that the identified problems are also common for other software development companies. The new framework should not be limited for use by NextSelect only.

To identify the problems this chapter starts by looking back into the history of NextSelect, focusing on the software development (subsection 2.1.1), why certain choices have been made that resulted in the current problems and how these problems make future software development difficult (subsection 2.1.2). Some attempts have be made to resolve certain issues; unfortunately they have all failed (subsection 2.1.3). NextSelect learned from its mistakes and in cooperation with MASER a solution is presented (subsection 2.1.4).

In this thesis the terms ‘business logic’ and ‘application logic’ are both used. In many software engineering contexts they mean the same, in this thesis they are different and should not be mixed up. In subsection 2.2.1 this is elaborated.

2.1. PROBLEM ANALYSIS

The first version of MIDS was released in the year 2002. Compared to the current version it was less complex, however this first version is still forms the basis of the current versions and even the new applications that were built later on for different customers. Due to design flaws and certain implementation choices, problems remained for the development of MIDS and other applications managed by NextSelect.

2.1.1. HISTORY

MIDS is a fully web-based application. This means that the client has to use a web browser, i.e. Internet Explorer and Firefox, to access the application. The application itself is located at a central server and is accessible via the intranet or the internet. MIDS is developed in the language PHP, MySQL is used for data storage and retrieval and Apache is used for web server, illustrated in the following figure.

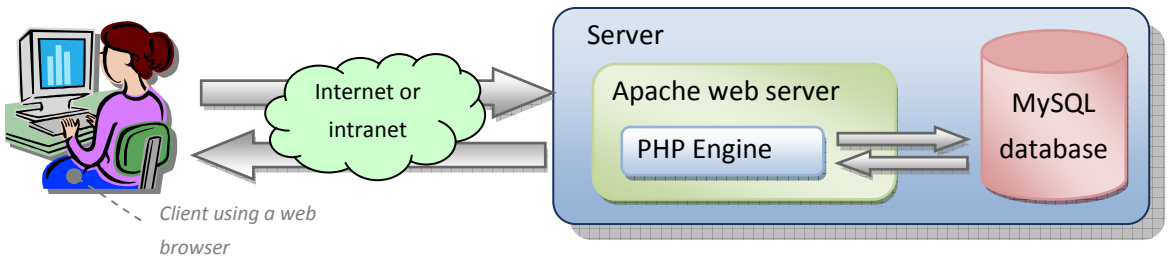


Figure 2.1: Schematic view of web based system. A client accesses a web based application by using a web browser. The client's requests are sent by the web browser through the internet or intranet to a server. The web server 'Apache' with the programming language PHP will process the request and send the output back to the client. PHP is able to use a database called 'MySQL' for data storage and retrieval.

The advantage of using a web based system is the fact that the application itself is centrally located. This makes updating the system easier because the clients do not have to be updated individually. The application will also run on any system that has a internet or intranet connection and a web browser.

During the early stages of the development of MIDS a lot of code duplication was discovered. This code duplication was mainly formed in the generation of the web based form (a form in HTML) to store new or modify existing data. Duplication of code was time consuming and sensitive for errors. If a mistake or bug was discovered it had to be replaced in different locations. Not the best software development approach and in fact a design flaw. Fortunately this was at the very beginning of the development and a new design was made to make a system that would allow the reuse of code to generate these forms. The first version of the MIDS framework was born, with an important philosophy of code reusability.

As good as the intentions of the first framework were, now looking back to the design of the framework with the current knowledge of software development and the issues involved with

this project, it has a poor design. Due to inexperience, not knowing that the system would grow to a large and complex enterprise application and of course time limitations, the first version of the MIDS framework was implemented with poor design. It was partly object oriented and the parts that were built using an object oriented approach did not apply the object oriented paradigm correctly. Aside from the lack of object oriented code, a much larger mistake was the fact that the framework was not built to be used for other applications, i.e. customer specific code and framework core were merged. For MIDS development this was no problem and as the years passed and MIDS grew, so did the framework. Unfortunately the problems grew also as more customer specific code was placed into the framework. Due to the fact that issues needed to be resolved quickly it often proved easiest to implement the customer specific code directly into the framework. Due to the fact that the framework already contained customer specific code which required a simple expansion or the bad design of the framework, it was time consuming to implement a new feature in a 'nice' manner (what is nice will be discussed later on).

In the year 2006, after about four years of 'wrong' development, NextSelect had to build a new enterprise application for a new customer. With the knowledge that the existing MIDS code had a poor design, the most proper method was to build a new framework with a better design which would not contain any customer specific code. As honorable as this idea was, the MIDS framework was so large and complex that a rebuilt of the framework would take a considerable amount of time and the costs for the customer would rise significantly. Without a proper budget from the customer it was impossible to build a new framework from scratch. So only one option remained: take the existing framework and rebuild the parts that were built specifically for MASER.

The MIDS framework was actually copied for the new application. All the containing MIDS code was removed or replaced by code for the new application which still kept the framework dirty. This resulted in two versions of the framework, essentially the same but different due to the customer specific code. Because the two versions co-existed both had their own development path making the differences between the two frameworks bigger. After some time the first real problems emerged, a bug was discovered that existed in both frameworks. The solution was first implemented in the MIDS framework , but the same solution could not be used for the second framework because it was not compatible due to the fact that part of the framework had a different implementation caused by the different development paths.

Having two versions of the framework already formed problems, this got even worse when more customers required new enterprise applications. With the same reasons as the second enterprise application, the framework was again copied and the parts that held customer specific code removed and replaced. Although more effort was put into separating the customer specific code this problem was not completely resolved due to the nature of the framework design.

2.1.2. THE PROBLEMS

The main problem originated with the first version of the framework. It was partly developed using an object oriented approach with a poor design. The framework was filled with customer specific code which made it impossible to easily separate the framework from the application to use it for other applications for other customers. Due to this improper framework a new framework had to be copied and adapted to be deployable for new enterprise applications. This resulted in different versions of the same framework which needed further development and maintenance as the enterprise applications required new functionality.

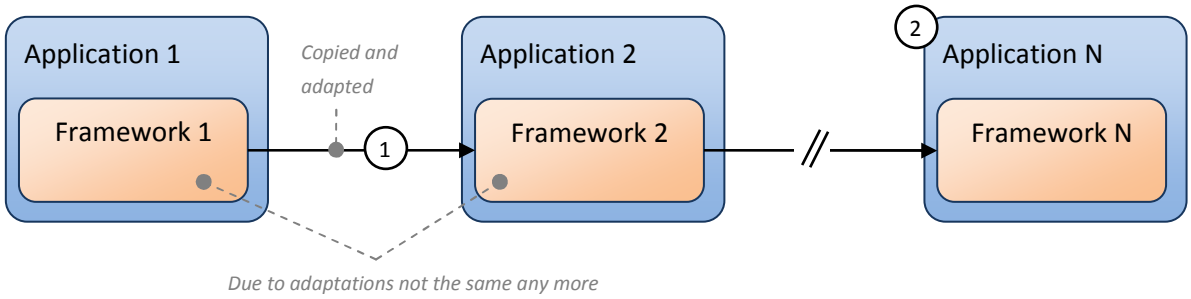


Figure 2.2: Problem origination. For every new application built the existing framework was copied to the new application (1). However, the copied framework was further developed for the new application making it a different version. More application were built resulting in more different versions of the framework (2). All applications are continuously in development, in result the corresponding frameworks also are in continuous development making the differences between the frameworks larger.

Beside the fact that the framework consisted of customer specific code, it has another problem. There is no clear separation between application logic, model and presentation. Due to the fact that the framework is written to be used for web based applications, it has to output its presentation in HTML (Hyper Text Markup Language). This HTML output is sent back to the browser of the client which is able to render the HTML to a graphical environment for the client. Without going into much detail about HTML and web based principles, the problem is that HTML output code is spread throughout the framework and application code. Changing the output due to customer specifications or a new specification of HTML (e.g. XHTML) requires changes not only in the framework, but also throughout the whole application. This is also a time consuming job which increases the change of making errors. As for the model, most of the data is stored in a database; however, to access this data different methods are used throughout the application. If, for example, another type of database is required it would require changes throughout the application and framework.

To summarize the problems:

- Poor, partly object oriented, design of frameworks and applications;
- Existing frameworks contain customer specific code (business logic);
- No separation between application logic, model and presentation;
- Multiple versions of the framework exist, requiring development and maintenance;

As a consequence, the problems lead to:

- Inefficient software development;
- Difficult maintenance of the software product.

2.1.3. ATTEMPTS TO RESOLVE THE PROBLEMS

Having different enterprise applications for different customers resulted in different versions of the framework. Managing these frameworks takes precious time which could be spent more efficiently. A new approach is required.

Two attempts have been made already. The first involved the integration of the second developed framework into the first enterprise application, MIDS. However, the integration did not go smoothly; the new framework differed a lot from the old framework in MIDS. As a result a lot of time would be required to replace the old framework and fully deploy the new framework. Unfortunately there was not enough budget to completely replace the old framework. In consultation with the customer (MASER) a compromise was made to use two frameworks next to each other and replace the old framework in phases. This process of replacing the old framework with the newer framework in phases is still active. The new framework has a better design, more functionality and new HTML support. For the MASER this is a good thing; the application is upgraded and supports more features. In spite of this, looking at a software developer's point of view, this approach made things more complicated. The framework should remain compatible with the application it came from, with the goal that MIDS and the other application could share the same framework instead of two different versions. Due to the fact that the new copied framework could not be fully deployed, the result was that the new framework required adaptations to make it compatible with MIDS and a second framework (the old framework which remained in MIDS). Due to these adaptations the new copied framework directly became a different version compared to the version it was copied from, i.e. again two versions of the framework were created.

To conclude the first attempt; from a software developer's point of view, the problems increased due to the fact that MIDS now has two frameworks that both require development and maintenance instead of one. In the following figures (Figure 2.3, Figure 2.4 and Figure 2.5) the process of the first attempt is illustrated:

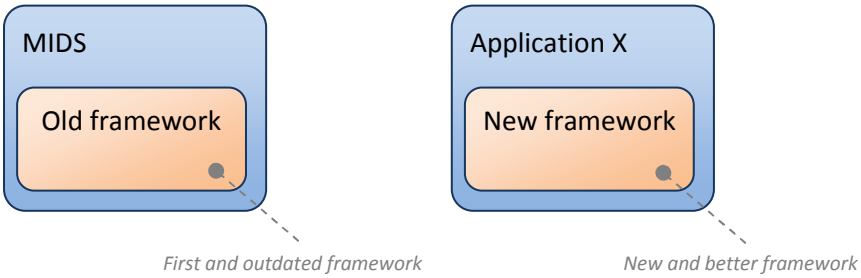


Figure 2.3: Situation before the first attempt: two enterprise applications; MIDS and application X. MIDS has an older framework compared to application X, which was later built and therefore has a newer framework.

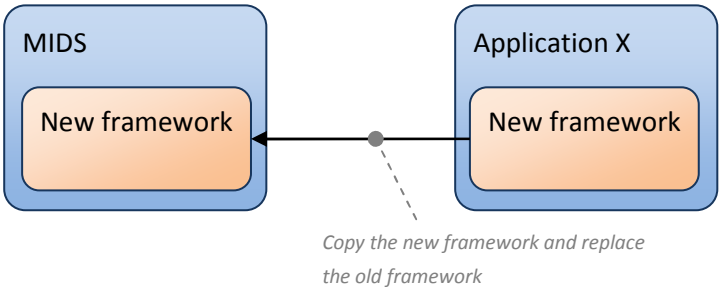


Figure 2.4: Intended situation: The older framework, used for the MIDS application, is replaced by the newer framework from application X. MIDS and application X share the same framework. Result: software development and maintenance is easier due to the fact that only one framework has to be managed.

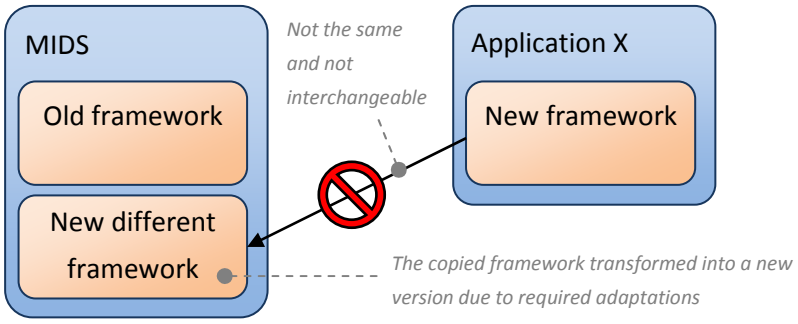


Figure 2.5: Current situation: due to insufficient budget the new framework could not be fully deployed at once. The customer agreed with the option to divide the deployment of the new framework into phases. However, the new framework required adaptations for this deployment in phases. As a result, the framework became different compared to the framework from application X. Furthermore, the software developers now have to manage two frameworks for the MIDS application until the deployment of the new framework is complete.

With the first attempt failing a second attempt was made. Like the first attempt a new framework is copied from a newer application to replace an older framework from a older application. However, in this case the old framework did not differ a lot from the new framework. This made it possible to replace the old framework with the new framework with a considerable small amount of adjustments. The second attempt was a success and the intended situation, just like Figure 2.4, was achieved.

Unfortunately this success did not hold very long. The design flaws that still existed in the frameworks made maintenance and further development inefficient and time consuming. This, and the fact that no actual development plan was formulated to maintain the frameworks, had the result that the two identical frameworks eventually split up into two different versions.

The second attempt achieved the intended situation where two applications shared the same framework. However, after a certain period of time they got back to the situation illustrated in Figure 2.3 (only different applications). It appears that due to the existing problems of the framework , it is impossible to maintain one framework for multiple applications.

2.1.4. THE REQUIRED SOLUTION

It is clear that walking the old path with the existing framework becomes more difficult as more enterprise applications have to be built. With two earlier attempts to merge different versions of the framework failing, one solution remained: build a completely new framework.

Rebuilding the framework must not be underestimated; it has to be deployable for different and complex enterprise applications. It has to be designed and built in such a way that the issues discussed in subsection 2.1.2 will not cause problems again.

NextSelect turned to MASER to discuss a possible solution. The choice to go to MASER was simple: MASER and NextSelect share the longest history compared to the other customers of NextSelect. Furthermore, the application 'MIDS' is the most complex of all applications developed by NextSelect but holds the oldest framework. It could be said that MIDS is in 'desperate' need of an 'upgrade'.

For NextSelect the new framework has to make development and maintenance of enterprise applications easy and fast. This is the foundation of the philosophy that is discussed in section 3.1. Customer demands have to be transformed into business logic which can, with simple commands, direct the framework to form an application that meets the customer demands.

In our opinion other software development companies experience (partly) the same problems. Especially companies that also develop enterprise applications. With this in mind the framework concept is broadened and the research is not narrowed down by the boundaries of NextSelect. The framework design should not be limited to be deployed only for 'MIDS' like enterprise applications, it has to be deployable for a wide range of applications, even when they are non web-based applications. The design of the framework takes this in to consideration.

2.1.5. EXISTING FRAMEWORKS

Most of the problems described in subsection 2.1.2 are not uncommon in the software development projects. Every good programmer wants to produce good, stable and reusable code. Before just starting to build a new framework it is important to perform some research to check if software exists that resolves (part of) the problems. This has two reasons: first, it easier to reuse somebody else' research instead of reinventing the wheel. Secondly, looking at other work and research, even if it will not resolve the problems directly, can give good insights about certain subjects and may prevent making mistakes.

During this research a lot of frameworks and even whole 'high level' programming languages were discovered to build applications. Research has been performed to identify those frameworks that could satisfy the objectives. One framework in particular was examined, Ruby on Rails. According to Ruby et al. [3] Ruby on Rails is a framework that makes it easier to develop, deploy, and maintain web-based applications. According to Smith et al. [4] it is possible to use Ruby on Rails to build enterprise applications. If the principles of Ruby are followed it will meet a lot of the objectives discussed in section 2.2 and it therefore might solve a lot of problems. However, a few objectives are not met; Ruby on Rails is not platform independent, Ruby is the programming language that is running on Rails. So there are two dependencies; the programming language Ruby and Rails as a programming framework.

Although not all the objectives are met, Ruby on Rails is still a high level programming language, however the framework in this research is of higher level. The philosophy is that an (enterprise) application is not programmed but 'scripted' (discussed in section 3.1), Ruby on Rails requires programming to build a complex application. However, due to the philosophy of Ruby on Rails it makes it a very good candidate to be used to implement a concrete implementation of the framework.

Due to the fact no existing framework could be found that satisfies the high level needs of MASER and NextSelect, the choice was made to build a new framework which will be managed by NextSelect. Beside the fact that none existing framework could satisfy the needs, MASER and NextSelect feel that having full control over the framework outweighs the advantages of a third party providing a framework.

Due to the complexity of the existing enterprise applications of NextSelect it is not possible to completely build a new framework within this research. Therefore this research focuses on the first step: a basic abstract design of the framework. This basic abstract design provides rules and specifications to build a concrete framework. Actually building a new framework is labeled as future work.

2.2. RESEARCH OBJECTIVES

The main goal is to design a framework that solves the problems described in subsection 2.1.2. By analyzing the derived problems and the required solution the following objectives are formulated.

Design a new framework that is:

- separating business logic and application logic; (2.2.1)
- object oriented by design; (2.2.2)
- completely modular built; (2.2.3)
- directed through a high level interface; (2.2.4)
- deployable for a wide range of applications; (2.2.5)
- system and platform independent; (2.2.6)
- fast and scalable; (2.2.7)
- maximizing code reusability; (2.2.8)
- able to rapidly build applications; (2.2.9)
- easy to extend or modify. (2.2.10)

With these objectives almost all the software development issues in the world are solved. For obvious reasons there are limitations to this research; the framework research focuses on enterprise application development and specifically enterprise applications built by NextSelect.

2.2.1. SEPARATING BUSINESS LOGIC AND APPLICATION LOGIC

In this thesis business logic and application logic mean two different things: business logic is a translation of customer demands and specifications. Application logic contains all functionality that facilitates business logic in order to meet the customer demands. In other words: different customers want the same application but have different parameters on which the application should operate.

In most applications business logic and application logic are mixed together because by some interpretations they are the same. However, do to the fact that in this framework business logic has to be separated from the actual implemented functionality that forms the application logic, i.e. the framework (see Figure 3.1), the two terms have different meanings.

One of the main concerns in the NextSelect case is the fact that business logic (customer specific code) is tied up with the framework. To be able to deploy the framework for different customers it is important that framework itself does not contain any business logic. Therefore this research focuses on completely separating the business logic and application logic, i.e. business logic is completely separated from the framework which holds the application logic.

2.2.2. OBJECT ORIENTED BY DESIGN

Using an object oriented paradigm gives several advantages compared to non object oriented approaches. According to Riehle [1] object-oriented frameworks promise higher productivity and shorter time-to-market of application development through design and code reuse.

One of the major problems with non object oriented approaches is that functions are given a higher priority than data, whereas data should be more important. Data can easily get corrupted because it is accessible throughout all the functions, even in functions that actually do not have any rights to access the data.

Using the Object Oriented paradigm these drawbacks are taken away. In this approach the data has the first priority. This is realized by packing the data and the functions, which are supposed to have access to the data, into one object. Unauthorized access of data and chances to corrupt data are limited, this is called encapsulation. Objects can be reused within the application or by other applications. This increases the code reusability that is one of the objectives of this framework (to maximize code reusability).

Lots of design patterns exist for the Object Oriented paradigm. This makes the development, future extending and modifications of the framework easier, especially when this is done by different programmers. An Object Oriented approach seems the best suited solution for this project.

2.2.3. COMPLETELY MODULAR BUILT

The main idea is that the application is split into two parts. The first part consists of all the modules that contain the business logic. The second part is the actual framework. Modules can be inserted into the framework and thus forming the application.

The framework itself is also split into two parts. The first part contains the framework core and the second part contains all the extensions. Extensions can be seen as tools that have the responsibility to facilitate the framework with reusable functionality and provide a high level interface for the modules. The framework core can be seen as a central hub that manages all communication between modules and extensions.

2.2.4. DIRECTED THROUGH A HIGH LEVEL INTERFACE

The framework has to offer a high level interface so that it becomes very easy to build a new application or to update an existing application. Following the philosophy (discussed in section 3.1) customer demands have to be translated into business logic that is placed in modules. Modules contain only a kind of scripting language telling the framework what to do and when to do it through this high level interface. This high level interfaces makes sure that with a few

commands the application can be put together which will save the developer time (see subsection 2.2.9) . Please note that if functionality does not exist within the framework this has to be implemented first.

There is no direct link between a request and a function within an extension. The framework is more flexible as it has to find the appropriate extension that offers the feature which can be done in several ways depending on the concrete implementation of the framework. More on this subject in section 3.1 and 3.2.

2.2.5. DEPLOYABLE FOR A WIDE RANGE OF APPLICATIONS

As stated in subsection 2.1.4, it is likely that other software development companies endure the same problems as described in subsection 2.1.2. For this research project the framework concept is therefore broadened to take into account that the framework has to be deployable for a wide range of applications. The design of the framework must therefore be abstracted to the point that there is no relation with the abstract design and a application.

Abstracting the design supports the wide range application deployment of the framework. However, it should be noticed that the main goal is to develop a framework that resolves the issues of NextSelect. Although the design makes the framework deployable for a wide range of applications, the research will be focused on enterprise applications and especially enterprise applications for NextSelect.

2.2.6. SYSTEM AND PLATFORM INDEPENDENT

If the framework has to be deployable for a wide range of applications, it should not be limited to any system or platform. Therefore the framework design has to be system and platform independent.

The design has to take into consideration that it might be used in different programming languages on different platforms. However, by the object oriented nature of the framework it can only be used in programming languages that support the object oriented paradigm. This is a limitation due to the objective described in subsection 2.2.2.

Furthermore, the current framework is currently web-based; the design of the new framework should not be limited to the choice if it is going to be a web-based or normal application.

2.2.7. FAST AND SCALABLE

Every user that is operating an application wants a quick response from the system, the user does not want to wait. The framework has to be fast in terms that the application that is facilitated by the framework has to be fast in responding to the user's request.

Taken into account the NextSelect case, the same framework should be used for different applications and different customers. In this way only one framework has to be maintained and further developed. Looking back to the history of NextSelect it is a known fact that the framework stays the same after a release. New applications will be built for new different customers and existing customers will ask for changes and/or new features. Due to this the framework will always increase in numbers of extensions instead of decreasing.

If the framework grows it is likely that the framework gets slower. Of course the concrete implementation of the framework has a great influence on the performance; nevertheless, the framework abstract design must be scalable. If adding new extensions or features leads to exponentially slowing down the framework, it means the framework is not scalable and therefore not fast and not usable.

In this thesis tests have been performed to test the scalability of the abstract framework. Due to the fact that it is not possible to benchmark an abstract design, a concrete implementation has been chosen that is based on the web based applications built by NextSelect. These tests are based on a concrete implementation of the framework that is web based and uses the same framework design found in Appendix F.

The speed of the framework is not tested due to the fact that such a test requires a concrete framework with actual functionality. Furthermore, this test would only be valid for that specific concrete framework and will not say much about the abstract design in this thesis.

Details about the scalability test can be found in section 4.2.

2.2.8. MAXIMIZING CODE REUSABILITY

With the same reasons mentioned in subsection 2.2.7, it is most likely that a concrete implementation will grow in numbers of extensions and functionality. This will be the case for NextSelect. To keep the framework 'clean', i.e. never implement (parts of) functionality more than once, it is important to maximize code reusability.

Imagine that a concrete implementation of the framework consist of hundreds of extensions that offer thousands of features. When a programmer has to implement a new feature how can he know if the features already exists in the framework ? Somebody that implemented all the features might know it, but humans tend to forget thing. Furthermore, what if new programmers have to implement new features to an existing framework.

Repetition of code makes the framework unnecessarily larger and it also contributes to the number of bugs; consider the NextSelect case, an extreme example where the framework itself was repeated for different applications. It does not matter how small the code is, the goal is to never repeat code.

To minimize code repetition, code reusability must be maximized. With code reusability two aspects are important; The first aspect plays a role when a new feature is designed. The design must take into account that the new functionality within the feature might be reused. For example: if all functionality of an feature is placed in one function, other features are unable to make use of that functionality and code will most probably be repeated. It is the responsibility of the software developer to take this into account when designing and implementing a new feature.

The second aspect is finding functionality or 'code retrieval'. If a new feature requires functionality it might be possible that this functionality is already implemented somewhere. Finding code is hard and falls under the information retrieval domain. In section 4.3 a possible solution is presented, however it requires future research.

2.2.9. ABLE TO RAPIDLY BUILD APPLICATIONS

Due to the problems described in subsection 2.1.2, development and maintenance of the current enterprise software is inefficient and time consuming. It is a must requirement for NextSelect that the new framework resolves this. The new framework has to make it possible to easily translate business logic into the modules without a lot of programming when a new enterprise application has to be developed. This forms the basis of the philosophy of the framework, which is discussed in section 3.2.

Besides this must requirement of NextSelect this objective is also useful for other software companies that might want to use the framework. However, it must be noted that this objective depends on the actual concrete implementation of the framework. Therefore rules have to be defined beside the abstract design, this is discussed in section 3.4.

2.2.10. EASY TO EXTEND OR MODIFY

This objective is a direct result from the previous objective discussed in subsection 2.2.9 (Able to rapidly build applications). The framework not only has to facilitate rapid building of new enterprise applications, it must also be easy to modify existing business logic and to add or modify features for the framework itself.

As discussed in subsection 2.2.7, an existing framework will eventually get new features. The design of the framework has to make this process easy. In this case also rules have to defined as the design itself will not suffice, also discussed in the previous subsection 2.2.9.

2.3. HPOTHESIS AND RESEARCH QUESTIONS

The research hypothesis is formulated as following:

> ***Does the framework design, sketched in Figure 1.1 and repeated below, offer a solution for the problems described in section 2.1?***

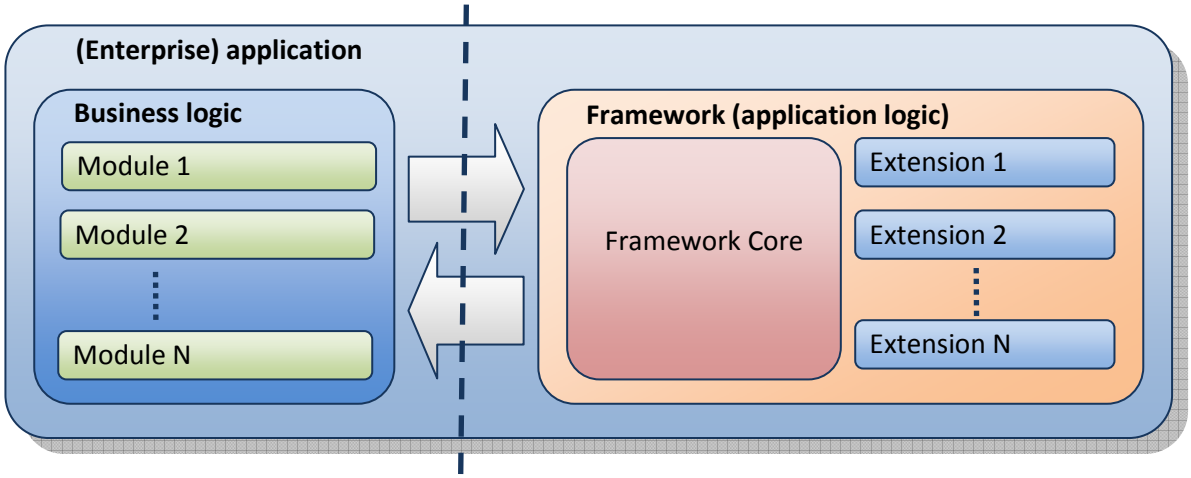


Figure 2.6: Framework architectural design. It separates business logic and framework (dashed line). Business logic is represented in the modules. A developer only has to script or configure a module, hence making software development easier. A module tells the framework what to do and when to do it. Features and functionality are put into extensions, which have the responsibility to facilitate the framework. The Framework Core manages the extensions and modules and takes care of the communications.

Besides the fact that this formulation of the hypothesis may look attractive, closer inspection and investigation raises a number of questions that need to be answered, they are listed in the following subsections. Some low level research questions are stated and elaborated in the appendices.

2.3.1. QUESTION Q1

> ***Is it possible to apply the framework design to other platforms, programming languages and non web-based applications?***

This research question discussed in all of Appendix B. The answer is summarized in subsection 5.2.2.

2.3.2. QUESTION Q2

> ***The CORBA and Web Service architectures show resemblance to the current framework architecture, is the framework concept - or parts of - the same as the architecture of CORBA and/or web services?***

One of the first questions that came up: “what is on the market?”. In other words, are there existing systems available? This research question is discussed in Appendix E. The answer is summarized in subsection 5.2.3.

2.3.3. QUESTION Q3

> ***How are modules and extensions implemented and how are they going to communicate?***

This research question is discussed in Appendix F and subsection 3.4.6, subsection 3.4.7. A summary of the answer is in subsection 5.2.4.

2.3.4. QUESTION Q4

> ***What is the usefulness of the Framework Core and why is it required?***

The research question is discussed in Appendix F, where the advantage of the framework is tested with a prototype. A summary of the answer can be found in subsection 5.2.5.

3. THE FRAMEWORK

Our philosophy:

> ***"Making (enterprise) applications has to be done fast and easy. A developer simply has to script or configure his whishes in simple calls without knowing much about the underlying implementations. The framework will take care of everything. One framework for multiple applications."***

This philosophy of the framework is based on the NextSelect case that provides problems and whishes, which in turn lead to the objectives and research questions. The framework philosophy tells the 'why, what and how' about the framework and makes this framework unique. To the best of our knowledge, no other framework exists in the world that satisfy the needs.

The framework consists of modules, *Framework Core* and extensions. Modules contain only business logic (i.e. whishes and demands of the customer) in a high level scripting language what tells the framework what to do and when to do it. Extensions can be seen as tools that have the responsibility to facilitate the framework with reusable functionality and provide a high level interface for the modules. The *Framework Core* can be seen as a central hub that manages all communication between modules and extensions.

The architecture and detailed design of the framework do not completely establish the philosophy, therefore a set of rules and specifications are provided in section 3.4. The success of the framework, i.e. resolving the issues described in subsection 2.1.2, depends completely on these rules and specifications. The design of the framework is abstract, this means that a concrete version of the framework has to be implemented; if the rules and specifications are not correctly applied during the implementation of a framework, the objectives (section 2.2) are not guaranteed.

Using this framework to build (enterprise) applications divides the development into two parts; framework development and application development. Framework development consists of low level implementations (core and extensions) whereas application development consist of high level implementations (modules). Due to the different development phases different kind of developers can be used.

3.1. PHILOSOPHY

What makes this framework unique is the philosophy behind it; it forms the foundation of the framework. The philosophy is derived from the problems and whishes that came out of the NextSelect case and is supported by the required objectives. The problems, described in subsection 2.1.2, lead to inefficient software development and difficult maintenance.

The wishes of NextSelect, also found in the objectives in section 2.2, evolved from the experiences obtained during development and/or maintenance of the existing frameworks and applications. The very first framework for MIDS was developed due to massive repetition of code. The first framework resolved a lot; however, due to a poor design and a lack of knowledge a lot was still repeated unnecessary. During the development of the latest application of NextSelect some important changes were implemented that resolved again a lot of code repetition, at that moment the philosophy was born. However, due to the poor design of the framework and application it would require building a new framework, leading to this research.

The enterprise applications that NextSelect builds and maintains should make use of one framework. This framework holds all functionality that any of the enterprise applications could offer to a customer. One framework means one framework to maintain and upgrade, hence saving time.

Customer demands have to be translated into business logic, which will only consist out of high level calls telling the framework what to do and when to do it. The implemented business logic may not contain any functionality except for some handling ‘when’ to tell the framework what to do. The philosophy is illustrated in Figure 3.1.

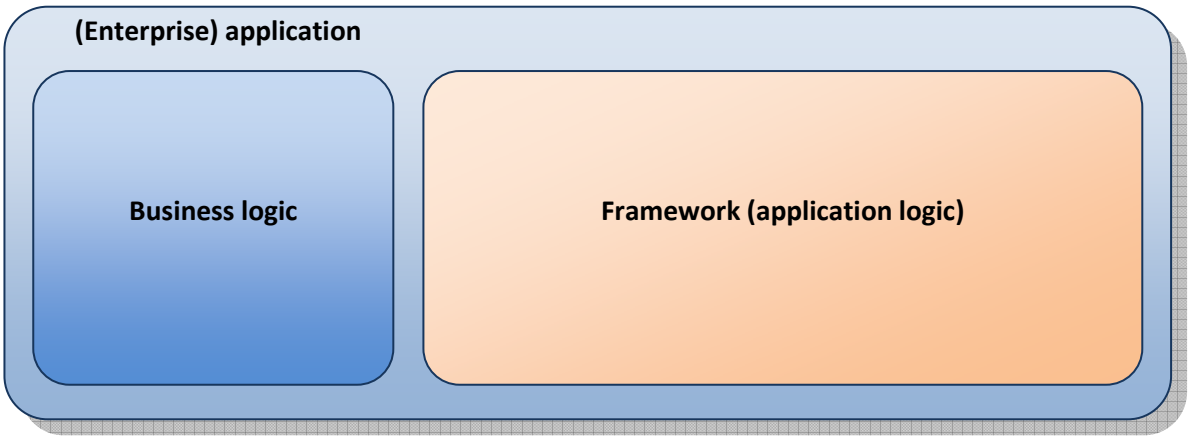


Figure 3.1: Illustration of the framework philosophy. Business logic is completely seperated from the framework and it only contains calls to the framework telling what to do. The business logic is translated in some form of scripting language, not programmed. Furthermore, the business logic may not contain any functionality. All functionality required for the application has to be located in the framework. One framework can be used for multiple applications, making mainainability easier.

Two situations are possible in the context of applications making use of one framework. The first situation is illustrated Figure 3.2; multiple applications share the same instance of a framework. This means that the framework itself is centrally located and the application business logic can access it. Similar applications can make use of shared libraries, these applications make use of a shared framework.

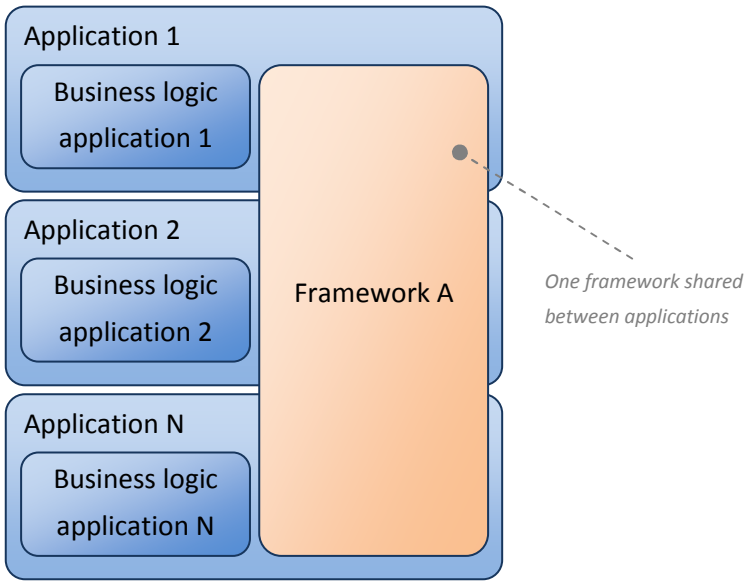


Figure 3.2: Applications share the framework. In this example N applications exist that share the same instance of the framework. The framework is located on a central place and only one instance of the framework exists. This gives the advantage only to have to update one version of the framework. Disadvantage is the fact that all application have to share a server that provides the framework; this is not always possible and it is possible that customers simply do not want it.

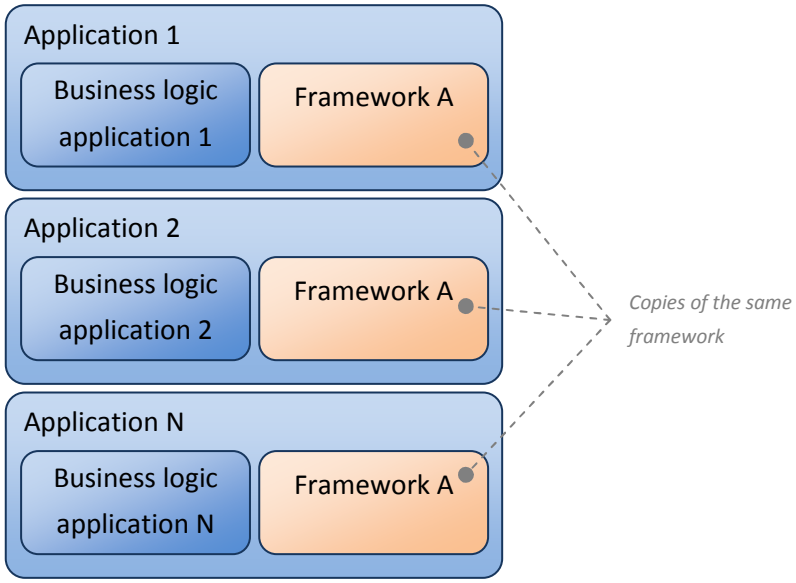


Figure 3.3: Applications have a copy of the same framework. Sharing a framework is either not possible or by customer demand that every application is working stand alone. In this philosophy example N applications exist that have their own instance of the framework. If a new version of the framework is released it should replace all the older versions of all applications, keeping all applications updated and ensuring that all applications have the same framework.

In the second situation multiple applications exist that operate completely stand alone. This might be the case when the software needs to be located at the customer or even by the simple

reason that a customer might demand it. In any case, it is not allowed to update a framework instance for one application. Some kind of framework management has to be applied to ensure that all the applications continue to have the same framework (see subsection 3.4.3). Framework management is necessary, otherwise it is possible that multiple different versions of the framework exist and the problems start all over again.

The philosophy goes further; customer demands have to be translated into business logic and business logic consists of scripts telling the framework what to do and when to do it. Only having to script makes development of a new enterprise application or updating an existing enterprise application easier and faster; due to the fact that the developer is limited by the scripting rules and does not have the freedom of a programming language. Making programming errors and bugs are therefore limited and, for the most part, only exist in the framework itself.

The framework has to offer a means to fast and easy 'implement' (i.e. script) business logic of a customer. This is different from other existing frameworks (see subsection 2.1.5) because the framework offers a form of high level programming language; with a few simple commands an complex application can be built. NextSelect core business process consists of analyzing and possibly optimizing customer business processes and translate this into enterprise software. Due to common changes in the business processes of customers the software often requires changes or new features. The framework philosophy optimizes the NextSelect business process by making the software development of enterprise application easier and faster. Furthermore, as the problems of NextSelect are assumed to be common for other software development companies, this philosophy can also optimize development processes of other companies.

The business logic of the customer can hold a lot of information. To keep the application well-ordered it is required to split the business logic up in to modular parts; which are called modules, see Figure 3.4. Each of these modules hold a piece of the business logic. It depends on the application itself how the business logic is split up. For example: the enterprise applications NextSelect develops are divided into different section and each section holds different business logic of the customer; the business logic of each section can be placed in a separate module.

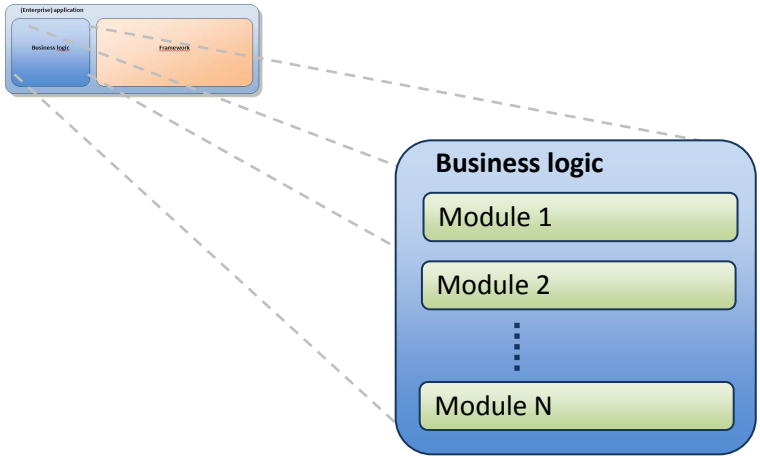


Figure 3.4: Business logic split into modules. A module contains only business logic, scripted and not programmed. Furthermore, a module does not contain functionality in terms of application logic, this is placed into the extensions of the framework (see Figure 3.5).

A module may only consist of business logic and it should not contain any functionality in terms of application logic. If it does, the philosophy is broken; due to the fact that maximizing code reusability cannot be guaranteed any more. Functionality placed in a module is not reusable for other applications that utilize the same framework (modules are not shared, only the framework with extensions). If one of those applications require the same functionality it has to be implemented again, hence leading to code duplication. To make the building of (enterprise) applications easier a module does not communicate directly to the extensions. It asks the framework to do 'something' in a plain and simple request. Examples are presented in subsection 3.2.1. The framework has to handle the request and transfer it to the extension responsible for the feature.

Splitting up the application into modules and just a 'framework' is not enough and does not comply with the objective that the framework has to be modular built (see subsection 2.2.3). Therefore the framework itself is split into two parts: the *Framework Core* and a unlimited amount of extensions, illustrated in Figure 3.5.

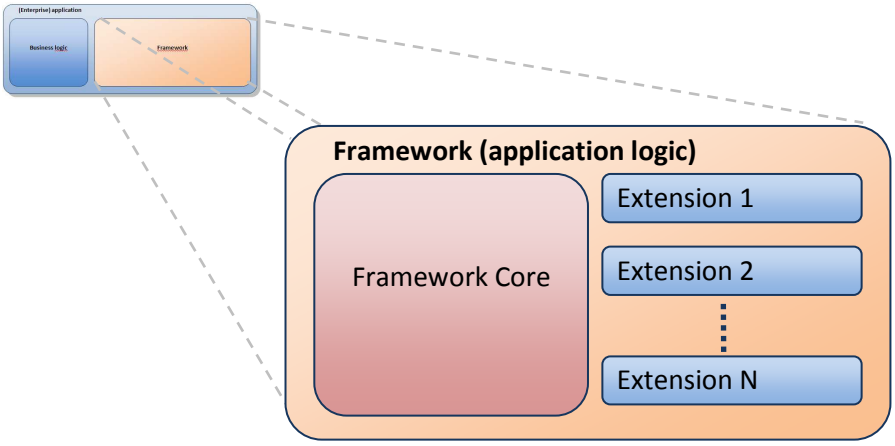


Figure 3.5: Modular framework. The framework is split into two parts. The first consists of the *Framework Core* and the second part contains all the framework extensions. The extensions contain all reusable functionality and can be seen as tools that have the responsibility to facilitate the framework and modules. The *Framework Core* manages the extensions and communication with the application.

Features are split up into modular parts called extensions. An extension can be seen as tool that has the responsibility to facilitate the framework and modules with reusable functionality. The framework may consist of an unlimited amount of extensions.

The *Framework Core* controls the framework and the extensions. The *Framework Core* is required due to the fact that modules communicate with the framework via a high level interface (see subsection 2.2.4) and the *Framework Core* takes care of this communication. Modules do not know anything about the underlying implementation that exists in the extensions. This has two reasons; modules are scripted not programmed and secondly, if no direct calls to extensions are used it is easier to modify the framework and maintain compatibility with older modules from older applications (see subsection 3.4.3).

To summarize the philosophy: the application is split into two parts: business logics and the framework, which can also be used by other applications. Business logic is divided into an unlimited amount of modules. Application logic in terms of features and functionality, without any business logic, is placed into an unlimited amount of framework extensions. Extensions hold the reusable part of the functionality offered. The *Framework Core* will manage the modules and extensions, and their communication. Modules tell the framework what to do without communicating directly to the extensions. A module is scripted, not programmed, it can be seen as a configuration part of the framework. A software developer only has to script or configure his whishes and the framework will take care of the rest.

Keep in mind that achieving this philosophy depends on the actual implementation of the framework. If the framework is not well equipped with functionality the philosophy can fail. Furthermore, a developer cannot be forced to follow the philosophy. However, rules and specifications to implement the framework are provided. An application satisfies the objectives

if the developer follows the philosophy, design (section 3.2 and 3.3) and rules and specification (section 3.4).

3.1.1. REAL LIFE EXAMPLE

Before more details about the architecture are discussed, a real life example is given next for a better insight about the frameworks philosophy:

Example:

> Imagine a DVD player, it has a lot of features, but most the important feature is playing a DVD. The device itself can be seen as a framework; the internal components provide high level features, such as playing a DVD or CD. These internal components can be seen as the extensions of the framework which are controlled by some sort of CPU, the *Framework Core*.
>
> The device itself does not operate automatically, it needs to be told what to do and when to do it. In this case a person has to operate the device, hence a person can be seen as a module. This person does not care how the device is working internally and how the internal components are communicating with each other. The person expects a high level interface (eject, play, rewind, etc.) to make use of the features offered by the device. If some high level input is given, the device has to process the request and deliver it to the appropriate components. Under water a high level request, such as 'play', results in a lot of low level internal calls between the CPU of the device (*Framework Core*) and internal components (extensions), e.g. read block from DVD, decode block, send video frame to output buffer, etc.
>
> Now this person bought a new Blu-ray disc and wants to play it in the DVD player. However, this DVD player does not support reading the new Blu-ray discs, hence it lacks a feature that the person wants to have. Of course the person is not going to read the Blu-ray disk itself (i.e. features may not be implemented at the module side), the person has to buy a new device that does provide the feature (i.e. the framework gets replaced by a newer version). This new device can still play the DVD's and CD's, but adds a new feature of playing Blu-ray's.

This example shows the exact philosophy but in another context. NextSelect wants to have this DVD example philosophy with the software it develops. Modules containing business logic tell the framework what to do and when to do it in high level calls. The actual functionality is located in the extensions, a module does not care how a feature is implemented and how the results are realized as long as the framework does what it is told to do.

3.2. ARCHITECTURE

The architecture of the framework is based on the problems, the whishes and most of all: the philosophy of the framework. According to the philosophy the application consists of two main parts: the customer specific business logic and the framework that can be used for multiple applications (see Figure 3.1). The part holding the business logic is split up into an unlimited amount of modules, each module representing a piece of the business logic. The framework part consists of a *Framework Core* and an unlimited amount of extensions. This is translated into the following architecture:

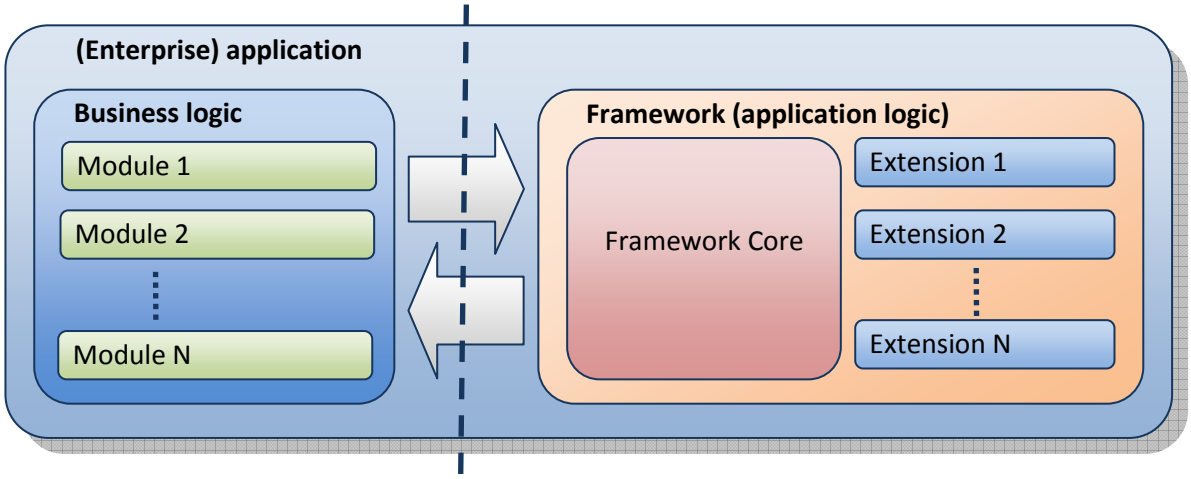


Figure 3.6: Framework architectural design. It separates business logic and framework (dashed line). Business logic is represented in the modules. A developer only has to script or configure a module, hence making software development easier. A module tells the framework what to do and when to do it. Features and functionality are put into extensions, which have the responsibility to facilitate the framework. The *Framework Core* manages the extensions and modules and takes care of the communications.

Modules do not contain functionality and are scripted by a software developer using simple calls that will tell the framework what to do and when to it. A module can also be seen as controller with a certain configuration that controls the framework. The *Framework Core* acts as a façade (software design pattern), transferring module request to the extensions responsible. A response may return from the framework; however, it will not contain complex data due to fact that modules may not contain functionality and can therefore not handle it (discussed in section 3.4).

Extensions contain all the features and reusable functionalities of an application. If the same concrete version of a framework is deployed in multiple applications (as illustrated in Figure 3.3) it is possible that certain features are not used. Features could be disabled or could be left aside, this is the choice of the software developer. In any case, the *Framework Core* has to know which extensions are available and the functionality they offer. Hence it is possible to transfer module calls to the responsible extensions.

The *Framework Core* controls the modules and extensions. It has the responsibility to process requests from the modules and find the right functionality in the features offered by the

extensions. How the *Framework Core* has to find functionality depends on the concrete implementation of the framework. This can be done in a simple way, e.g. finding a functionality by name, or in a very complex way where the module communicates in a natural language that the *Framework Core* can 'understand'. In any case, modules will not communicate directly with extensions, the *Framework Core* processes the requests of the modules and tries to find the required functionality within the extensions.

3.2.1. SIMPLE EXAMPLE

To demonstrate the framework a simple example is presented in Figure 3.7 and Figure 3.8. Although the request in this example does not represent a high level enterprise application request that the framework should handle, it gives a clear perception how the framework operates. For an example using a part of the enterprise application MIDS see Appendix F. The module *Person* has two operations for the framework. The first operation is illustrated in the Figure 3.7, the second operation in Figure 3.8.

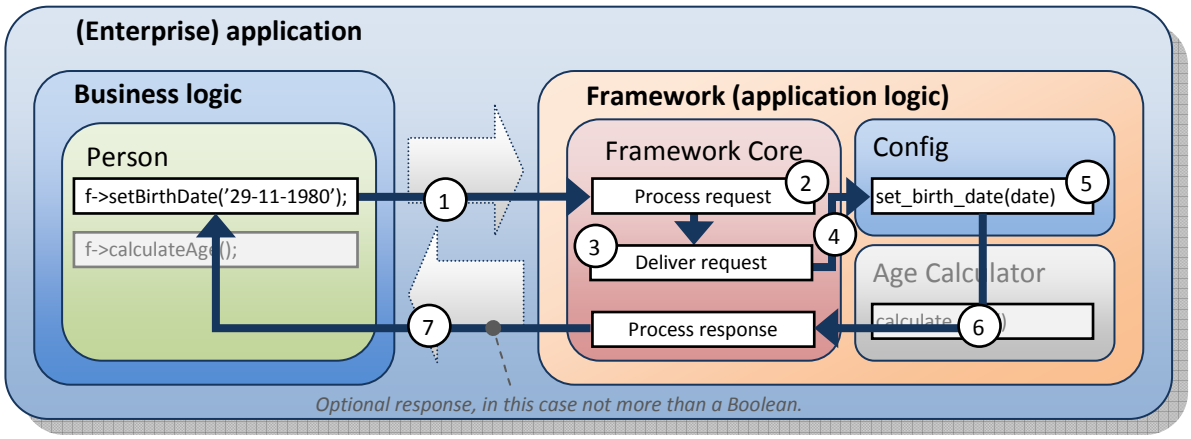


Figure 3.7: Small example (part 1). A module called '*Person*' tells (1) the framework to execute '*setBirthDate()*' with a given date. The *Framework Core* processes (2) the request, locates the appropriate extension (3) '*Config*' and delivers (4) the request. The extension executes the request (5), An optional response from the extension is sent back to the *Framework Core*(6) and module (7). The Framework Core's responsibility to load and register the extensions and module is not shown.

This example framework has two extensions: *Config* and *Age Calcucation*. The *Config* extension is able to store a birth date. Note: actual storage is not shown in the figure only a function to set the birth date. The second extension *Age Calculator* (inactive in this example, therefore grayed out) offers functionality to calculate an age with a birth date that has been set in the *Config* extension.

In the first operation the *Person* module tells the framework to set the birth date to '29-11-1980'. This request is processed by the *Framework Core* which tries to locate the appropriate extension that holds the functionality to actually set the birth date. In this case it is the *Config* extension. The request is delivered to the *Config* extension which will store the birth date in a local variable (not shown). The extension can send a response back to the *Framework Core*

which in turn can also send a response back to the module. This depends on the concrete implementation of the Framework Core and the extensions. A response back to the module has to be simple due to the fact that a module may not contain any application logic to process the returned data. In this example it could be a Boolean telling that the operation succeeded or failed.

In Figure 3.8 the second operation of the *Person* module is illustrated. In this case the module tells the framework to calculate the age, the framework already knows the birth date from the first operation.

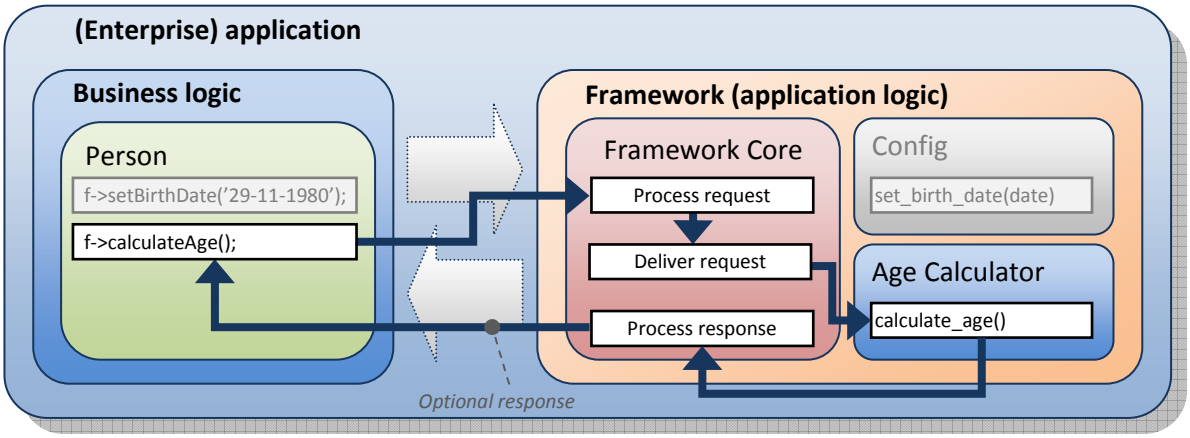


Figure 3.8: Small example (part 2). The module 'Person' now tells the framework to calculate an age, the software developer knows that the framework has a feature to calculate an age given a birth date. Again the request is processed by the *Framework Core* and it locates the appropriate extensions and delivers the request. The Framework Core's responsibility to load and register the extensions and module is not shown.

Important to see in the example is that the request (*calculateAge*) sent by the module *Person* does not correspond directly to the functionality offered in the extensions, e.g. the module requests '*calculateAge*' and the request is executed by the function '*calculate_age*'. The *Framework Core* is responsible to find functionality offered by an extension and deliver the request. A module does not have to know what happens 'under water' and which functions are used within the extensions, it only cares if the high level request is executed. In another scenario (discussed in subsection 3.2.2) it is even possible that multiple age calculation functions exist and the *Framework Core* has to figure out which one to use.

Also important to see is the implementation of the module. The module only has some simple requests for the framework. The framework has all the functionality to form an application, it just needs a module or multiple modules to tell it what to do.

In this example nothing is done with the actual age that is calculated. A third extension that offers the functionality to print values to a console or other output device could be added. The module could in a third operation tell the framework to output the calculate age. For a more complex and high level example see Appendix F for more details.

3.2.2. BROKER ARCHITECTURE

For the NextSelect enterprise applications the modules and extensions are all located within one application. However, this is not compulsorily as the architecture does not state how these parts have to communicate with each other. As a result it could be the case that for some implementation of the framework modules and / or extensions are not located within one application.

Imagine an application with a framework that uses extensions that are located on the internet. The Framework Core will handle all communication with the extensions. For a module this is transparent, i.e. as if the extensions are located within one application. A scenario could be that several age calculation extensions exist on the internet offering their services to any framework. One instance of a framework has a list of all these external extensions. Hence it can pick an extension as desired. The choice which one to choose is completely free, it could be based on a simple load balancing algorithm or based on the performance of these extensions (use the fastest available).

Note that having the same functionality spread over the internet seems to undermine the philosophy and the code reusability objective (subsection 2.2.8) due to the fact that code is duplicated in several extensions. This scenario forms an exception; although the external extensions are part of the whole application (see figure Figure 3.6) the implemented code is not duplicated on runtime level, it is only duplicated in a distributed environment. In section 3.4 this will be further discussed.

Due to the fact that the *Framework Core* has to 'find' functionality, it exhibits broker aspects that can be found in the CORBA and Web Service architecture. Especially when extensions or modules are distributed over a network. However, they are not the same. CORBA and Web Service are technologies designed to operate with different distributed systems over a network, they facilitate communications between different distributed applications. This is a key difference with the framework that facilitates software development, with the focus on enterprise applications.

The framework can be seen as a higher level architecture than the CORBA and Web Service architecture. The intended goal of the framework is to build applications, whereas CORBA and Web Services facilitate communication between distributed applications. Furthermore, CORBA does not offer a high level architecture like the framework does.

CORBA or Web Services can be used as a method for the framework components to communicate with each other. For example, a framework that uses extensions that are located on the internet. The communications between the *Framework Core* and the extensions could be facilitated by use of CORBA or Web Services.

For more details about CORBA and Web Services compared to the framework please see Appendix E.

3.3. DETAILED DESIGN

Designing a framework with functionality and at the same having the desire that same framework is deployable for a wide range of different applications is highly unfeasible, if not impossible. Hence the architecture of the framework is abstract and it does not state anything about a application it can be deployed for. During the prototype development this became clear, see Appendix A, Appendix B and Appendix C for more detailed information about this subject. Due to the objective that the framework should be fully object oriented (see subsection 2.2.2), the detailed design of the framework is represented in the Unified Modeling Language (UML).

3.3.1. BASIC COMPONENTS

The design of the framework, presented in subsection 3.3.3, is abstract and needs to be abstract, due to the fact that the framework design has to be application independent. This is supported by the objective that the framework needs to be deployable for a wide range of applications (see subsection 2.2.5). When parts of the framework are designed with application specific parts, the framework will lose its application independency resulting in a decreased range of applications the framework can be deployed for.

This research provides the basic abstract design of the framework. A software developer has to design its application purposes on top of the basic abstract design. To support this process a set of rules and specifications are provided (see section 3.4).

From the architecture, illustrated in Figure 3.6, the following classes can be derived: *Module*, *Framework Core* and *Extension*. This represented in the following UML diagram:

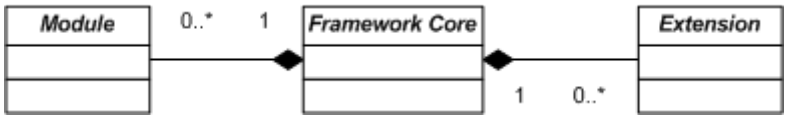


Figure 3.9: Basic compontents presented in UML. The class *Framework Core* can have an unlimited amount of reference to *Module* and *Extension* classes.

Looking at the interface level, extensions and modules basically contain the same functionality (see subsection 3.3.2). Duplicating code has to be limited, therefore a *FrameworkComponent* class (see Figure 3.10) is introduced that holds the basic functionality of the *Module* and *Extension* class (for more information about the derivation of the *FrameworkComponent* class, see Appendix B).

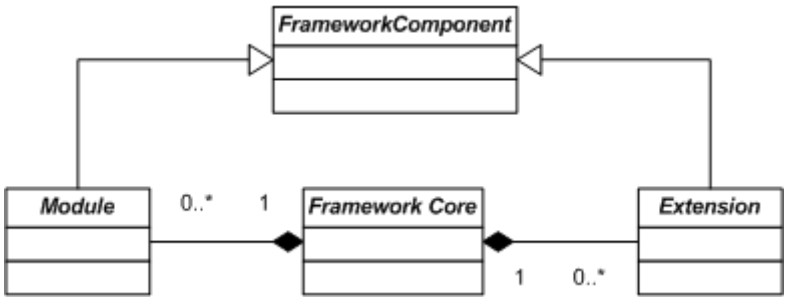


Figure 3.10: The *FrameworkComponent* class hold the common functionality of the *Module* and *Extension* class. Hence, code duplication is prevented.

The problem with the design, shown Figure 3.10, is the fact that the *Extension* class inherits from the *FrameworkComponent* class; in a concrete framework implementation it might be required that a certain extension class has to inherit from another class, hence leading to multiple inheritance. This is illustrated in Figure 3.11.

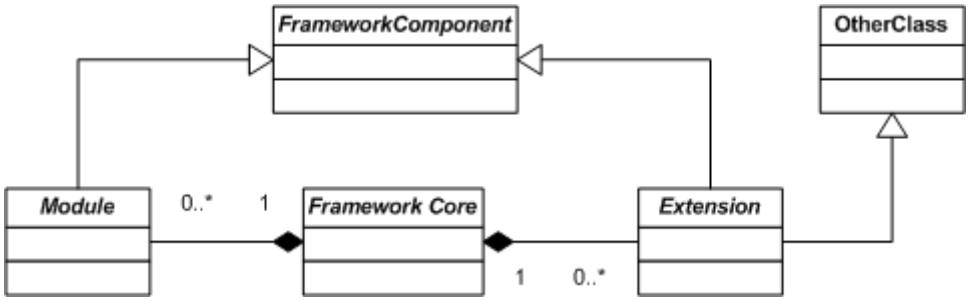


Figure 3.11: Mutiple inheritance example. The class Extension inherits from two classes: *FrameworkComponent* and *OtherClass*. Mutiple inheritance is not supported in all programming languages, e.g. Java.

Multiple inheritance using interface is not a problem, however, using multiple inheritance on classes does form an issue. Multiple inheritance with classes is not supported by all object oriented programming languages due to the complexity it can generate [13]. This also includes the popular Java and PHP, which is currently used by Nextselect to develop enterprise applications. This design would be in conflict with the objective that the framework has to be system and platform independent (subsection 2.2.6); multiple inheritance is only available in a limited amount of object oriented programming languages.

It is possible to simulate multiple inheritance [13]. Two aspects are important: parent class reuse and interface conformance. Parent class reuse means that the child class has all the functionality of its parents class, it inherits this functionality. Interface conformance means that the child class has a least the same interface as the parent class and can therefore act as if it is the parent class.

Parent class reuse can be simulated by using the delegation technique. An instance of the class that needs to be reused has to be instantiated in the child class, instead of implementing all the operations over again, the implementation only holds code that passes on the request to the instantiated class.

Interface conformance can be achieved by using interface classes. The class that needs to be inherit has to have a corresponding interface class. This interface ensures interface conformance. If at the same time the delegation technique is applied inheritance is simulated. With this method it is possible to simulate multiple inheritance.

There are some limitations when using multiple inheritance, e.g. protects members of the delegated class are not available. Furthermore, every class that might be inherited from needs to have a corresponding interface for the interface conformance. If no interface exists the programmer can create it, however this is only possible it the programmer has the source code of the class. The core Java API does not give the programmer this control. These limitations do not form a problem for this design of the framework, no protected data members are used and full control over the classes is available.

In the following figure an abstract example is illustrated that simulates multiple inheritance:

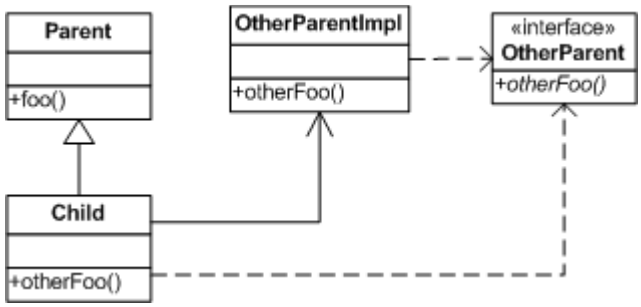


Figure 3.12: Simulating multiple inheritance example; The class *Child* inherits from the class *Parent* and simulates inheriting from the *OtherParent* class by using the delegation and interface conformance techniques.

The basic rule for simulating multiple inheritance to work is making sure that every class that might be inherit from, such as the *Extension* class also has a separate interface class. Translating the simulation techniques into the UML framework design result is illustrated in Figure 3.13. More information about the derivation of this diagram can be found in Appendix D.

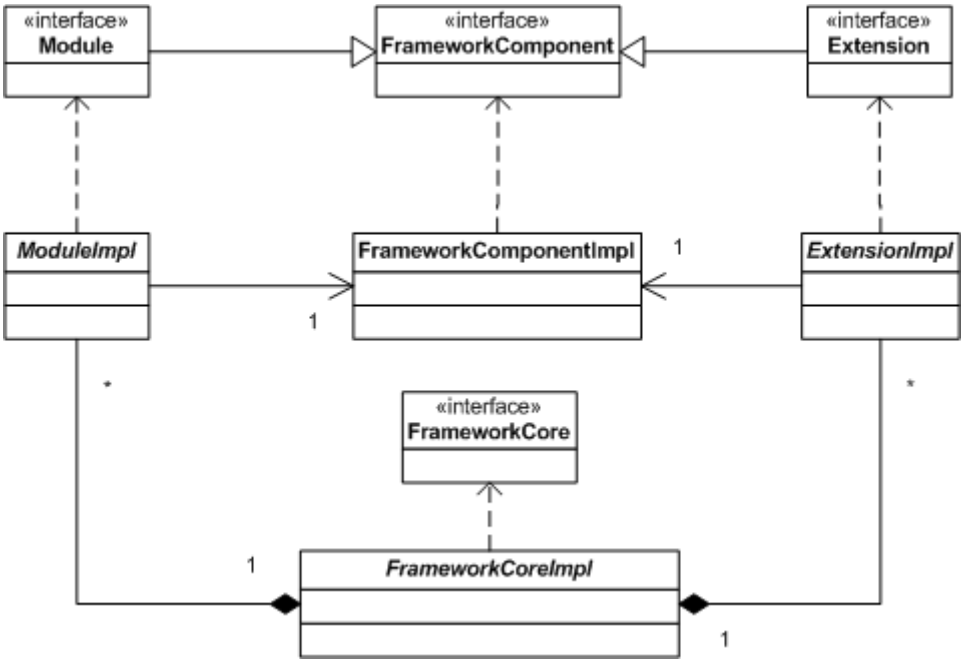


Figure 3.13: Final framework design showing basic components. It is ready for simulating multiple inheritance. All the classes that might be inherited from now have a corresponding interface class which makes it possible to use the delegation and interface conformance techniques. The classes are abstract (*italic* name) or interfaces (<<interface>> label).

3.3.2. CLASS METHODS

In the UML diagram illustrated in Figure 3.13 every abstract class (*italic* name) also has a corresponding interface, i.e.: *FrameworkCoreImpl* is an abstract class conforming to the interface class *FrameworkCore.* The abstract class and the interface class have the same basic methods in the UML diagram. These methods are described next, for more details on how these basic methods were derived from the research and development in all of the appendices.

Class FrameworkComponent and FrameworkComponentImpl:

- **setComponentName:** stores the name of the component.
- **getComponentName:** returns the stored name of the component.
- **setFramework:** stores a pointer of the *Framework Core.*
- **getFramework:** returns a pointer of the *Framework Core.*
- **initialize:** initializes the component. This is the so called 'second initialization point'. The first point is the constructor of the object. For detailed information about the two initialization points see subsection 3.4.9. In this class this method is abstract an needs to be implemented within a concrete *Module* or *Extension* class.

Class Module and ModuleImpl:

- **setModuleName:** set the name of a module. This function actually uses the *setComponentName* method of the parent *FrameworkComponent* class.

- **getModuleName:** returns the name of the module. Like the *setModuleName*, this method also passes the request to the parent *FrameworkComponent* class.
- **execute:** this is the location where most of the philosophy (see section 3.1) is based on; this is the place where the business logic has to be put in. The code located in this method will tell the framework what to do and when to do it.

Details about implementing a module and these basic methods can be found in subsection 3.4.6.

Class Extension and ExtensionImpl:

- **setExtensionName:** set the name of an extension. This function actually uses the *setComponentName* method of the parent *FrameworkComponent* class.
- **getExtensionName:** returns the name of the extension. Like the *setExtensionName*, this method also passes the request to the parent *FrameworkComponent* class.

Details about implementing an extension and these basic methods can be found in subsection 3.4.7.

Class FrameworkCore and FrameworkCoreImp:

- **loadModule:** tries to locate and load a module by a given module name.
- **registerModule:** if a module is loaded by the *loadModule* method, this function registers the module with the internal registers of the *Framework Core*.
- **getModule:** tries to locate a module with a given name in the internal register of the *Framework Core*; if found a pointer to the module is returned.
- **loadExtension:** tries to locate and load an extension by a given module name.
- **registerExtension:** if an extension is loaded by the *loadExtension* method, this function registers the extension with the internal registers of the *Framework Core*. Not only the extension is registered, also the functionality that the extensions offers. This is the point where the *Framework Core* gets aware of new features.
- **getExtension:** tries to locate an extension with a given name in the internal register of the *Framework Core*; if found a pointer to the module is returned.
- **initialize:** initializes the *Framework Core* before starting the application and in this function the second initialization point of all registered modules and extensions is called. More details about this initialization can be found in subsection 3.4.8.
- **run:** fires up the application; this function has to be called after all initialization are complete.

More details about implementing the *Framework Core* and these basic methods can be found in subsection 3.4.5.

Note that the Framework Core's basic design does not hold a basic methods for processing request. In short this is because the design of the framework is abstract and multiple methods of processing request exist. How these are implemented is up to the developer that implements a concrete framework. More on this subject with examples can be found in subsection 3.4.5 and 3.4.10.

3.3.3. UML DIAGRAM

Combining the UML diagram of Figure 3.13 and the methods described in subsection 3.3.2 results in an UML diagram that is illustrated in Figure 3.14; it forms the basic abstract design of the framework.

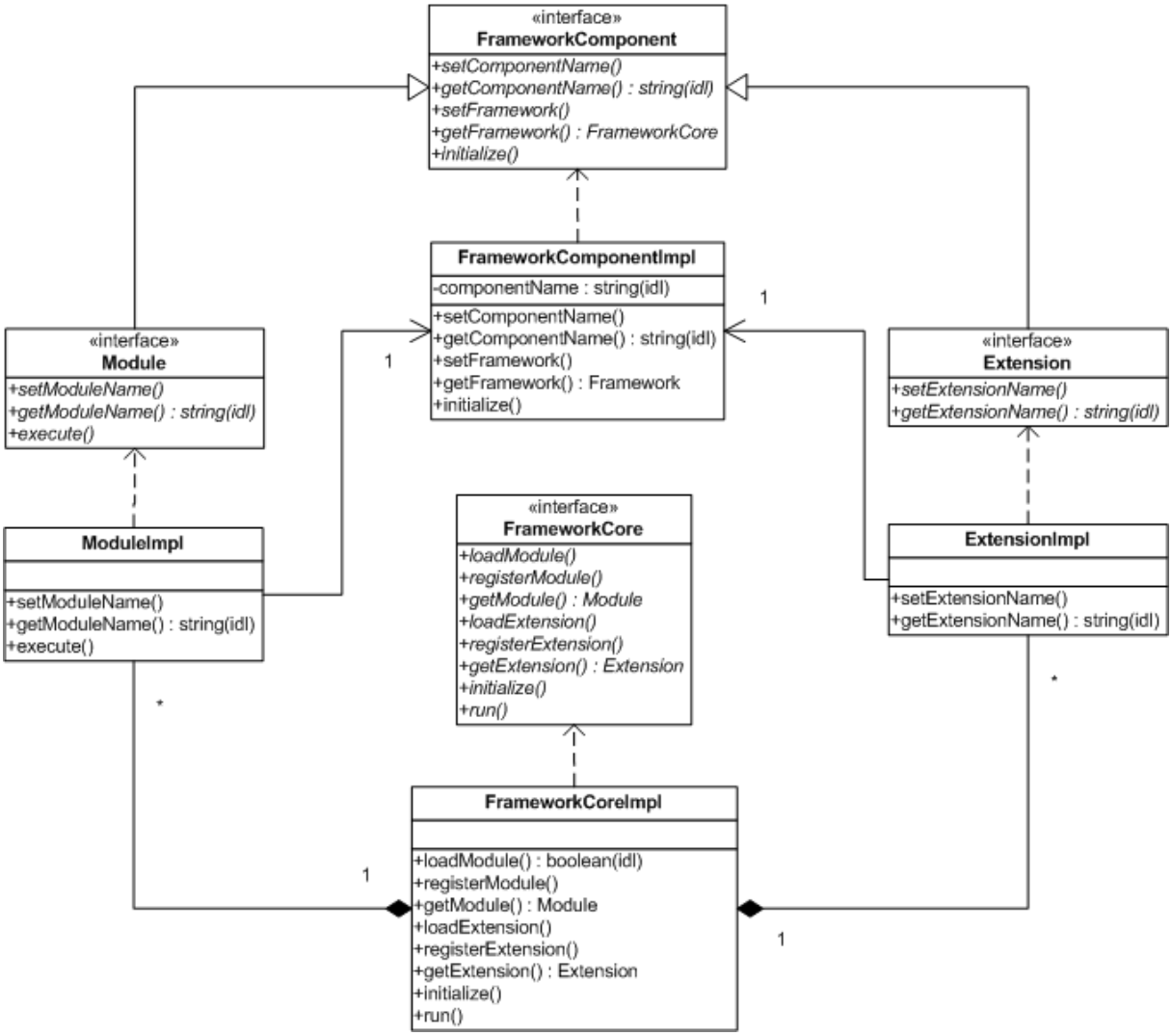


Figure 3.14: Final framework design showing basic components and methods. It is ready for simulating multiple inheritance. All the classes that might be inherited from now have a corresponding interface class which makes it possible to use the delegation and interface conformance techniques. The classes are abstract (*italic* name) or interfaces (<<interface>> label). This basic abstract design does not specify how requests are handled, see subsection 3.4.5 and 3.4.10 for more details on this subject.

3.4. RULES AND SPECIFICATIONS

The framework design itself cannot be used for building applications, it is abstract and intended as a basis for concrete frameworks which facilitate to build applications. The research provides a basic design, specifications and a set of rules that need to be followed when designing a new framework. If these rules and specifications are followed, the objectives (described in section 2.2) are guaranteed. However; keep in mind that, although the framework should be deployable for a wide range of applications, this research for the most part has been focusing on enterprise applications.

The development is divided into two phases. Framework development and application development. Each phase can have its own developers. Framework development consists of implementing the Framework Core and the low level reusable functionality located in the extensions. Application development consists of implementing modules using a high level scripting language provided by the framework. For the application development it is important to translate customer demands into business logic for the application. As a result there are high level developers and low level developers.

In the following subsections all important issues concerning the development of a concrete framework are discussed. These are not only advices but also rules and specifications that have to be followed in order to guarantee the objectives of section 2.2.

3.4.1. IMPLEMENTING A NEW FRAMEWORK

If a new framework has to be implemented for a specific range of applications (or even one application), the concrete parts have to be designed and important choices have to be made. This starts by mapping what kind of applications the framework is going to serve and what kind of functionality the framework has to offer the application. Once these choices have been made the amount of applications the framework can be deployed for decreases due to the fact that application specific choices have been made, i.e. a framework designed to build games cannot be used to build office applications. If it could, although highly unlikely but not impossible, the framework most likely get very large and cumbersome with extensions, due the fact that it has to serve many different applications that have no relation with each other.

As mentioned before, the philosophy of the framework is that programming within modules is limited; a module contains a script or configuration that tells the framework what to do and when to do it in a high level form. The business logic of the customer has to be translated into ‘simple’ calls for the framework to execute with the purpose of saving time when a new application has to be built. Hence, modules contain scripts that configure the framework instead of implementing real functionality themselves. This means that all application functionality is

located within the extensions. The choices that are made on how to design the extensions influence the behavior of the framework and hence the level of ease of implementing a module.

The software developer that builds or extends a concrete framework has to perform two main steps; first design the feature and all the components required like any other software development design process. Secondly the software developer has to build a high level interface for the feature in a way that a module can easily use it. The term 'easily' is ambiguous due to the fact that it is immeasurable. The developer has to keep in mind that the high level interface has to make sure the feature can be invoked with a limited amount of calls (with the exception of feature configuration settings, see Appendix F for a concrete example of configuring the framework from a module). What helps is to see extensions as tools that have the responsibility to facilitate the framework and the modules. If somebody uses a tool to achieve a goal, the tool needs to be as easy as possible for usage.

To understand the high level concept that has to be maintained within the modules and offered by the extensions, please see the example given in subsection 3.1.1.

Next to the high level design of a feature the design also has to take into account that it might be used for other applications. Of course it is impossible to take all scenarios into account, however by looking at the NextSelect case a lot of repetition of code was discovered due to the fact that features were implemented with too much reference to the business logic of the customer. Therefore extensions have to be configurable in such a way that the feature offered by the extension can be configured to meet the demands of the customer. It depends on the software developer's judgment if the configurable parts of the extension are general enough for the range of applications the concrete framework has to be deployed for.

3.4.2. FORMULATE DEVELOPMENT RULES

When a software development company wants to use this framework for application development, it has to design and implement a concrete version first. If the design and implementation follow the rules and specification, the objectives of section 2.2 are maintained and the problems described in subsection 2.1.2 avoided. However, in order to maintain the objectives during further development of the concrete framework, the software company has to formulate a set of development rules for its developers.

The reason for this is simple; the software development company has to make implementation choices for the concrete framework (e.g. communication between modules, core-, extension- and feature implementation, high level scripting language format, etc). Furthermore, the concrete framework most likely will grow in time with complex structures. Software developers come and go, if a new developer joins the software development company (for application development, framework development or both) the developer needs to be trained with the

framework. Therefore the software development company has to formulate rules for their own concrete framework for developers to follow with the goal to preserve the objectives.

Due to the fact that the development is split up into two phases it is advisable to do the same for the development rules. This results in a set of rules for the low level framework development and a set of rules for the application development.

Low level development rules

Low level framework development consistsof the development of the *Framework Core* and the extensions. Some examples on which a software company has to formulate rules for its developers:

- Communication between extensions (see subsection 3.4.10);
- Adding new features in existing or new extensions;
- When to implement a new feature (new feature required, does it already exists?);
- Implementation of the high level interface that features have to provide for modules;
- How to maintain backwards compatibility with old modules.

High level development rules

High level development consists of the development of modules. Modules may only contain a high level interface scripting language that tells the framework what to do. The framework extensions provide the high level interface. Developers need to know how they can use the high level interface, what is allowed and how features had to be configured. Furthermore, they have to know the scripting language itself, even if it is in the most simple form, due to the fact that the business logic has to be translated into the scripting language. Some examples:

- The high level scripting language; how are modules implemented, what features are available, how are the features configured (e.g. a manual for the application developers).
- Adding new required business logic in existing or new modules;

3.4.3. MAINTAINING THE FRAMEWORK

When a concrete instance of the framework has been realized and deployed in one or more applications another important aspect arises: maintaining the framework. For deployment two scenarios are possible: the concrete framework is shared between different applications (illustrated in Figure 3.15) and the concrete framework exists in different applications (illustrated in Figure 3.16). It is also possible that a combination is used; some applications sharing the same framework and others having their own copy of the framework. In any case, when the framework requires an update (e.g. due to a new feature) it will have an influence on all the applications using the framework. Therefore this process must be well coordinated.

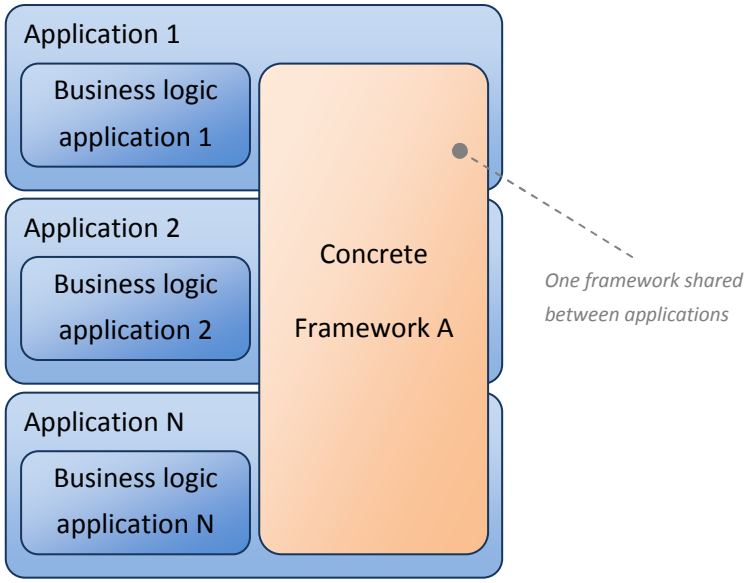


Figure 3.15: Applications share the concrete framework. In this example N applications exist that share the same instance of the concrete framework. The framework is located on a central place and only one instance (source code) of the framework exists. This gives the advantage only to have to update one version of the framework. Disadvantage is the fact that all application have to share a server that provides the framework; this is not always possible and it is possible that customers simply do not want it.

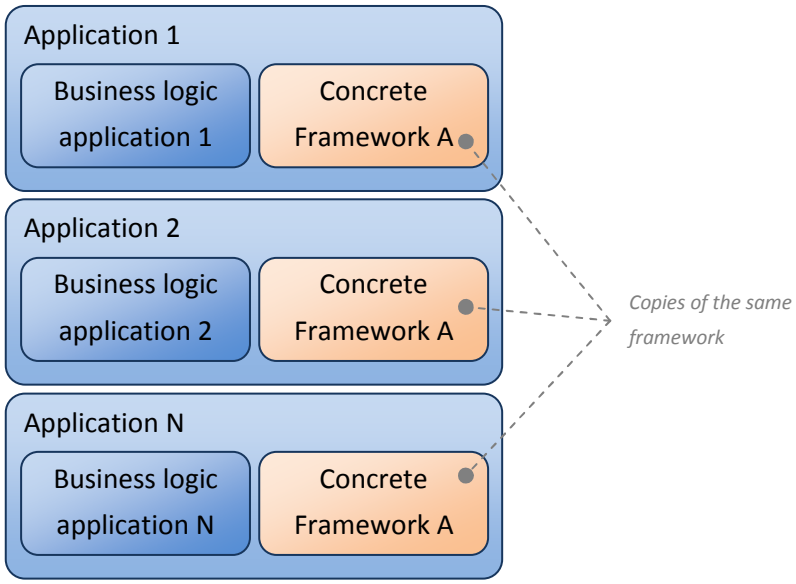


Figure 3.16: Applications have a copy of the same concrete framework “A”. It is either not possible or by customer demand that every application is working stand alone. In this example N applications exist that have their own instance of the framework. If a new version of the framework is released it should replace all the older versions of all applications, keeping all applications updated and ensuring that all applications have the same framework.

It is important to have a separate framework development tree next to the application development trees (e.g. the software development company could use separate SVN repository tree for the framework). When a new version of the framework is released it should be backwards compatible with all existing applications running the old framework. In the first scenario the update of the framework is the quickest due to the fact only one central framework needs to be replaced.

Updating the application in the second scenario is a different story. As easy as the architecture of Figure 3.16 may look, it can get complex when for every application the framework has to be updated with the new version of the framework. This process takes more time than updating only one framework. Furthermore, the framework extensions might not be available for every customer (due to licensing, etc.), this means that for every customer the framework has to be configured with the right extensions.

3.4.4. MULTIPLE FRAMEWORK DEVELOPMENT TREES

The main idea of managing a concrete framework, as described in the previous subsection 3.4.3, is that one version of the framework is maintained to build the same kind of application for different customers demanding different parameters (business logic). This process is illustrated in Figure 3.17. The reason why only one framework development tree should be maintained is that if multiple versions of the "same" framework exist the philosophy is broken and the problems that resulted this research will start all over again (see subsection 2.1.2).

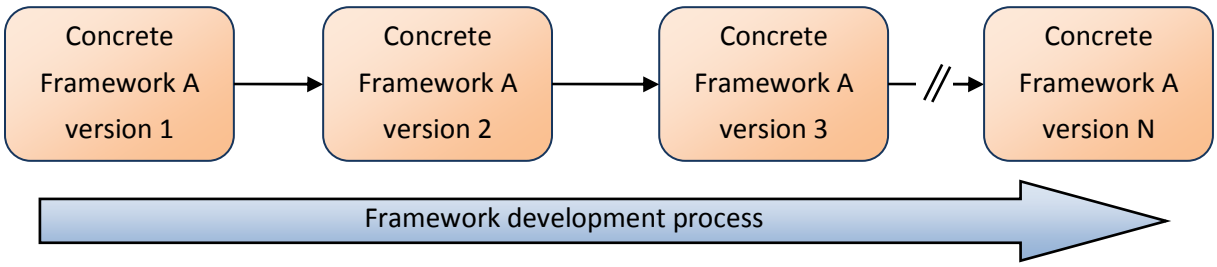


Figure 3.17: Single framework development tree. Concrete framework "A" starts with version 1, then gets an upgrade to version 2, then to version 3 and finally version N. Only one framework is maintained in this example.

There are two exceptions when multiple development trees are allowed. The first is trivial; if two completely different frameworks exist for different kind of applications, this is illustrated in Figure 3.18. Different kind of frameworks are for example a framework for building ERP systems in Java and another framework could be for building board games like Tic Tac Toe in C++. Two completely different frameworks where it is impossible to reuse code.

For the second exception (illustrated in Figure 3.19) it is allowed to split up an existing framework temporarily into two or more development trees with the goal to merge them to one in a later stadium. The split up could be for testing reasons or to develop a specific part of the framework without interfering other development trees (e.g. using multiple developers at once on the same framework). When the development is finished the trees are merged.

Important to note with the second exception is that the temporary frameworks are not used for application development, it is only used for the development or testing of a new or existing part without interfering with other development processes.

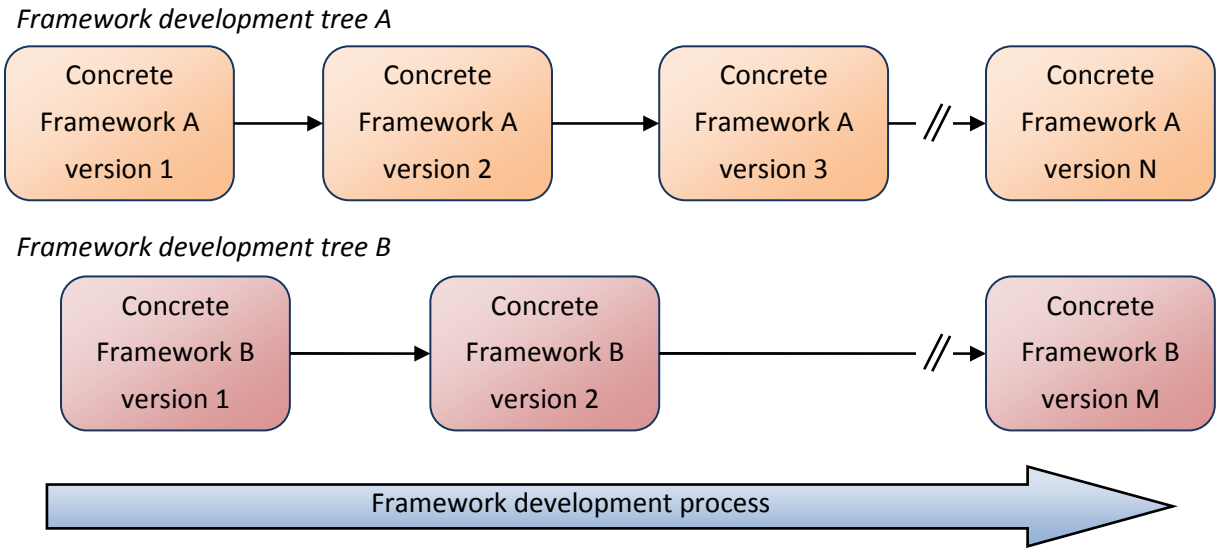


Figure 3.18: Multiple framework development trees. In this example two development tree are maintained; one for framework "A" and one for framework "B". These frameworks are different and used for different kind of applications. Therefore the two development trees are allowed. For example: Framework "A" is for building enterprise applications and framework "B" is for building Tic Tac toe games.

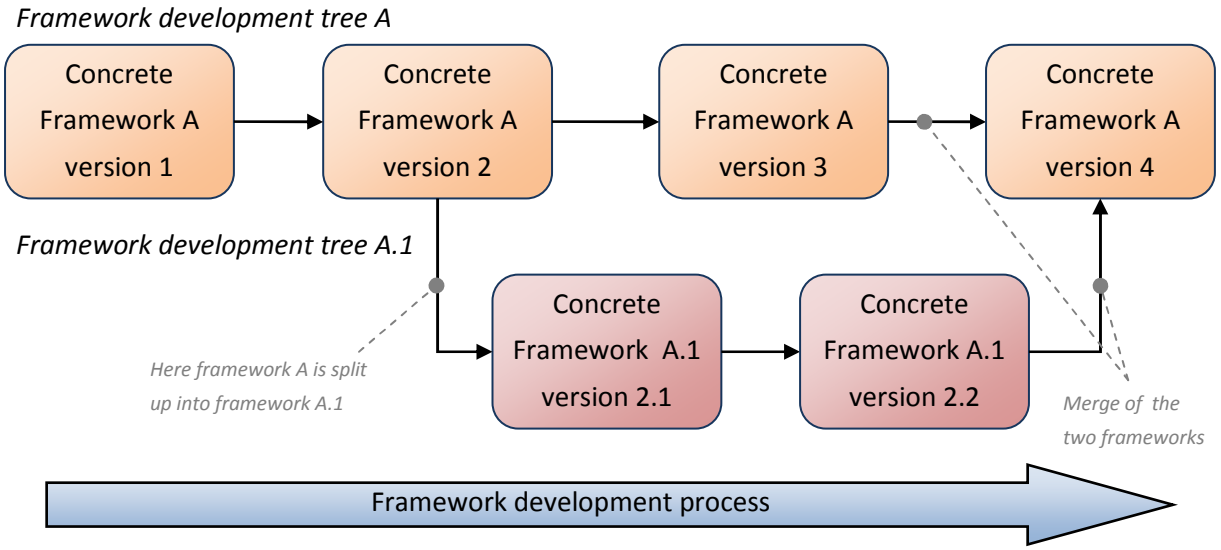


Figure 3.19: Temporary multiple development trees. In this example framework "A" is split up into framework "A.1". Framework "A.1" will be used to develop a new or existing part to be later merged with the original framework "A". The development of this new or existing part is put in a separate development tree with the purpose not to interfere with the development process of framework "A".

One of the main problems with the NextSelect case is that an existing framework was split up for every new application. This resulted in multiple frameworks that for the most part were the same. More frameworks means more time spent on the management of these frameworks which can lead to other problems like making mistakes. Duplication of frameworks without the intention to merge them leads to (massive) duplication of code and this is therefore not allowed (illustrated in Figure 3.20).

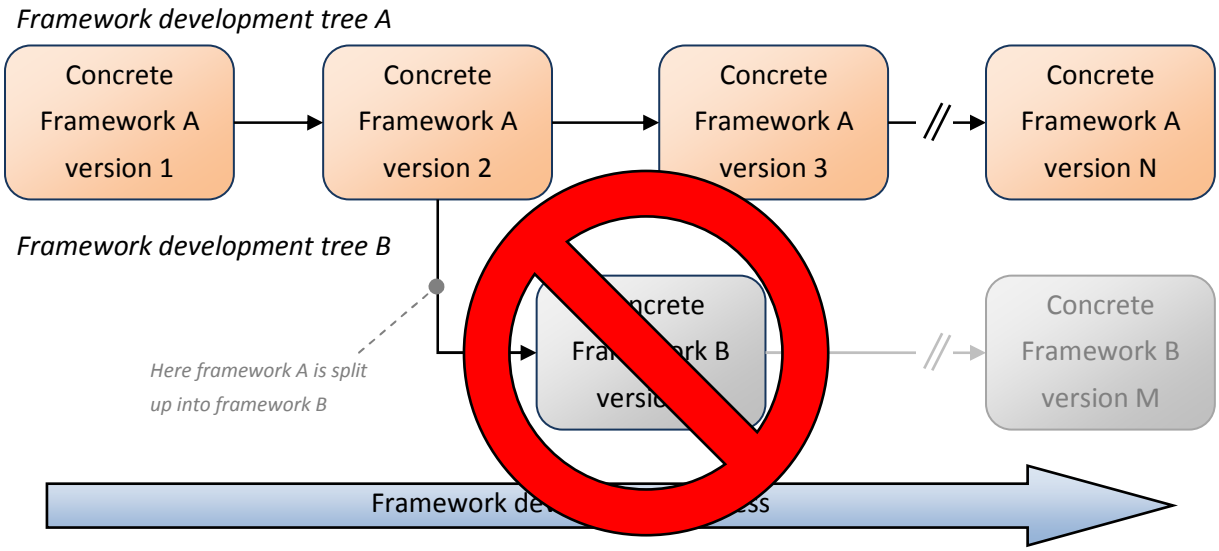


Figure 3.20: Duplication of a framework that is not allowed. In this example gets framework "A" split up in some point of the development process. The split up generates a new framework "B" that uses framework "A" initially as a basis. Due to the fact that framework "A" and "B" share the same code this is not allowed.

To summarize, in only two scenarios is it allowed to have more than one framework development process:

- When different frameworks have to be maintained for different applications;
- When a temporary framework is required to implement/test a new or existing part without interfering with the normal development process of the framework.

3.4.5. IMPLEMENTING THE FRAMEWORK CORE

The *Framework Core* forms a important part of the framework, it binds all the components together and controls the loading, initialization, registration and termination of the modules and extensions. In Figure 3.7 and Figure 3.8 a simplified representation of the framework is represented which assumes that the loading, initialization and registration of the extensions and modules has been performed by the *Framework Core*.

The kind of application, platform, programming language and the design choices that have been made in subsection 3.4.1, influence how the core has to be implemented.

The *Framework Core* mainly has the following four responsibilities:

1. **managing communication between all the components:** like processing the request of a module. For more details about this part see subsection 3.4.10.

2. **Loading modules and extensions:**

For web based applications

As stated before it is likely that the framework will grow with functionality in time. Hence the number of extensions will grow. Also the application may grow leading to more modules. Depending on the application type, runtime application (C++ or Java) or web based application (PHP), requires different approaches.

A web based application, built for example in PHP, does not keep running. The user triggers a request through its web browser which in turn sends the request to the web server that runs the application. Within the server a new PHP process is started which will run the PHP program. In normal runtime applications, the process keeps running until the user closes the application, but with this web based application the process is finished when the request output is sent back to the client's web browser. This means that the PHP program is started and closed immediately, this also has to be done as fast as possible because the user does not want to wait too long for his request. Hence it is important only to load the modules and extensions required to process the request of the user. Loading unnecessary modules or extensions takes more time and also requires more memory usage.

In Appendix F a possible solution is presented. The *Framework Core* loads only the modules and extensions it always requires upon initialization of the framework. One of those is an extension that analyzes the request of the user and determines the modules and extensions required to process the request. Every extension will also indicate if it depends on another extensions. Using this approach only the required modules and extensions are loaded for a specific request. As a result the web based application is stays optimized even if the number of extensions grow (what will happen).

For normal run time applications

For a normal run time application, an application that will not stop until the user says it to stop, the approach is different. Unlike the web based approach it is not known at initialization point of the framework, which modules and extensions are required to process a request. This is unknown due to the fact that the applications keeps on running and multiple request can be received at different times.

One solution is to load all the modules and extensions. However this is not very efficient, especially when the framework is relatively large, it will consume more memory and it has a negative effect on the framework's performance.

A better solution is to load the modules and extensions dynamically at runtime. This means that when a request is sent to the framework the *Framework Core* has to check if the appropriate module or extension is already loaded, if not it will try to load it. After a certain point of time the module or extension can be unloaded to free memory and other resources.

Dynamically loading modules or extensions at runtime means that the *Framework Core* has to know the functionality of the modules or extensions before they are loaded. The developer has to take into account that the *Framework Core* has to know functionality of all the modules and extensions in order to use them. One method is to load all modules and extensions when the application is started so that they can register their functionality with the *Framework Core*, without calling the second initialization point (see subsection 3.4.9 for more details about the initialization points). The modules and extensions not directly required can be unloaded to free memory and resources at a later point in time, e.g. like the garbage collector of Java.

Note that when a module or extension is loaded at runtime the *Framework Core* will not automatically call the second initialization point. This either has to be done manually or the *Framework Core* has to be adapted.

3. **Register functionalities:** the *Framework Core* has to know what functionality is available in order to process a request from a module and deliver it to the appropriate extension. The developer has to implement a data structure to map functionality to extensions and means to register the functionality. It depends on the level of communication (see subsection 3.4.10 for more information about the communication between modules and extensions) how complex this implementation will get. In Appendix F sample approaches are given.

4. **Check dependencies :** extensions can have dependencies, i.e. certain functionality within an extension depends on a functionality from another extension. The programmer has to provide the *Framework Core* with a method for extensions to indicate their dependencies. The *Framework Core* is able to check if all the dependencies are met upon initialization of the framework.

3.4.6. IMPLEMENTING MODULES

Extensions are implemented as tools with the responsibility to facilitate the framework and modules with reusable functionality. Modules are implemented with business logic telling the framework what to do and when to do it. They are not supposed to have actual application code according to the philosophy of the framework. A module should contain a script telling the

framework what to do in a certain point in time with high level calls. This can also be seen as configuring the framework and the features offered to form the actual application.

It is allowed the use the same programming language, which has been used to implement the framework, to implement the a module. Important here is to make sure no functionality for the application is added to the module. Statements like 'if', 'else', 'switch' are allowed, even the implementation of functions is allowed. As long as no new functionality is added the developer is free to use the tools the programming language provides.

The term 'new functionality' is ambiguous due to fact that the same programming language of the framework may be used to implement a module. Is, for example, the subtraction or addition of two values new functionality? The given example is allowed, however a grey area on this matter remains. The following statement can give some enlightenment:

> ***An implementation in a module can be seen as 'new functionality' if the implementation tries to manipulate any kind of data. Manipulating data may only be done in extensions.***

The problem with adding functionality to a module is the fact that it will tear down the objective of maximizing code reusability. If functionality is implemented in a module it means that it will only be available for one customer/application. If another customer requires the same functionality it has to be implemented over again, resulting in two similar implementations. This is in conflict with the philosophy of the framework that wants to maximize code reusability.

Even though the programming language of the framework may be used to implement modules, another option is to offer a real scripting language. The framework had to offer this scripting language and an interpreter that 'understands' the scripting language and is able to translate a script into framework calls. Using this option to offer a scripting language to the application developer supports the framework philosophy better due to the fact that the application developer has to maintain the rules of the framework and is not able to implement new functionality.

The interpreter could be located in the *Framework Core*. However, in our opinion it seems better to put the interpreter in an extension because it is not a responsibility of the Framework Core to interpret a scripting language and if it is put in an extension more interpreters could be added as features. The application developer has the possibility to choose an interpreter for a specific application. Please note that actual scripting in a module is marked as future work, as in this thesis no actual research was performed on the possibilities of an interpreter.

3.4.7. IMPLEMENTING EXTENSIONS

Extensions facilitate the framework and modules with one or more features and contain all the reusable functionality of the application. Modules feed the extensions with 'information' how to act. Due to the fact that this has to be done with simple calls the extensions have to be implemented offering some form of high level interface. Underwater they can be as complex as possible as long as the interface for the modules is plain and simple.

It is the developers choice to implement multiple features within one extension or to spread them over multiple extensions. The features could be categorized on the functionality, importance and size. It is best to state the rules when to implement certain features in one extension or in multiple extension when the framework is designed. However these rules might change during the development process of the framework resulting in splitting up existing extensions or merging multiple extensions to one. This should be no issue due to the fact that the *Framework Core* has to find functionality, therefore it does not matter in which extension the functionality is located and will the changes not influence the existing modules that use the framework.

The objective Maximizing code reusability (subsection 2.2.8) is applicable and important. It becomes harder and harder to maintain this objective considering that extensions contain all reusable functionality and it is most probable that the framework will grow as time progresses. When a framework developer needs to implement a new required feature it will most likely make use of existing functionality of other extensions, how does a framework developer know which functionality is already available? If a framework developer has no knowledge of existing functionality the developer will implement the functionality over again, hence code is duplicated and the objective to maximize code reusability is not maintained.

It is impossible to guarantee 100% code reusability, especially when the framework consists of hundreds of functions spread over tens of extensions. Code reusability has to be maximized and code retrieval plays an important role here. However, not much research in the literature has been performed to cover the subject of code retrieval (a form of information retrieval) and in this thesis it is marked as future work since information retrieval is hard. Nevertheless a part solution is provided: the use of extension data types.

Extension data types are simple classes that have to replace all the standard programming types like integer, double, real, boolean, char, etc. The extension data type class will not only represent a value as the standard programming type does, it also labels the value. Example to store an age of a person an *Age* extension data type class can be used. *Age* is an object representing the integer value of an age, whereas normally only an integer would be used. The code retrieval results can now be narrowed down by searching for specific functions that, for example, only accept *Age* as input or return *Age* as output instead of searching for functions that accept an integer as input or output. More details on this subject can be found in section 4.3.

3.4.8. FRAMEWORK CORE INITIALIZATION

The Framework Core has two initialization points; the first, because it is an object class, is the constructor. When the *Framework Core* gets instantiated the constructor is called automatically. The second initialization point has to be called manually, hence it needs to be hard coded.

In the first initialization point (constructor) it is possible for the framework developer to specify default extensions and even modules that need to be loaded when the application is started.

At the second initialization point the *Framework Core* has two responsibilities. The first responsibility is to check if other modules are required to be loaded. This information might be provided by an extension, see Appendix F for an example where an extension of a web-based application processes an incoming URL and can tell which corresponding module has to be loaded. The second responsibility is to call the *initialize()* function of all modules and extensions (see subsection 3.4.9 for more details about initialization modules and extensions). If the information is not available through an extension the Framework Core has to provide its own means to determine which module has to be loaded. Depending on the type of application it might even be the case that all modules have to be loaded.

The framework developer is free on how to implement the method that will determine which modules have to be loaded, because this depends on the type of application the framework has to be deployed for.

3.4.9. INITIALIZATION OF EXTENSIONS AND MODULES

Modules and extensions have, just like the *Framework Core*, two initialization points, see subsection 3.3.2. The first, because it is an object class, is the constructor. When the module or extension gets instantiated the constructor is called automatically. The second initialization point is when the *Framework Core* has finished loading all modules and extensions, after this the *Framework Core* will call for all modules and extensions the *initialization()* function (see subsection 3.4.8).

A module or extension might depend on another module or extension for initialization. However, at constructor initialization (the first initialization point), it is not guaranteed that the dependency is already loaded. This means that calls to the framework might not be executed, simply because the functionality is not loaded. At the second initialization point this is guaranteed.

The developer of the framework (and enterprise application) has to take the above into consideration when implementing the framework components. Requesting the *Framework Core* to load a module or extension is best placed at constructor initialization of modules and

extensions, due to the fact that this ensures all required modules and extensions of the application are loaded before *the Framework Core* will start the second initialization.

Calls to other extensions, requesting functionality or 'telling' the framework what to do, are best placed in or after the second initialization point, due to the fact that the module or extension placing the request is ensured that the dependency (extension providing functionality) is loaded and available.

However, there is an exception to this rule; asking for functionality is in some cases required at constructor initialization (an example can be found in Appendix F). This is allowed as long as the framework developer can ensure that the dependency is loaded at the time the constructor is called. To ensure this knowledge about the sequence of loading the extensions and modules is important. The framework developer has control over this sequence when implementing the framework (i.e., this sequence can be hard coded).

For example, the framework loads some extensions and a module that holds configuration for the framework (framework configuration specific for a customer is placed in a module). The module has some calls for the framework which will set some configuration settings, like the database that needs to be used and credentials for connecting to this database. Suppose that the module will follow the rule and places these calls in the second initialization point. This would mean that all required extensions and modules are loaded and their first initialization points are finished at the time the configuration module's second initialization point is called. What if some of those extensions required some configuration settings at their first initialization point? For example an extension that sets up the database connection in the first initialization point. They would fail due to the fact that these settings are not available yet. This is a typical "chicken or the egg" problem.

To resolve this the framework developer has to ensure that the right sequence of loading modules and extensions is maintained. In this example the framework developer first has to load the extension that processes and stores the configuration, then the configuration module and at last the extension that will set up the database connection at the first initialization point. Additionally, the module's configuration calls need to be placed in the first initialization point (constructor) of the module. These calls will not fail if due to the fact that the extension that needs to process these calls is already loaded.

The reason why this needs to be implemented hard coded by the framework developer is simple; if not the dependency is not guaranteed and initializing modules and or extensions might give unpredictable results as it unsure when certain modules or extensions are loaded.

A situation that should be avoided is when two extensions depend on each other during the second initialization point. For extension A to be able to initialize it requires functionality offered by extension B, however this functionality is unavailable because extension B is not yet

initialized. Extension B in turn requires functionality offered by extension A to be able to initialize, however this functionality is also unavailable because extension A is not yet initialized. In practice the application will fail when the Framework Core will call the second initialization point of extension A. This situation is not a problem for modules due to the fact that modules are not allowed to contain any functionality.

3.4.10. COMMUNICATIONS

Modules and in particular extensions have to register their functionality with the *Framework Core*. Following the philosophy of the framework modules will not communicate directly with extensions. They have to send the request through the framework, i.e. telling the framework what to do. The *Framework Core* has the responsibility to be process an incoming request and delivered it to the appropriate extensions. The actual implementation of processing and delivering a request depends on the implementation of the framework. The framework developer is free to implement this in any way, as long as the rules and specifications are not broken. A concrete example can be found in Appendix F; a prototype was built with an internal data structure containing all the functionality and reference to the extensions and modules was used. When a request of a module is received the *Framework Core* looks up in which extension the functionality can be found.

In a more advanced implementation of the *Framework Core* it might be possible, for example, that functionality is not directly asked by name, like in this prototype, but only by input and output parameters. Using only the extension data type method (see subsection 3.4.7) as input and output parameters, it is relatively easy to search for functionality; the *Framework Core* iterates through its own internal repository for an extension that offers a function with the same input and output extension data type parameters. In the framework that NextSelect is going to implement this is not really desired due to the fact that it is not sure which extension and functionality are going to be used and this can give unpredictable results. However, in another type of application with a different framework it might be desirable to let the *Framework Core* search for the best functionality. For example: a framework that is using web services that are located on the internet as extensions. A module requesting functionally will trigger the *Framework Core* to find an appropriate extension that offers the requested feature. Because the extensions are implemented as web services it is not guaranteed that they are always available, e.g. an extensions might be unreachable because they could be offline so the *Framework Core* has to try to locate an extension that is online with the requested functionality.

Another example is that modules are built with some kind of scripting language that the *Framework Core* is able to interpret. This is actually the ultimate goal of the framework philosophy, to provide a framework that is able to be told what to do in a natural scripting language. The interpreter could be located in the *Framework Core,* a better solution is to add an

extension providing functionality to interpret a particular language. This allows to add more language interpreters if required and keeps the *Framework Core* clean.

For the extension it is a different story. Just by looking at the functionality of the existing frameworks of NextSelect, it is clear that rebuilding a new framework conform specifications of the new framework design will generate a lot of extensions. These extensions not only have to facilitate modules, they also have to facilitate other extensions.

Extensions are able to generate thousands, if not tens of thousands requests to other extensions just to handle one request of a module. If extensions are required to ask the *Framework Core* for functionality in a high level interface scripting language, just like modules are obligated to do, it will create a lot of overhead. Furthermore, extensions consist of low level implementations by low level framework developers. Creating the overhead is therefore not necessary, hence it is allowed that extensions have direct communication with other extensions without having to communicate through the *Framework Core* by means of direct class method calls. Using direct class methods between extensions instead of high level interface calls is allowed. This is however a choice for the framework developer to make and put in a development rule (see subsection 3.4.2).

To conclude, the actual communication between modules and extensions depends on the concrete implementation of the framework. Module have to communicate though the *Framework Core* for requesting functionality, this can be done in the same programming language or in a more advanced (scripting) language that the *Framework Core* is able to interpret by itself or by using interpret extensions (see subsection 3.4.6). Extensions are allowed to communicate directly to each other by means of direct class method calls instead of using the high level interface modules have to use through the *Framework Core*.

3.4.11. MULTIPLE INSTANCES OF THE SAME EXTENSION

At first glance it seems not effective to allow multiple instances of the same extension, as this would mean that the features that are offered by these extensions also exist multiple times within the framework. The *Framework Core* does not know which feature to choose due to the fact that the features are exactly the same. There is, however, a situation thinkable when multiple instances are useful.

Consider an application where multiple web services are used as extensions (an example is given in Appendix F, subsection F.12, A7-5). Every web service extension could hold the same functionality and it is up to the *Framework Core* to decide which one to use. This could be based on a simple load-balancing algorithm or on performance statistics based on previous requests. In any case, multiple instances are possible and even useful in some situations.

It is important that the Framework Core has functionality to deal with the fact that multiple instances of extensions and features exist, so it can choose the right instance of a feature at the right moment.

An important rule here is that these extensions may not store any business logic data with the purpose to use it in the future. If an extension exists with multiple instances and it contains business logic data it is not known which extension hold which data and this can give unpredictable results and is therefore not desirable.

4. VALIDATION

Validation is required to ensure that the objectives from section 2.2 are guaranteed. However, due to time limitations not all objectives could be validated or tested extensively. Furthermore, some objectives are hard, if not impossible, to validate.

What has been tested is the usefulness of the framework by means of a proof of concept. With this proof of concept three objectives were tested using a prototype. The objectives and the results are described in section 4.1.

The objective that the framework has to be fast and scalable has been validated. However, testing the speed of an abstract framework is impossible, due to the fact that this can only be tested on a concrete implementation of the framework, and the test will only be valid for that specific concrete framework. Therefore only the scalable part has been tested. The objective and the results are described in section 4.2.

The framework has to maximize code reusability. Although a lot of research on this subject has still to be performed, a partial solution is presented and substantiated in section 4.3.

The remaining objectives that could not be tested extensively are:

- object oriented by design (subsection 2.2.2): the detailed design provided is completely object oriented and furthermore, specified in UML. UML is a modeling language to design objected oriented systems. The objective thus far has been maintained. The detailed design can be found in subsection 3.3.3.

- completely modular built (subsection 2.2.3): this objective is harder to validate due to the fact that no concrete framework has been built, only an abstract design is provided. The design conforms to the architecture which is modular based. If the design and specifications are followed when implementing a concrete framework the objective is maintained. However, there is no control over a developer implementing a concrete framework. To conclude, only the provided design is modular and if the framework developer followed the design and specifications the concrete framework will also be modular built.

- deployable for a wide range of applications (subsection 2.2.5): the design is abstract and has no application specific specifications or modeling. Therefore the provided framework design is deployable for a wide range of applications. For a concrete

implementation of the framework this depends how the framework is implemented. Typically a concrete implementation will be deployable for a specific kind of application.

- system and platform independent (subsection 2.2.6): the framework design is an abstract design, also in the rules and specifications nothing is said about a system or platform. Furthermore, the design also takes into account that not all programming languages support multiple inheritance and provides a solution (found in subsection 3.3.1) . There is a limitation, the objective that the design has to be object oriented has as result that only object oriented programming languages can be used.

- easy to extend or modify (subsection 2.2.10): the architecture and design of the framework suggest that the framework is easily extended and modified. However, the term “easy” is relative, hence a concrete implemented framework this is not guaranteed due to the fact that there is no control of the framework developer. If the framework developer follows the design and specification it is our believe that the framework can be “easily” extended or updated.

4.1. PROOF OF CONCEPT

Recall the following objectives, from subsection 2.2.1, 2.2.9 and 2.2.4, respectively:

> ***design a new framework that is separating business logic and application logic;***
>
> ***design a new framework that is able to rapidly build applications;***
>
> ***design a new framework that is directed through a high level interface.***

These objectives are validated by a proof of concept prototype implementing a piece of one of the enterprise applications of NextSelect (see Appendix F). This prototype shows how the framework gives an advantage and that the philosophy works.

The framework philosophy and architecture demand that business logic and application logic are separated. In the tested prototype this is achieved by implementing business logic in the *Configuration* and *Employee* module. All the functionality to facilitate the modules is located in the framework extensions.

This prototype framework is simple, it offers a few high level features, like adding a new employee or modifying an existing employee. The proof of concept is tested by putting the built framework into action and although it is simple by the amount of extensions and features it showed that the offered high level interface by extensions provided means to “easy” and “fast” construction of a new application.

With this proof of concept all the objectives were achieved (to the best of our knowledge). However, the during the prototype testing attention was also paid to the three objectives stated in the beginning of this section. The first two are meant to prevent the main problems that occurred in the NextSelect case, described in subsection 2.1.2. The third objective is important due to the fact that it holds an important part of the philosophy and it gives other objectives support (like the second mentioned in this section).

The prototype framework was implemented according to the rules and specification of section 3.4. The objectives were tested with a prototype application that was built to show that the framework actually works. The prototype framework, although limited in features, showed that if the design, rules and specifications are followed the objectives are met.

4.2. SCALABILITY

Recall the following objective from subsection 2.2.7:

> ***design a new framework that is fast and scalable***

If a concrete implementation of a framework grows it is likely that the framework performance decreases due to more memory usage, more features for the Framework Core to search for etc. The framework design has to be fast and scalable; due to the fact that the term “fast” is relative and cannot be validated this part is left aside during the tests. The framework needs to be scalable for two reasons; it will grow with functionality and must not slow down exponentially when new features or extensions are added. Secondly, if the application is going to ask more of the framework (for example more concurrent users on the same application) it must not slow down exponentially. If the scalability test had failed, the framework design is useless.

The objective states that the framework design has to be scalable. This design is abstract and therefore a concrete test version of the framework is implemented for the scalability tests. While in a normal situation a concrete implementation of the framework most probably has optimizations (only load extensions that are required or intelligent code retrieval to find a feature/function) this prototype utilizes no optimizations. The implemented test framework can be seen as a ‘worst case’ scenario implementation without any optimizations while following the design and specifications. Therefore this framework is suitable for validating the scalability.

It should be noted that this validation does not state anything about scalability of an application using the framework. The scalability of a concrete implemented framework does not only depend on the design that is tested here, but also on the implemented extensions. If the implemented extensions are not scalable the framework is also not scalable.

4.2.1. TEST ENVIRONMENT AND METHODS

To test the framework design the following test parameters are used to perform scalability tests:

- Number of extensions;
- Number of features within an extension;
- Number of calls simulated to request a feature;
- Type of call: request the first, the last or a random feature of an extension.

A call is the simulation of a module requesting the *Framework Core* for a feature. The implementation used to process the call and find the feature is the same as represented in Appendix F. There are three types of calls; 1: to the first registered function, 2: to the last registered function, 3: to a random function. The prototype performance framework is dynamically constructed by a test application. This test application generates a test framework

given the test parameters, executes the test using the generated framework and retrieves the results. The test application and test framework are web-based with respect to the NextSelect case that builds web based applications. All the tests were executed on one Linux server with all non essential processes killed in order to decrease the change of another process influencing the test result data.

The scalability tests are divided into three groups; each group represents a different amount of feature requests (1.000, 10.000 and 100.000 requests). Within each group three test sets are executed representing a different amount of extensions (10, 100 and 1.000 extensions). Every test set is performed for all of the three call types (first, last and random). Two times of timings are taken into consideration for every test. The total time required to execute the requests and the amount of time required to load all extension (worst case scenario).

Every test set consists of 100 performance tests, testing different amount of functions within the extensions (from 10 till 1.000 in steps of 10). The test set is executed three times for the three call methods, resulting in 300 performance tests. Every group has three test sets resulting in 900 performance tests for each group. There are three groups in total resulting in 2.700 scalability tests executed.

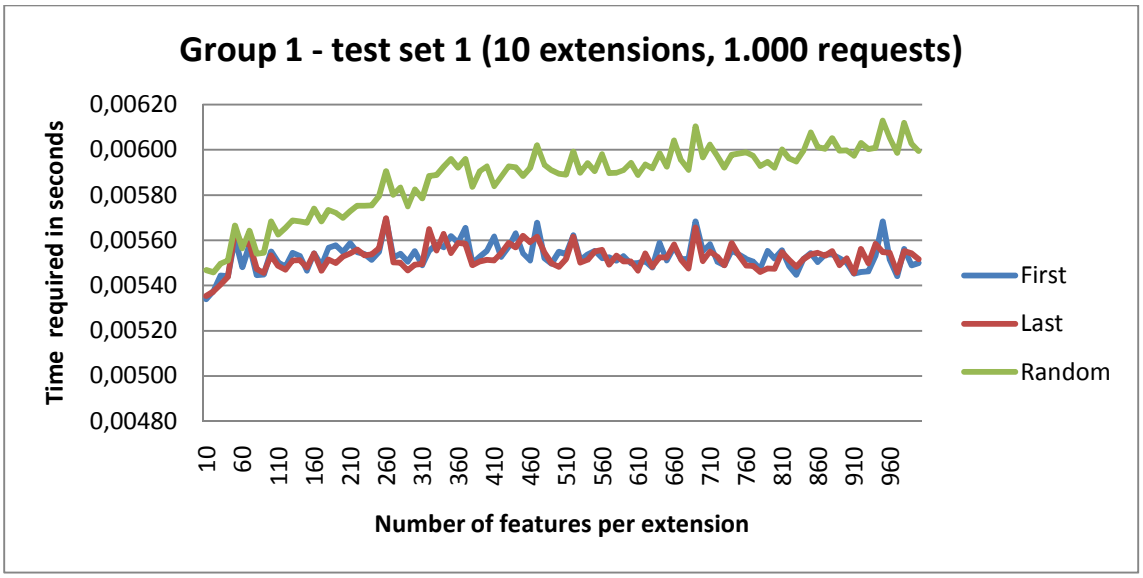
Eventually the tests sets had to be limited due to hardware limitations of the server. However, the results provide enough data for a conclusion.

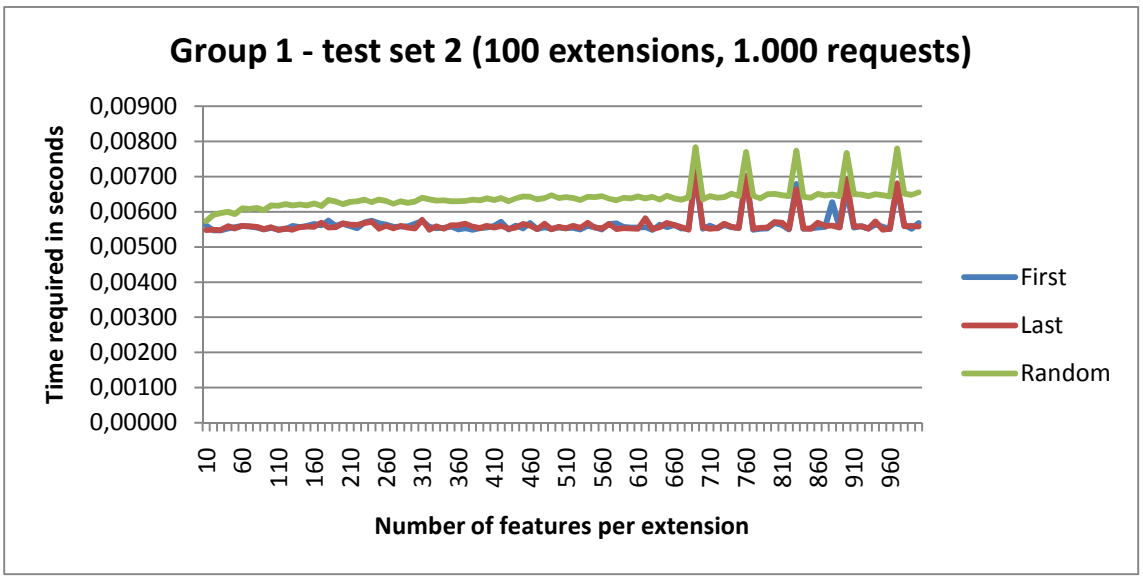
4.2.2. RESULTS GROUP 1 (1.000 REQUESTS)

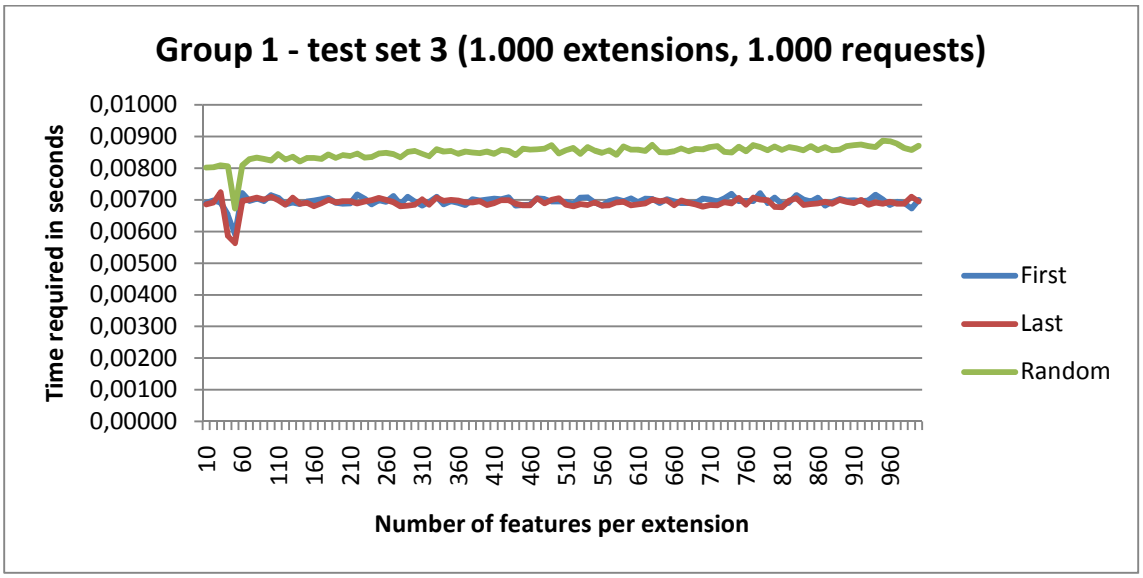
Parameters:

- Test set 1
  - Number of extensions: 10
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 1.000
- Test set 2
  - Number of extensions: 100
  - Number of features: 10 till 1000 (steps of 10)
  - Number of requests: 1.000
- Test set 3:
  - Number of extensions: 1.000
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 1.000

Results of time required to execute 1.000 requests on N extensions and M features:





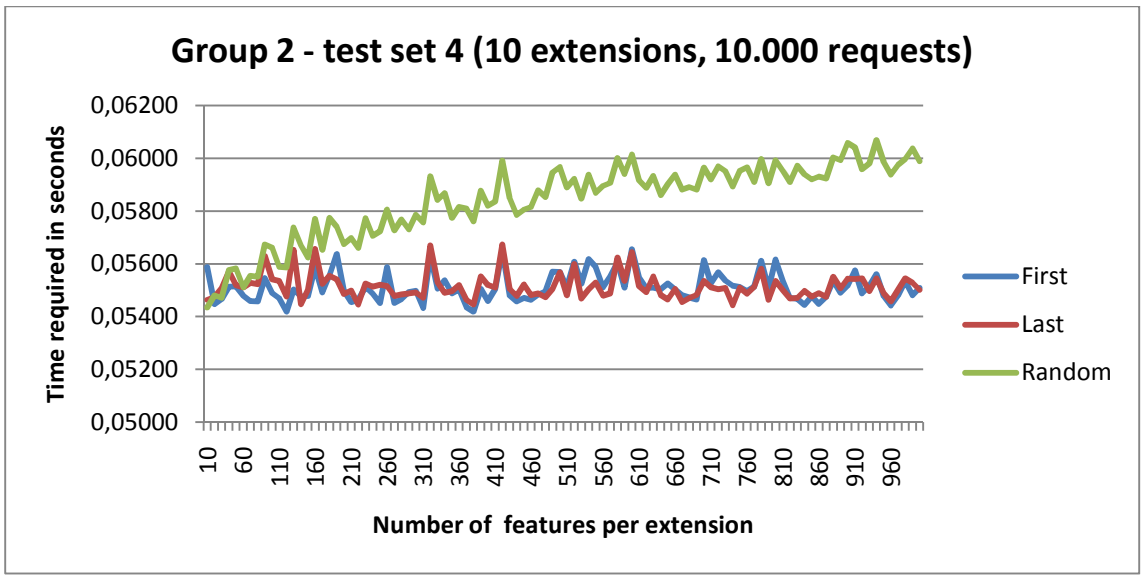


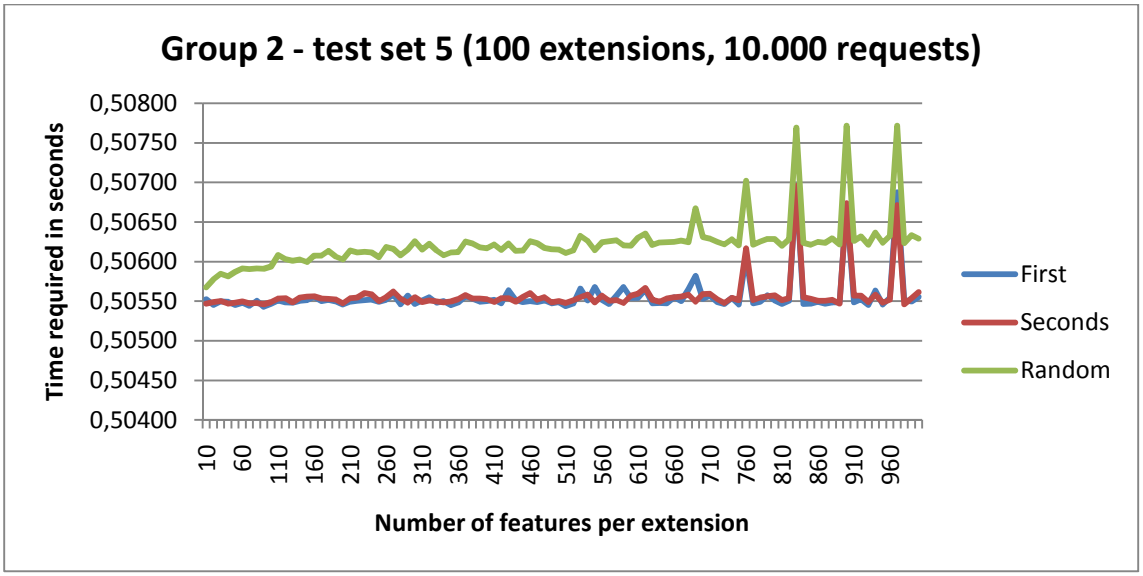
4.2.3. RESULTS GROUP 2 (10.000 REQUESTS)

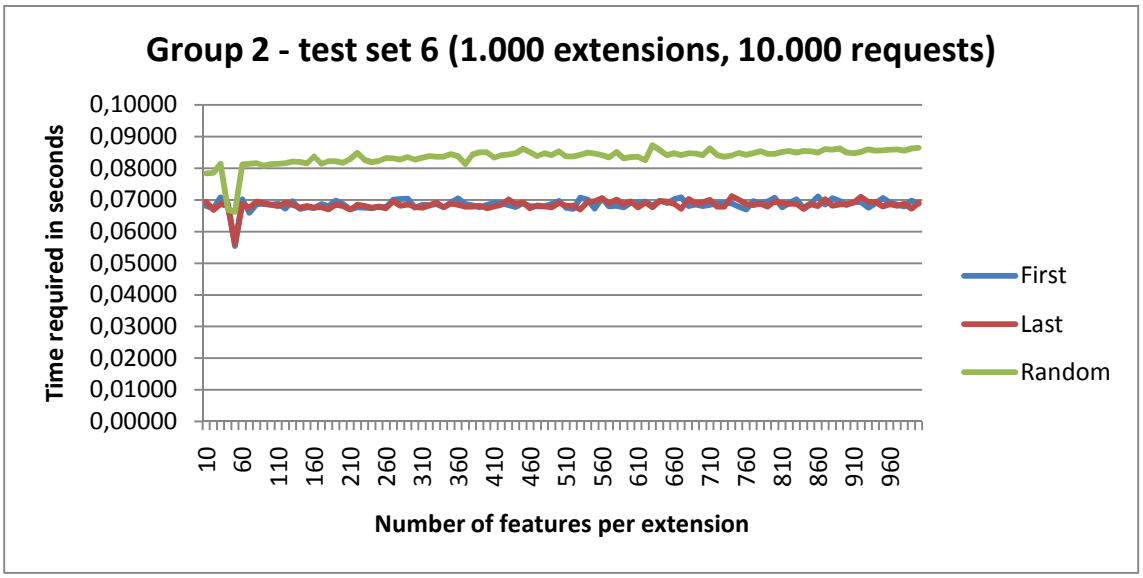
Parameters:

- Test set 4
  - Number of extensions: 10
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 10.000
- Test set 5
  - Number of extensions: 100
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 10.000
- Test set 6:
  - Number of extensions: 1.000
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 10.000

Results of time required to execute 10.000 requests on N extensions and M features:







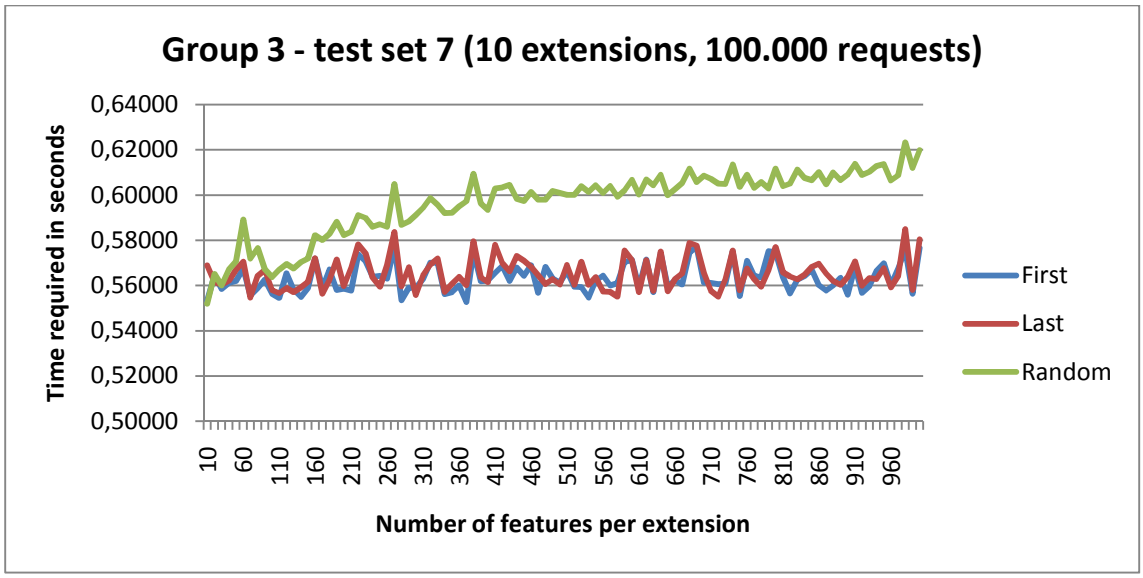
4.2.4. RESULTS GROUP 3 (100.000 REQUESTS)

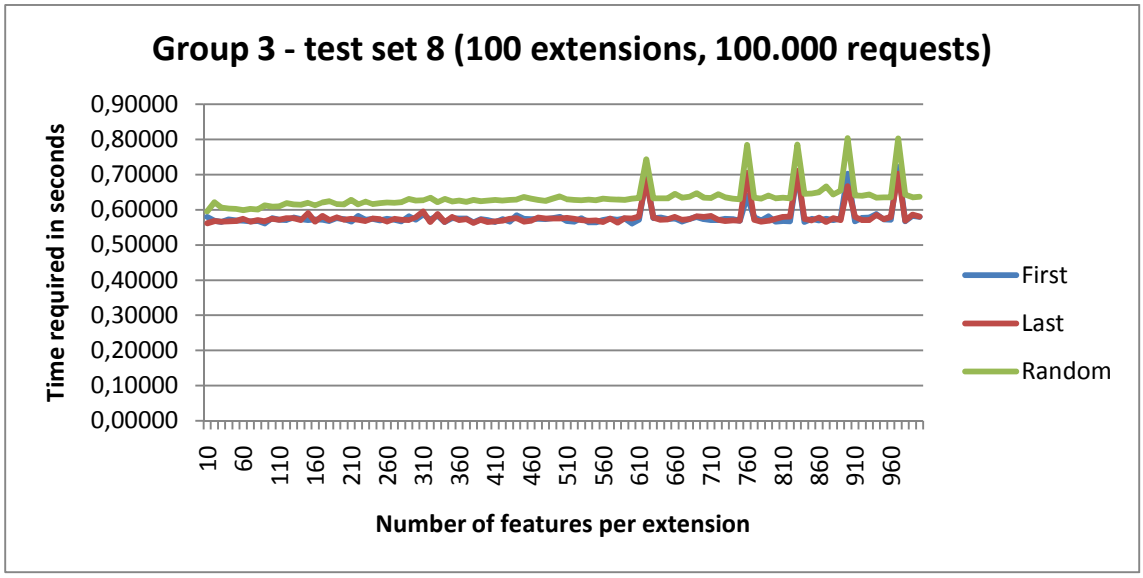
Parameters:

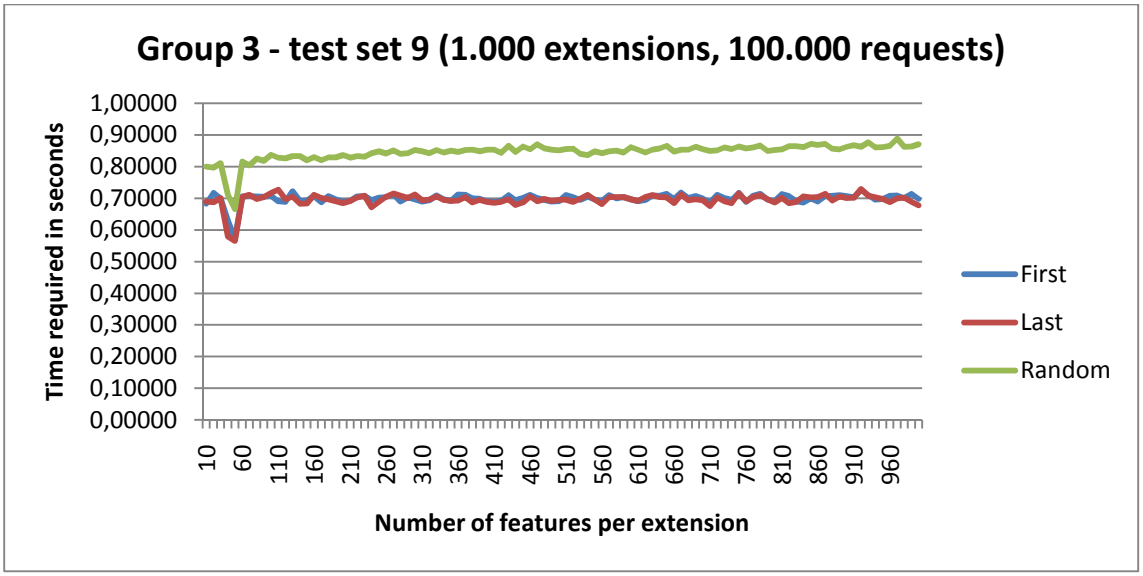
- Test set 4
  - Number of extensions: 10
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 100.000
- Test set 5
  - Number of extensions: 100
  - Number of features: 10 till 1.000 (steps of 10)
  - Number of requests: 100.000
- Test set 6:
  - Number of extensions: 1.000

- Number of features: 10 till 1.000 (steps of 10)
- Number of requests: 100.000

Results of time required to execute 100.000 requests on N extensions and M features:

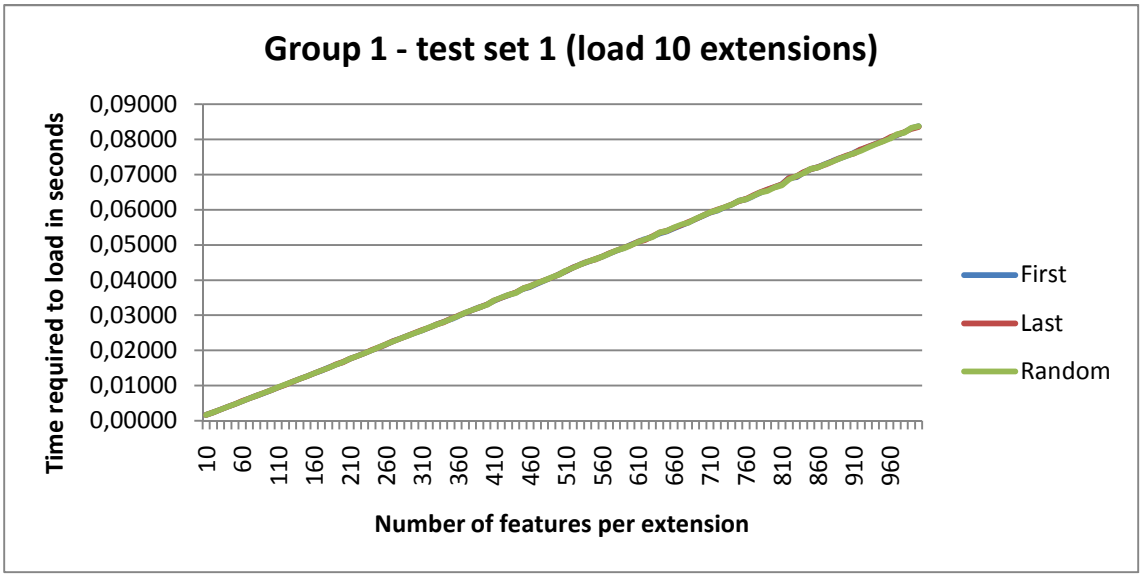


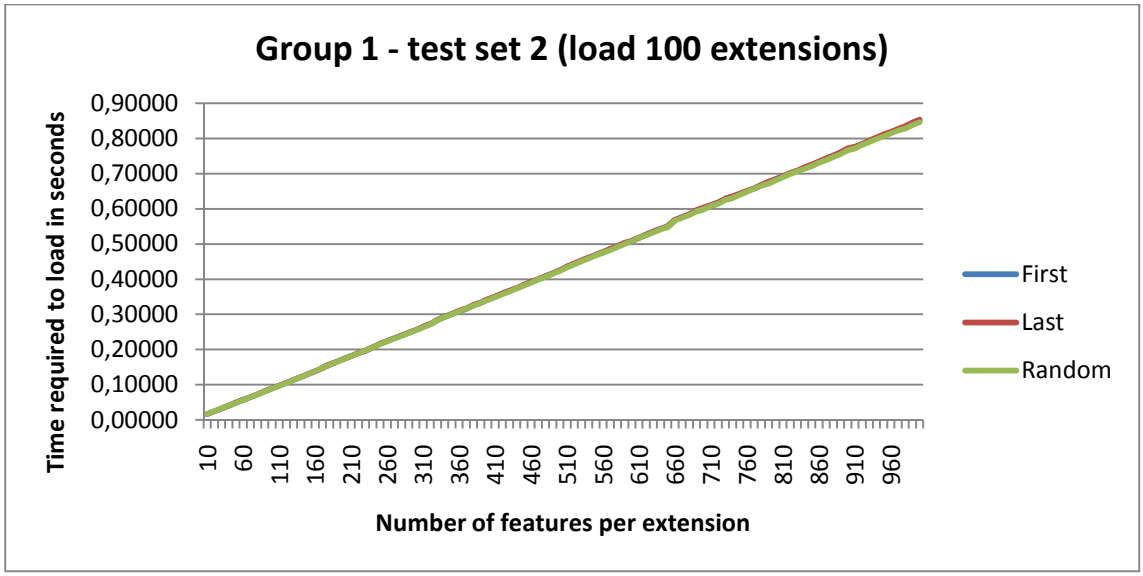


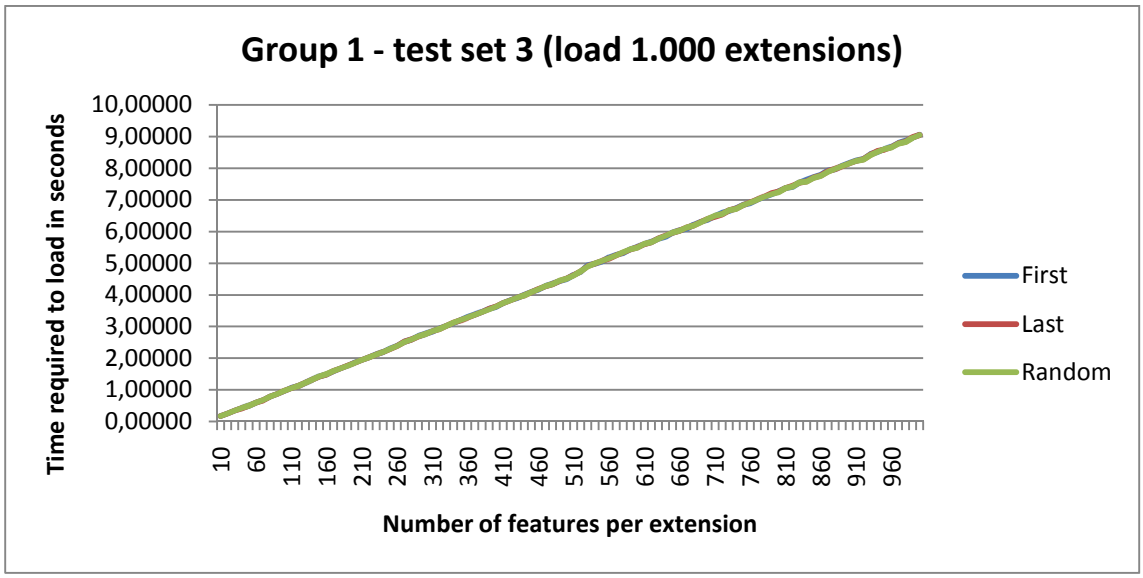


4.2.5. TIMING RESULTS LOADING EXTENSIONS

As stated also the time required to load N extensions holding M features had been tested. Due to the fact that the results of every set group are (more or less) the same, only the results of the first group are used. In the following diagrams the –not so shocking- results are presented:







4.2.6. DISCUSSION

The results are promising! The time required to sent requests for features does not grow exponentially when the number of extensions and features grow. With some fluctuations and some unexplained readings that repeatedly occurred with an amount of extensions, the time required is linear. The time required to load extensions is clearly linear in all cases. Ten times the amount of extensions takes 10 times the amount of time to load them. Although it would be better to perform the scalability test on multiple platforms it is reasonable to assume that the results will not drastically change.

Keep in mind that no optimizations were used and therefore these test can be seen as a worst case scenario for the framework design to test the scalability. A concrete implementation of the framework will produce different results.; but if the abstract design is not even scalable the

framework design would be rendered useless due to the fact that is that a concrete framework will grow with time.

4.3. MAXIMIZING CODE REUSABILITY

Recall the following objective from subsection 2.2.8:

> ***design a new framework that is maximizing code reusability***

For this objective no hard tests have been (or could have been) performed. In this section a possible solution is presented how the abstract framework is able to help maximizing code reusability. In subsection 3.4.7 the matter was shortly addressed and pointed out that 100% maximization of code reusability (e.g. 0% code duplication) is a honorable yet impossible task. In general, studies have shown that about 20% of the total code in large software systems is duplicated [36, 37]. The abstract design and specifications of the framework provide two aspects that will help the framework developer to maximize code reusability.

The first aspect originates from the objectives (section 2.2) and philosophy (section 3.1) and is supported in the abstract design of the framework. Business logic and application logic have to be completely separated (the difference between the two can be found in subsection 2.2.1). Modules contain the business logic translated into some form of scripting language. Extensions have the responsibility to facilitate the framework and modules with high level features and reusable functionality. It is not allowed to implement functionality within a module (subsection 3.4.6), hence the application developer is forced to reuse code offered by the framework extensions.

The second aspect, addressed in subsection 3.4.7, is a concept that is based on basic variable labeling and code retrieval. During the research it became clear that for code reusability a kind of code retrieval system is required. Due to the fact that the main research was not about code retrieval but to design the abstract framework this marked as future work. However, the basic variable concept will provide means to make the code retrieval easier.

Normally data is represented as integers, doubles, booleans , strings etc. The concept states that all of these basic variables have to put in a special extension data type class. This class holds the variable, and with methods like operating overloading it is possible to let this class act as if it were a normal variable. Example, instead of storing an age of a person in an integer it is stored in an extension data type class called “Age”. This has to be done for all basic variables which means that, if executed correctly, all of the application logic consists of object classes and no basic variables (except for the actual value storage within an extension data type).

The extension data type gives an advantage for a possible code retrieval system. With this concept one can only search for existing code using keywords but also find functions by looking at the input or output parameters. A possible code retrieval system now has the ability to search more accurately for certain functionality by utilizing the extension data type.

Why does the concept of the extension data type class have potential? The reason for this is that in a way extra information is added to the variable, it is labeled. With this extra information a more precise search can be performed. For example: “give a list of functions that accept an *BirthDate* class as input and return an *Age* class” instead of “give a list of functions that accept an date as input and return a integer as output. This concept is also supported by Date et al. [35] and Codd [38-40].

5. CONCLUSIONS AND RECOMMENDATIONS

This research project started with the goal to design a framework that would solve all the problems of NextSelect, with the purpose to start building new software for MASER Engineering and at the same time also solve common problems in the software development world, especially for enterprise application development. Due to the latter the goal changes to broaden the research and not focus only on the NextSelect case. When the research project started it became clear that the objectives stated in section 2.2 would solve almost all the software development issues in the world. Presenting a complete and working solution that solves all the problems of NextSelect, thus also for the world, became a naïve vision. It can be compared to finding the holy grail for the software engineering. As a result boundaries had to be defined that were focused on the NextSelect issues described in section 2.1 and at the same time try to also take in to account the more general problems that are assumed to exist in general.

This thesis provides the first steps in building an framework that maintains the objectives stated in section2.2 and prevent the problems described in section 2.1.

5.1. THE RESULTS

What has been realized is an basic abstract design of a framework (see subsection 3.3.3), accompanied with a set of rules and specifications (see section 3.4). The design and specifications of the framework are driven by a philosophy (see section 3.2) which makes this framework unique. The design is abstract and can be used for a wide range of applications; however, due to the abstractness a concrete version has to be implemented for a specific kind of application. If a framework is implemented following the design, rules and specifications, it is for the most part guaranteed that the objectives of section 2.2 are achieved and the general problems described in subsection 2.1.2 are avoided. There are two development phases: development of the framework and development of the application. The framework has to bedeveloped using any kind of object oriented programming language. The application is developed using a high level interface that the framework has to provide.

5.1.1. FRAMEWORK DEVELOPMENT

Implementing a new framework following the design and specifications starts with identifying for what kind of applications it has to be deployed for. If it will only be deployed for one application this is not a good solution to use, the design and specifications are based on the fact that a concrete framework will (eventually) serve more than one application. The framework is divided into two sections: The *Framework Core* and the extensions (see section 3.2). The *Framework Core* acts as a façade which offers the high level interface to the outside world; it processes incoming high level requests and has the responsibility to deliver the requests to the corresponding extensions. How to implement the Framework Core can be found in subsection 3.4.5. Extensions contain all the functionality to provide features that have a high level interface. Extension may not contain any application specific code, they must be configurable for every application the framework is deployed for. How to implement extensions and for the rules described in subsection 3.4.7

5.1.2. APPLICATION DEVELOPMENT

Business logic is what defines the application, it has to be translated into the high level scripting interface offered by the developed framework and its extensions. The business logic is placed in modules; every module contains a part of the application, it depends on the application design how to divide it into modules. Modules are application specific, therefore every application has its own set of modules. How to implement modules can be found in subsection 3.4.6.

5.2. HYPOTHESIS AND RESEARCH QUESTIONS

5.2.1. HYPOTHESIS

> ***Does the framework design, sketched in Figure 1.1, offer a solution for the problems described in section 2.1 ?***

The hypothesis cannot answered simply by a 'yes' or a 'no'. The fact that the design is abstract means that the presented solution cannot yet be tested for NextSelect or any other software development company in that matter without a concrete version. However, the hypothesis states the question if 'a' solution is offered, not a total solution. Obviously, the total solution would solve all the software development issues in the world. With this in mind some issues and objectives have been tested using prototypes for validation. Due to time limitations not all aspects could be covered and are therefore marked as future work (see section 5.4).

The results thus far look promising (see chapter 4), most of the objectives are validated and even tested with prototypes. Based on this the hypothesis appears to offer a solution of the problems described in section 2.1. Nevertheless, without applying (see section 5.3) a concrete version of the framework it remains uncertain.

5.2.2. RESEARCH QUESTION Q1

> ***Is it possible to apply the framework design to other platforms, programming languages and non web-based applications?***

The answer is yes!

The architecture, design and specification state no demands on the programming language and or platform. With one exception, due to the objective that the framework design needs to be object oriented (see subsection 2.2.2) the design can only be used in object oriented programming languages. Furthermore, the design is abstract which means that it holds no platform, programming language or application specifications.

The specific design of the framework which represents the philosophy (described in section 3.1) could cause problems with multiple inheritance, whereas not all programming languages support this feature. For this a solution is presented with the purpose to make the design applicable for more object oriented programming languages like the popular Java. This solution is presented in subsection 3.3.1.

5.2.3. RESEARCH QUESTION Q2

> ***The CORBA and Web Service architectures show resemblance to the current framework architecture. Is the framework concept - or parts of - the same as the architecture of CORBA and/or web services?***

The answer is no!

The framework, CORBA and Web Services show resemblances by design and the fact that they are specification. However, they operate at a different level and they have different goals, they are therefore not the same. The framework is for building applications, CORBA and Web Services facilitate communication between applications.

Looking at the functionality that the *Framework Core* should provide compared to CORBA and Web Services, some components appear to have resemblances. For the most part the ORB, the CORBA broker. The idea behind the *Framework Core* is that an incoming request from a module gets processed and sent to the right extension that will handle the request. The *Framework Core* could be very complex, e.g. an concrete instance of the framework might search for the right extension and if multiple extensions exist that could process the request the *Framework Core* has to sort out which one to use. This is typical service broker behavior. CORBA and Web Services both have service broker. In CORBA the actual ORB is the broker and with Web Services the broker can be found in the UDDI registries. For more information and comparison see Appendix E.

5.2.4. RESEARCH QUESTION Q3

> ***How are modules and extensions implemented and how are they going to communicate?***

Modules are implemented by application developers and hold the business logic of a customer. The business logic needs to be translated into some sort of scripting language holding the customer parameters for the application (the framework programming language is allowed to be used). Details on how the module have to be implemented can be found in subsection 3.4.6.

Extensions are implemented by framework developers and have the responsibility to facilitate the framework and modules with one or more features and contain all the reusable functionality of the application. Modules feed the extensions with parameters how to act. Due to the fact that this has to be done with simple calls the extensions have to be implemented offering some form of high level interface. Underwater they can be as complex as possible as long as the interface for the modules is plain and simple. Details on how extensions have to be implemented can be found in subsection 3.4.7.

The communication between modules and extensions depends on the concrete implementation of the framework. Modules have to communicate though the *Framework Core* for requesting functionality, this can be done in the same programming language or in a more advanced (scripting) language that the *Framework Core* is able to interpret by itself or by using interpret extensions (see subsection 3.4.6). Extensions are allowed to communicate directly to each other by means of direct class method calls instead of using the high level interface that modules have to use through the *Framework Core*. For more details see subsection 3.4.10.

5.2.5. RESEARCH QUESTION Q4

> ***What is the usefulness of the Framework Core and why is it required?***

Framework Core acts as a façade which offers the high level interface to the outside world; it processes incoming high level feature requests from the modules and has the responsibility to deliver the requests to the corresponding extensions. Without the framework core modules have to communicate directly with extensions, hence demolishing the philosophy of the framework. With the Framework Core acting as a façade it is easier for the framework developers to keep the framework backwards compatible with old modules, as a result the application developers do not necessarily have to rebuild their modules if an structural change is carried through the framework's set of extensions. More information about this matter can be found in Appendix F, section 3.4 (in general), subsection 3.4.5 and section 5.1.

5.3. HOW CAN THIS BE APPLIED AT NEXTSELECT

NextSelect and MASER Engineering collaborated for this research project with the goal to find a solution for the problems stated in section 2.1. As stated in subsection 5.2.1, the hypothesis of the research cannot be answered with a simple 'yes' or 'no', primarily due to the fact that the design of the framework is abstract. To test if it really solves the NextSelect problems, a concrete version has to be created that is able to facilitate the applications of NextSelect. It goes without saying that implementing a framework is going to take quite an amount of man hours. No software development company will do this without some proof that the concept and philosophy of the framework works.

This thesis provides the first steps in building an framework that achieves the objectives stated in section2.2 and prevens the problems described in section 2.1. With the offered and partly validated results, NextSelect feels that the results are promising enough to build a concrete version of the framework taking into account the future work (see section 5.4).

Developing the new framework is a complex operation and will take a lot of time. The existing frameworks and applications have to be thoroughly studied to make a concrete design of a

framework that is able to facilitate all the applications for the different customers of NextSelect. Deploying the framework, hence replacing the old application and old framework (e.g. MIDS, illustrated in Figure 2.3) with new implemented modules containing the MIDS business logic is the first step. The implemented concrete framework can then be real life tested with the MIDS application. If it is successful the framework can be deployed to the other applications.

5.4. RECOMMENDATIONS AND FUTURE WORK

If there is a concern about the presented abstract design and specifications of the framework, it is that it is missing tangibility. This is caused by the abstractness of the framework, and although some aspects have been tested, much like a proof of concept, scalability and maximizing code reusability, it is missing a concrete implementation and real proof that the framework will work for NextSelect or any other software development company. The problem here is validation, the objectives and produced design cannot be validated formally which would give some hard evidence that the concept works.

Furthermore, the concept of the framework has to solve general problems in the software development world, focusing on enterprise applications. This has been taken into account for the research, however, the research for the most part has been focused on the web based applications that NextSelect produces. The problem analysis , that resulted in the objectives and philosophy of the framework, were based on the NextSelect case. This is not strange considering that NextSelect in cooperation with MASER Engineering provided the research project.

5.4.1. VALIDATION OF THE RESULTS

Due to the fact that it is assumed that the problems of the NextSelect case are general for most software development companies the validation in this research has been focusing for the most part on the NextSelect case. In this research also time limitations prevented that the produced resulted could all be validated and some constraints have been made. It is therefore recommended that the framework design and specifications are all validated without looking at the NextSelect case.

5.4.2. FRAMEWORK RESULTS ARE FOCUSED ON WEB-BASED APPLICATIONS

Also partly mentioned in the previous subsection 5.4.1, the results and validation are for the most part based on web based applications. Although this was known in the beginning of the research it is also recommended that other type of application are taken into consideration. Especially non web based applications.

5.4.3. IMPLEMENTING MODULES

Beside the specifications provided in subsection 3.4.6 it is still not completely clear what is allowed to implement in a modules and what is not. The specification states that no new functionality may be implemented, however the use of the programming languages (used for developing a concrete framework) may be used. When is a specific part of code new functionality or just a script telling the framework what to do? The specification is not clear about this. For example, is the calculation of the total sum of a price plus reduction new functionality (totalPrice = price + reduction)? This question remains unanswered and requires further research.

The philosophy states that some sort of scripting has to be used to represent the business logic of the customer and tell the framework what to do. The programming language used for building the framework may be used. However, for the philosophy it is better that the framework provides a scripting language for the modules to use, because application developers have no choice by follow the scripting language provided. With this approach the previously described issue disappears due to the fact the provided scripting language restricts the application developer. In this thesis no research has been performed about an actual scripting language. Again, this is because the framework is abstract. Still, it is a recommendation that research is performed on this subject to provide, for example, a general scripting interface for these kind of frameworks.

5.4.4. MODEL, VIEW AND CONTROLLER

This research project started with a framework that also supported MVC, Appendix A, Appendix B and Appendix C. However, during the research it became clear that taking MVC into the basic design of the framework would make the framework less abstract. Furthermore, by adding MVC to the basic design in effect added application specific design to the framework. This would limit the wide range of applications the framework could be deployed for, this is in contrast with the objective in subsection 2.2.5.

Nevertheless, when extensions are implemented with reusable functionality MVC has to be taken into account as means to produce clean software. For the abstract design rules and specifications could be added to ensure MVC is correctly implemented.

In Appendix section F.8 a possible solution is presented for web based application using templates. This is limited to web based applications, therefore this issue requires more research.

6. REFERENCES

[1] D. Riehle, *Framework Design – A Role Modeling Approach*, a dissertation submitted to the Swiss Federal Institute Of Technology Zurich, 2000

[2] Wikipedia , *Framework - Software framework*, Online documentation, viewed February 2009, http://en.wikipedia.org/wiki/Software_framework.

[3] Sam Ruby, Dave Thomas, David Heinemeier Hansson. *Agile Web Development with Rails.* Book, third edition, ISBN-13: 978-1-9343561-6-6.

[4] Elliot Smith, Rob Nichols. *Ruby on Rails Enterprise Application Development*. Book, ISBN-10: 1847190855, ISBN-13: 978-1847190857.

[5] Brian Getting. *Software Frameworks*. Online article, published on October 24, 2007, http://www.practicalecommerce.com/articles/593/Software-Frameworks/.

[6] Tony Sintes. *Abstract classes vs. Interfaces*. Online article, published March 20, 2001, http://www.javaworld.com/javaworld/javaqa/2001-04/03-qa-0420-abstract.html.

[7] Matthias Book, Volker Gruhn. *A Dialog Flow Notation For Web-Based Applications*. Chair of Applied Telematics/e-Business,* Department of Computer Science, University of Leipzig. Unknown publish date.

[8] Anurag Bhatt, Pratul Varshney, Kalyanmoy Deb. *In Search of No-loss Strategies for the Game of Tic-Tac-Toe using a Customized Genetic Algorithm*. Indian Institute of Technology Kanpur. Unknown publish date.

[9] Yanpei Chen, Patricia C. Fong, Jerry Hong, Deepa Mahajan, Cynthia Okita, David Eitan Poll, Alan Roytman, Ofer Sadgat, Daniel D. Garcia. *200 Students Can't Be Wrong! GamesCrafters, a Computational Game Theory Undergraduate Research and Development Group*. University of California, Berkeley. Unknown publish date.

[10] Edwin Soedarmadji. *Decentralized Decision Making in the Game of Tic-tac-toe*. IEEE, December 18, 2005.

[11] Jırı Grim, Petr Somol, Pavel Pudil. *Probabilistic Neural Network Playing a Simple Game*. Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic. Unknown publish date.

[12] Kevin Crowley, Robert S. Siegler. *Flexible strategy use in young children's tic-tac-toe*. Carnegie Mellon University, Cognitive Science 17, 531-561, 1993.

[13] Ewan Tempero and Robbert Biddle*. Simulating Multiple Inheritance In Java*. Victoria University of Wellington, Technical Report CS-TR-98/1, May 22, 1998.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Book, Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[15] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* A dissertation submitted to the faculty of The University of Utah in partial fulfillment of the requirements for the degree of Doctor of Philosophy. Department of Computer Science, The University of Utah, March 1992.

[16] Robert Biddle and Ewan Tempero. *Understanding the impact of language features on reusability*. Victoria University of Wellington, Technical Report CS-TR-95/17, September 4, 1995 (Revised January 12, 1996).

[17] *Common Object Request Broker Architecture: Core Specification*, March 2004, Version 3.03, OMG Document Number: formal/04-03-12

[18] *Common Object Request Broker Architecture (CORBA) Specification - Part 1: CORBA Interfaces*, Version 3.1 , OMG Document Number: formal/2008-01-04

[19] Springer New York. *Reliable Distributed Systems*, ISBN 978-0-387-21509-9, chapter 6: CORBA: The Common Object Request Broker Architecture, p.119-140. Unknown publish date.

[20] CyberObject, *Common Object Request Broker Architecture*, Online article, 1997, http://cyberobject.com/co/whitepaper/CORBA.PDF.

[21] Winston Lo, Yue-Shan Chang, Shyan-Ming Yuan and Deron Liang. *The Design and Implementation of a Multi-Threaded Object Request Broker*, Journal of Information Science and Engineering 16, 365-379 (2000).

[22] Gustavo Alonso, Fabio Casati. *Web Services and Service-Oriented Architectures.* Proceedings of the 21st International Conference on Data Engineering (ICDE 2005) 1084-4627/05.

[23] K. Mockford. *Web Services architecture*. BT Technology Journal, Vol. 22 No.1, January 2004.

[24] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard. *Web Services Architecture*. W3C Working Group Note 11, February 2004.

[25] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 - Part 1: Core Language*. W3C Recommendation, 26 June, 2007.

[26] Martin Gudgin , Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation, 27 April, 2007.

[27] Michael Niblett. *Distributed feature extraction: A web service approach.* University of Twente, Faculty of Electrical Engineer-ing, Mathematics and Computer Science, 2007.

[28] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers. *UDDI Version 3.0.2*. UDDI Spec Technical Committee Draft, Oktober 19, 2004.

[29] Aniruddha Gokhale, Bharat Kumar, Arnaud Sahuguet. *Reinventing the Wheel? CORBA vs. Web Services*. WWW2002 Program

[30] Propal, *Propel is an Object-Relational Mapping (ORM) framework for PHP5*, Online documentation, viewed 2009, http://propel.phpdb.org.

[31] Wikipedia, *Uniform Resource Locator*. Online documentation, viewed 2009, http://en.wikipedia.org/wiki/Uniform_Resource_Locator.

[32] PHP documentation, *Classes and Objects – Overloading*, Online documentation, viewed July 17, 2009, http://us2.php.net/manual/en/language.oop5.overloading.php.

[33] PHP documentation, *Classes and Objects – Magic Methods*. Online documentation, viewed July 17, 2009, http://us2.php.net/manual/en/language.oop5.magic.php.

[34] PHP documentation, *Classes and Objects – get_class_methods*, Online documentation, viewed July 17, 2009, http://us2.php.net/manual/en/function.get-class-methods.php.

[35] C.J. Date, Hugh Darwen. *Foundation for Object / Relational Databases*. Book, The Third Manifesto, ISBN 0-201-30978-5.

[36] Gilad Mishne. *Source Code Retrieval using Conceptual Graphs*. Master of Logic Thesis, Institute for Logic, Language and Computation University of Amsterdam, December, 2003.

[37] B.S. Baker. *On Finding Duplication and Near-Duplication in Large Software Systems*. In L. Wills, P. Newcomb, and E. Chikofsky, editors, Second Working Conference on Reverse Engineering, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[38] E.F. Codd. *Derivability redundancy consistency of relations stored in large data banks*. Research Division San Jose, California, August 19, 1969.

[39] E.F. Codd. *The relational model for database management: version 2*. Book, ISBN 0-201-14192-2, Addison-Wesley, 1990.

[40] E.F. Codd. *Relational completeness of database sublanguages*. IBM Research Laboratory San Jose, California, March 6, 1972, RJ 987 (#17041).

6.1. OTHER CONSULTED RESOURCES

The following resources have been consulted during the research, but are not used in this final report.

[41] Narayanan A.R. *Aspect- vs. Object-Oriented Programming: Which Technique, When?* Online article, http://www.devx.com/Java/Article/28422

[42] Graham O'Regan. *Introduction to Aspect-Oriented Programming*. Online article, http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html

[43] Improving-extensibility-of-object-oriented-frameworks-with-aspect-oriented-programming_2006_Lecture-Notes-in-Computer-Science-(including-subseries-Lecture-Notes-in-Artificial-Intelligence-and-Lect

[44] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. *Aspect-Oriented Programming*. Proc. of`ECOOP'97, Finland, Springer-Verlag LNCS 1241, June, 1997.

APPENDIX A. HOW TO IMPLEMENT MVC

One of the main concerns is to separate the application logic, model and user interface components. The design pattern 'Model View Controller' (MVC) plays an important role when designing an application. This prototype was completely built in PHP 5, a web-based programming language using the object oriented structure. The choice for PHP 5 was made due to the fact that using MVC within an web-based application was unknown territory, furthermore the design also has to be applicable for web-based systems.

This limited prototype framework should be able to calculate an age given a birth date. Because it is a web-based application some sort of HTML form also has to be generated. A module is required that tells the framework 'what to do' and thus containing the business logic. The framework itself contains all the tools and methods to facilitate the needs of the module. E.g. depending on the current action of the user the module will tell the framework if it needs the web form or the age calculation etc.

Please note that this structure of messaging between framework, modules and extensions is very basic, limited and specifically designed for this prototype. With this framework it is not at all possible to build advanced web-based applications. Another way of looking to this specific framework is to see it as a framework just for web-based age calculation.

A.1. RESEARCH QUESTIONS

Before and during the development of this prototype the following questions arose:

QA-1: With this design it is now clear how to implement MVC in a web-based application?

QA-2: Although the MVC components are put in the secondary design, the question is if it is possible to design general secondary components that are still deployable for a slightly limited range of applications?

QA-3: Basic functionality of the framework can be seen as separate extensions of the framework, the question now is: can a general interface be designed for all extensions and if so, what should the general interface contain?

QA-4: Is it possible to apply the design to other platforms, programming languages and non web-based application?

QA-5: The final framework design (first level) has to be an abstract design, with no concrete implementations. Can the abstract framework be seen as a new pattern, i.e. a framework pattern?

QA-6: What about threads? If the application has multiple threads, is the framework able to manage the threads and is it thread safe ?

A.2. DESIGNING THE PROTOTYPE

The design of the framework is based on the original architecture discussed in section 3.2. UML is used as a basis. In the original architecture the framework self consists of a framework core and extensions. By adding modules the whole forms the application. If this approach is translated into UML the following abstract UML diagram can be derived:

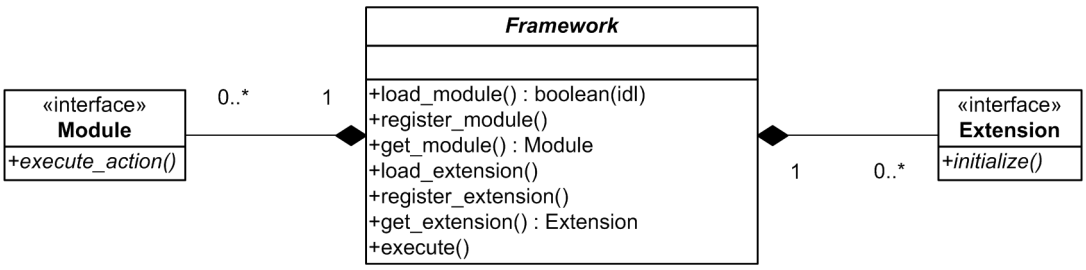


Figure appendix A-1: Framework design presented in a UML diagram

In this UML diagram there are three classes; *Module*, *Framework* and *Extension*. There is one framework class that knows zero or more modules and zero or more extensions. The *Module* and *Extension* classes are interfaces, they do not have any concrete implementations. In this prototype the *Framework* class only has some basic support to register modules and extensions.

Class *Framework*:

- **load_module/extension():** locates the module or extension and tries to open it via the PHP 'include' statement.
- **register_module/extension():** if the module or extension is successfully loaded this function adds the module or extension to the frameworks internal register (in this case an Array).
- **get_module/extension():** retrieves a module or extension by a given name.
- **execute():** this will call and execute a function in a Extension given a extension and function name. With the given extension name this function searches the internal register of the Framework to see if the extension is loaded en registered, if so it tries to execute the function within this extension. This is a very basic approach due to the fact the programmer of the module exactly has to know the extension and the function name within the extension. The ideal outcome is that de module requests a service without knowing which extensions and functions exactly are available. For this prototype the simple and basic approach will suffice.

Class *Module*:

- **execute_action():** executes the module with a given action. Depending on the action itself the module 'knows' what to do and to ask of the framework. In this prototype the function is called from an extension, this is discussed later on. Because this a interface class this function has to be implemented by a module itself.

Class *Extension*:

- **initialize():** this initializes the extension and gives it the possibility to perform required actions before the extension can be used, e.g. load other extensions or check dependencies etc. This function must be called when all other extensions are loaded because the initialization of one extension can depend on the existence of another extension that has to be loaded into the framework. Because this a interface class this function has to be implemented by an extension itself.

What about MVC?

At this point the design is abstract and is unable to form an application. For this more 'concreteness' is required; the functions within the Framework class have to be implemented and some concrete modules and extensions have to be realized. The most important issue in this

prototype is the implementation of MVC. Because MVC within a web-based application is unknown territory, designs of existing web-based frameworks (php.MVC, Drupal and Ruby on Rails [3]) were used as a reference.

During the research on MVC in other web-based systems it became clear that every component of MVC (The Model, View and Controller) can be seen as a separate parts of the system. In this way it is possible to implement for example different Views to represent a Model, one of the major advantages of the MVC pattern. It is clear that MVC is the responsibility of the framework and not of the module, because the module only tells the framework what to do. Although it is suggested that the module itself is somewhat a controller by itself. This concept will be handled later one and is not important for this prototype.

So where are these MVC components located in the framework? One option is to make MVC part of the framework itself, so it is built or 'fused' in the design of the framework. This sounds logical because one of the main goals is the design of a framework that separates business logic, model and user interface. But during the design of this prototype problems emerged; it seemed impossible to make the design in such a way that another important objective, deployable for a wide range of applications, could be preserved. If major design choices, such as MVC, are made before the framework itself is deployable for an application, it could limit the amount of applications this framework could build. I.e. if the MVC design is fixed within this framework it might be possible that the framework is useless for another type of application that requires a different approach. Because this is undesirable another approach is suggested.

This new approach states the following: **every component of the framework can be seen as an extension (with the exception of the framework core)**. Following this line of reasoning this would mean that the Model, View and Controller are in fact extensions on their own. By applying this concept the initial idea of the framework changes, where MVC should be part of the framework it now becomes an external part and the separation of business logic, model and user interface cannot be guaranteed. Maintaining this approach would require a set of guidelines added to the design of the framework, i.e. telling the programmer, who is implementing the framework, how MVC has to be implemented to maintain the separation if business logic, model and user interface. Another option is to split the design up into two levels; the first level is the core basic of the framework (see figure 7-1) and the second level design would consist of extra concrete or abstract extensions that provide important aspects to the framework, i.e. MVC. Because the second level design could limit the deployment for different kind of applications this would mean that in certain circumstances a (slightly) different second level design is required.

Applying second level design

The second approach (with the second level design) seemed logical to use because it guaranties the research objectives, whereas the first approach does not. Therefore the second level design

principle has been applied in this prototype. This means that the Model, View and Controller are implemented as extensions. In the following UML diagram the second level design is illustrated:

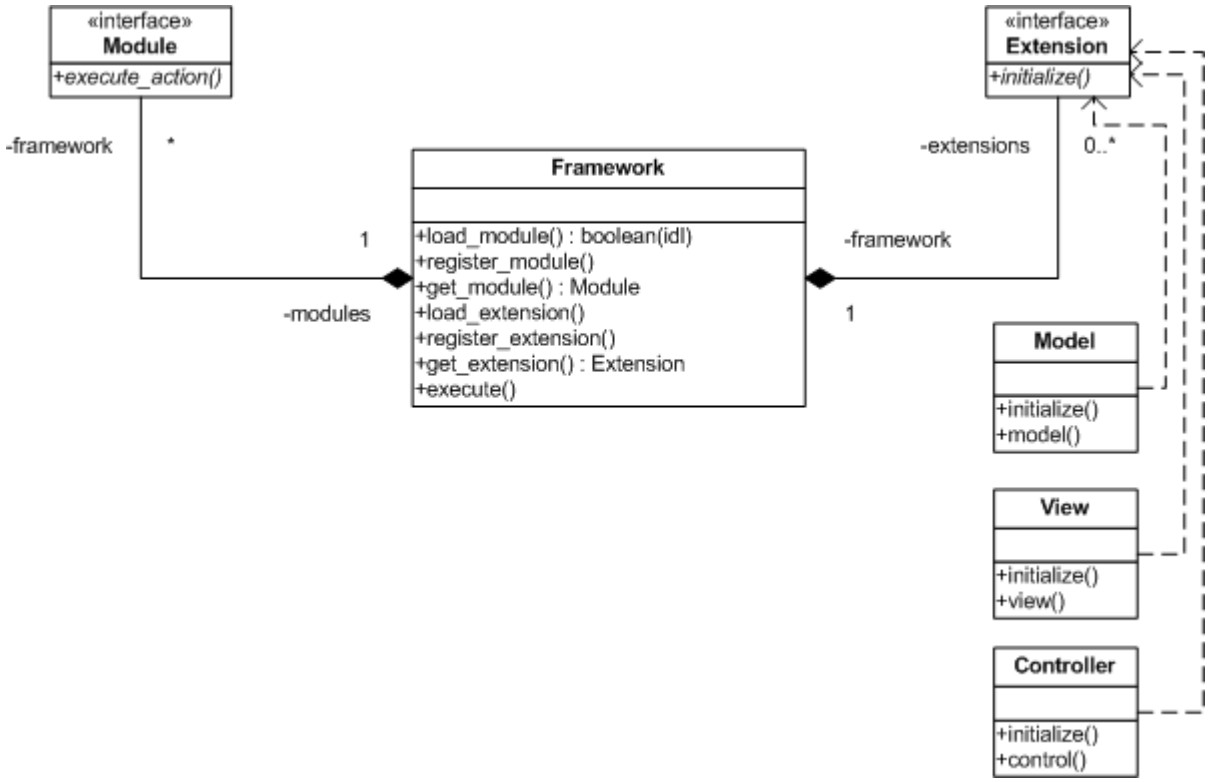


Figure appendix A-2: Framework UML diagram with second level design.

Although this UML diagram is different compared to the first discussed abstract UML diagram (see **Error! Reference source not found.**) it is still recognizable (*Module*, *Framework* and *Extension* classes). On the right side the discussed MVC classes (*Model*, *View*, *Controller*) are added as extensions. To make a class an extension these classes have to implement the *Extension* interface class, this means that the function *initialize()* has to be implemented in every extension class.

In this second level design some concreteness has been added by implementing the *Model*, *View* and *Controller* class. The idea is that these classes will form the basis of the framework, without them the framework itself cannot function. In other words: the frameworks basic extensions have a primary role to facilitate the framework itself.

The next step is to add functionality to the framework so the framework can be used to build an application. In this case one extra extension will suffice, the *Age* class, which will be able to calculate an age given a birth date. To make the *Age* class an official extension it also has to implement the Extension class. Furthermore a module is required that will control the framework. A module named *WhatsMyAge* is added. This module has to implement the Module interface class to make it an official module, this means that the module has to implement the

execute_action() function as defined in the Module interface class. This is illustrated in the following UML diagram:

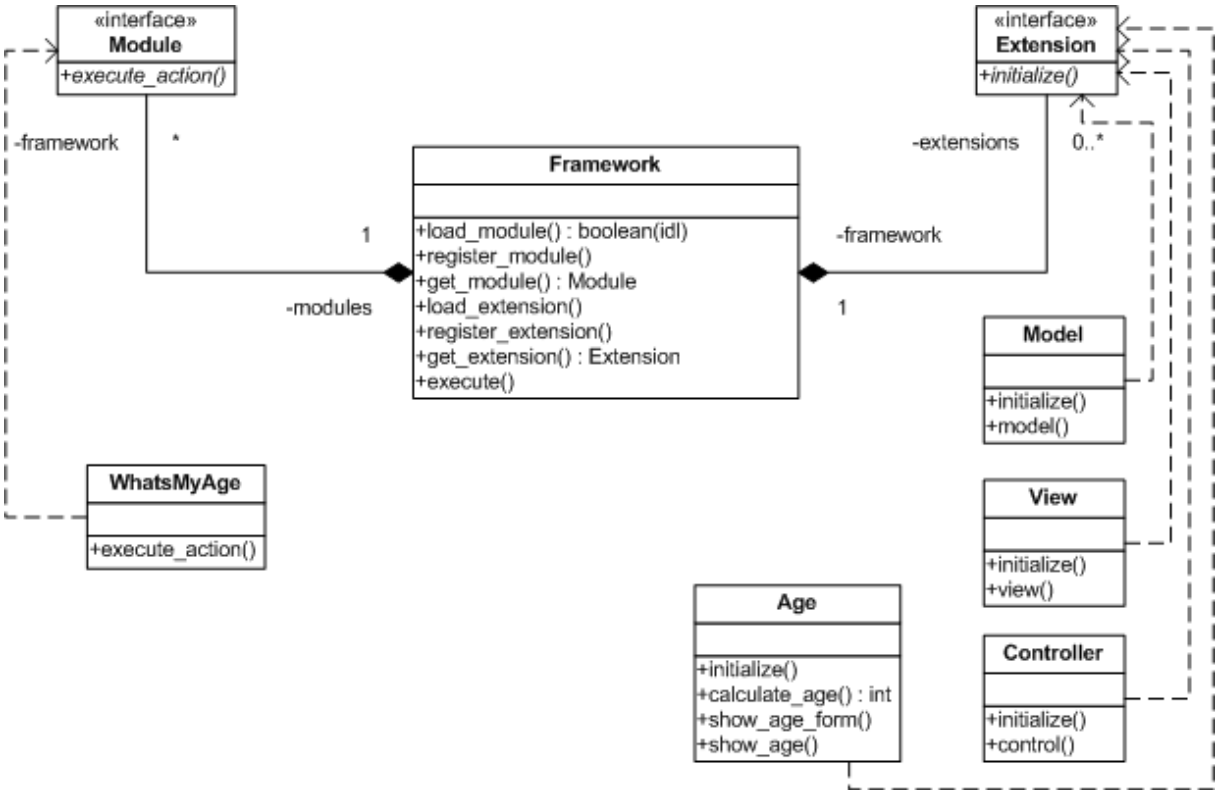


Figure appendix A-3: UML Diagram complete

Implementing the framework

As discussed in 7.1.1. the Framework class basically handles loading and registering the modules and extensions. It provides a method to retrieve a module or extension by a given name. When the Framework class is initialized its constructor will create the basic required classes for the framework to operate. In this case the *Model*, *View* and *Controller* class.

```
$this->load_extension("Controller");
$this->load_extension("Model");
$this->load_extension("View");
```

If all the basic extensions are loaded the constructor will perform a last task by issuing an *initialize()* command to all the extensions:

```
foreach($this->extensions as $key => $extension) {
	$extension->initialize();
}
```

At this point no module has been loaded. In this prototype the *Controller* extension will handle this aspect. Because of the web-based character of the application a variable holding the current section has been defined. This variable tells the system in which section of the web bases system the user currently is viewing. Of course in this example there is only one section, the section

where the user is able to input a birth date and the system will tell the age if the input is correct. So *Controller* will ask the framework to load the *WhatsMyAge* module. This is just an implementation choice for this prototype, e.g. the *WhatsMyAge* module could very well be loaded in the *Framework* constructor.

Implementing MVC

Important for this prototype is to learn how MVC can be implemented in a web-based application and furthermore in this framework. By looking at other web-based MVC frameworks the following is derived:

Class *Model*:

The model abstracts a domain specific data model and interacts with the data persistence layer on its behalf. This is usually a database but could very well be an XML or text document. Because of the simplicity of this prototype the model Extension itself is never used due to the fact that this application does not have any data to store or retrieve from e.g. a database. The birth date that the user specifies could have been used in the model, but for this prototype the date value is retrieved by *Age* extension itself.

Class *View*:

Adds presentation templates (e.g. styles, themes, etc.) to the data retrieved from the model before it is sent to the user's browser by the controller. The presentation is based on a markup language like HTML, XHTML or XML. In this prototype the View extension loads an XHTML template file with a given template name. For this application one template has been used.

Class *Controller*:

Receives or requests information from the user, routes these to the appropriate place in the application and returns the application's response to the user. For example: if a user presses a button in the web-based application the button will sent a request to the application and the controller will handle this request. The WhatsMyAge module has a list of different actions that depend on the current action of the user. The controller *Controller* will issue an *execute_action()* command to the current module loaded -depending on the current section- giving the current action. The module will then respond by executing the steps that are required for this action.

The Controller extension can be seen as a controller that dispatches the user request to the appropriate module. Because the module itself handles part of the request the module can, as stated before, be seen as part of the controller.

Class *WhatsMyAge* (the module):

Basically this module only has the function *execute_action()* that is invoked by the controller extension *Controller*. By a given action the module will tell the framework 'what to do'. In this example only one action is required to form the application. In this action two requests are sent to the framework. The first request is for the web-based form and the second request asks for the age. The latter only works if a birth date is specified (described later on in the Class *Age*) , but this is of no concern for the module. The following code describes the *execute_action()* function:

```
public function execute_action($action) {
	$this->current_action = $action;
	switch ($this->current_action) {
		case "calculate_age":
		default:
			$this->framework->execute("Age:show_age_form");
			$this->framework->execute("Age:show_age");
	}
}
```

The module sends a request to the framework by using the *Framework's* function *execute().* This request is sent with a string parameter so that the framework can invoke the right extension for the request. This string is composed by [Name of extension]:[Name of extension function]. In this example it is obvious that the module has to know the exact name of the extension and the exact name of the function within the extension. The ideal situation would be that the module requests a service from the framework without knowing any specific details of the extensions. For the purpose of demonstrating that the module sends it request to the framework and not directly to the extension this example will suffice.

Class Age (the extension):

This is the extension that facilitates the needs of the module *WhatsMyAge*. It relies on the *View* extension class to generate the web-based user interface. This extension has three functions:

- **calculate_age():** calculates an age given a birth date and returns this in an integer value.
- **show_age_form():** sends a request to the *View* extension to load an HTML template. In this case there is only one template, a web-based form with an input field for the birth date. This function is invoked by the *WhatsMyAge* module. The HTML template is sent directly to the user by the View extension.
- **show_age():** checks if the user has submitted his birth date, if so it retrieves the birth date value from the form data and used the extension internal *calculate_age()* function to calculate the age. Note that if the MVC model was fully used, the birth date value should be retrieved by using the *Model* extension.

A.3. SCREENSHOT

In the following screenshot this simple application is shown. First it has some framework debug output, telling which extension or module has been loaded. After this it will ask for the user's birth date and shows a form field where the user is able to enter this value. After the user presses the 'Calculate age' button the calculated age is shown at the bottom of the page.

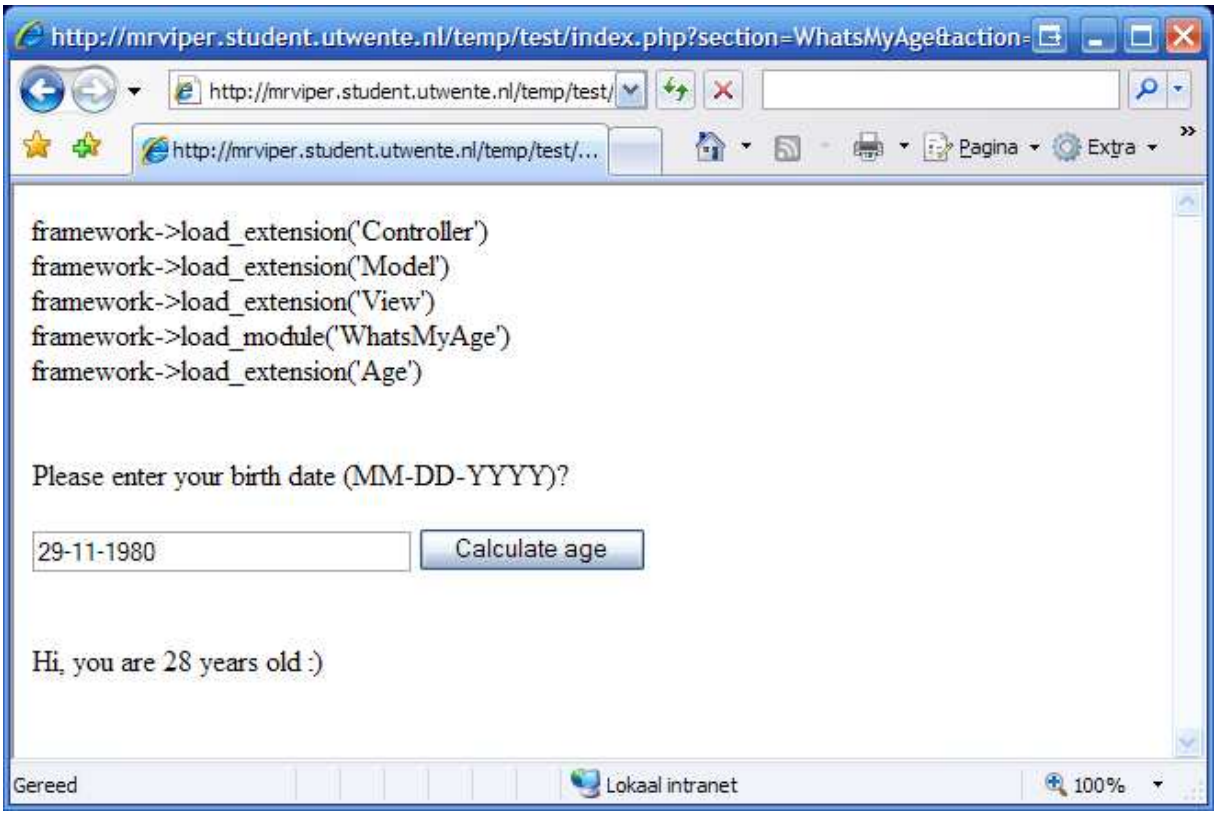


Figure appendix A-4: Screenshot of the application

A.4. DISCUSSION

During the design of this prototype a new concept was introduced. Is seemed that the Model, View and Controller classes could be seen as actual extensions. If we follow this line of reasoning every standard feature or aspect (e.g. MVC) of the framework can be put in an extension. Because MVC is required to meet our objectives it seems logical that the framework has to have a set of basic extensions to be able to function properly. But providing basic extensions could also mean that the framework is limited in deployment for a wide range of applications, this is in contrast with the research objectives. This introduced the idea that the design of the framework consists of multiple levels; the first level design forms the basis of the framework. This design is, for the most part, abstract (using interfaces instead of concrete classes) and complies with the research objectives. The second level design adds extensions to the framework with the primary role to facilitate the framework itself. For this prototype this approach seems the right one, other tests with other prototypes will have to determine if this multi level design will preserve

the research objectives. Is it possible to create a general set of rules or guidelines for creating the secondary design?

For a class to form an extension it has to implement the interface class ‘Extension’. This interface has to be uniform and sufficient for every extension to function within the framework, i.e. the framework uses this interface to handle and communicate with an extension. In this simple prototype it was possible to let the Model, View and Controller class act like an extension, but is it still possible in a more complex system?

During the development of this prototype it became clear how to implement MVC with an web-based application. Although this is a secondary design step, it might still be possible to make the design in such a way that it does not matter if the application is a web-based or normal application. At a glance the question seems to be yes, but this has to be determined in the following prototypes.

APPENDIX B. CROSS PLATFORM BOUNDARIES

The main objective in this case study is to determine if the design of the first case study can be applied in another programming language. In the first case study the programming language PHP (a web-based platform) was used, for this case study the popular object oriented language Java was used.

The application and framework design are based on the design of the first case study, the application itself is again an age calculation application. During the development the multi-level design aspect was used and the basic first level design has been further developed and improved.

B.1. DESIGNING THE PROTOTYPE

During the development of the second prototype the design of the first prototype was taken into consideration. Because the *Extension* and *Module* class are interfaces they could limit the development of these classes because it is only possible to specify empty functions, but it could be useful to store general data , such as a pointer to the framework, in the *Extension* and *Module* class. For this reason the *Module* and *Extension* interfaces classes were changed into abstract classes. The difference with interfaces is that abstract classes can have private variables and partial implementations of functions whereas interfaces classes only have empty functions. This gives for example the opportunity to declare a private pointer to the framework in the abstract *Module* and *Extension* class. In the first prototype these pointers were declared in the extensions themselves, this is not desired because it is against the objective of maximizing code reusability.

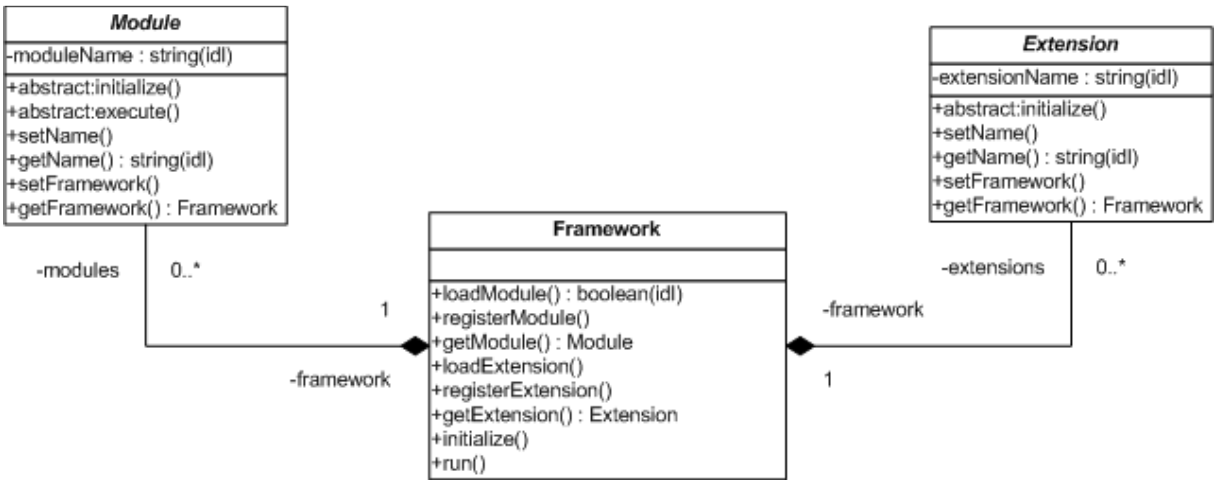


Figure appendix B-1: Framework design presented in a UML diagram. This is the first level design of the framework.

Because PHP supports abstract classes in its object oriented model, the new adjustments to the base first level design are backwards compatible with the first proto type. This means that the new design can be applied in the web-based programming language PHP and therefore securing the platform and system independency.

Class *Framework*:

The blueprint for this class has not changed a lot. The *execute()* function has been replaced by the function *run()* because this was name more applicable and a new function *initialize()* has been introduced. Although the names of the previous functions are written in a different matter (*load_module()* and *loadModule()*) they are still the same.

- **initialize():** In the first prototype the initialization of modules and extensions was performed by the constructor of the *Framework* class. To give the programmer more control this is put in a separate function. This means that if required the programmer is able to call the constructor of the framework, perform some other actions and they call

the initialization of the modules and extensions. In the first situation this was not possible.

- **run():** During the development of this prototype a missing link surfaced which was not a big problem for the web-based version. A method to 'start' the application did not exists. Because the web-based application only runs for a very short time (program is parsed, compiled, executed, output is sent to de users and the program exists), this application continues to run until the user tells the application to shut down. So this function *run()* tells the framework to start up the application.

Class *Module*:

PHP offers extra features that were used in the first prototype, in a result the design of the first *Module* class was limited. Because Java did not have the same support or some features even did not comply with the object oriented paradigm (i.e. dirty solutions) the class *Module* had to be extended. The extra functions are also needed to make the Module class more dynamic and deployable for a lot of different modules. Keep in mind that the final design should be able to produce modules for a wide range of applications. So the design of this Module class is probably not finished, e.g. communication between the module and the framework is still limited.

As explained before the class is now an abstract class as it was an interface class before. Some functions are already implemented, others still have to be implemented.

- **moduleName : String:** this is a private String variable that holds the name of the module. In the first prototype the class name itself was used. This gives the programmer more freedom because there could be limitation on the class name were there are (in principle) no limitations to this approach.
- **abstract:initialize():** as the *Extension* class in the first prototype already had an initialization function it is also advisable for the module class. In this case not all the initialization has to be done in the constructor of the module and is it possible to perform other actions before the initialization is called. This can for example be useful if a module depends on other modules which are not loaded yet. This function is abstract, this means that the concrete module has to implement this function.
- **abstract:execute():** This function replaces the function *execute_action()* that was used in the first prototype. By only using ' execute'as a name it is more general. This function is also abstract and thus needs to be implemented in a concrete class.
- **setName():** sets the private variable moduleName.
- **getName():** returns the private variable moduleName.
- **setFramework():** sets the private variable framework (visible in the connection between *Module* and *Framework*), this is a pointer to the *Framework* object.
- **getFramework():** returns the private variable framework, a pointer to the *Framework* object.

Class Extension:

With the same reasons the *Module* class was extended the *Extension* class is also extended. The only difference here is that the *Extension* class does not have a *execute* function. This is because the extension does not have to be executed, it only facilitates by means of functions.

All the other function in the *Extension* class are the same as in the *Module* class. Only the name of the extension is stored in the *extensionName* String variable. Because the other function provide the same functionality as described in the *Module* class, the function in this class are left aside.

B.2. WHAT ABOUT MVC?

At this point this is still a good question to ask! Although the MVC design might be the same, the low level implementation could be different. For the most part this has to do with fundamental design principle with web-based applications; web-based applications are stateless [7]. This means that with a web –based application the server can never be certain of what state the client is in. This differs with a normal application that runs on just one system.

With the desire to maintain the design of the first prototype and the fact that this application is not complex, the design of the first prototype has been applied for this application. So the *Model*, *View* and *Controller* are again built as extensions of the framework.

B.3. APPLYING SECOND LEVEL DESIGN

Because the MVC components are still the only required basic functionality of the framework, the additions for the second level design are almost the same as with the first prototype. A *Model*, *View* and *Controller* extension are added.

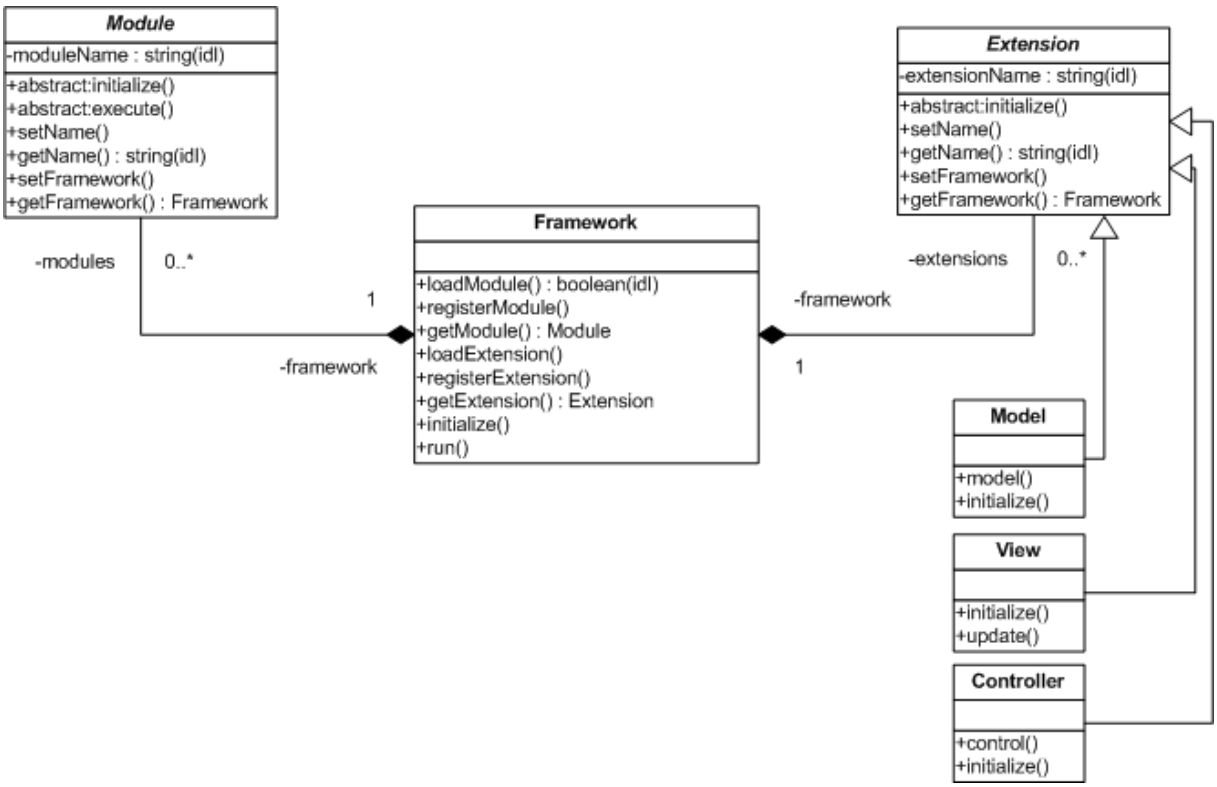


Figure appendix B-2: Applying second level design

The difference compared to the first prototype design is that an extension now inherits from an *Extension* class, instead of implementing it. Every extension also has to implement the abstract *initialize()* function.

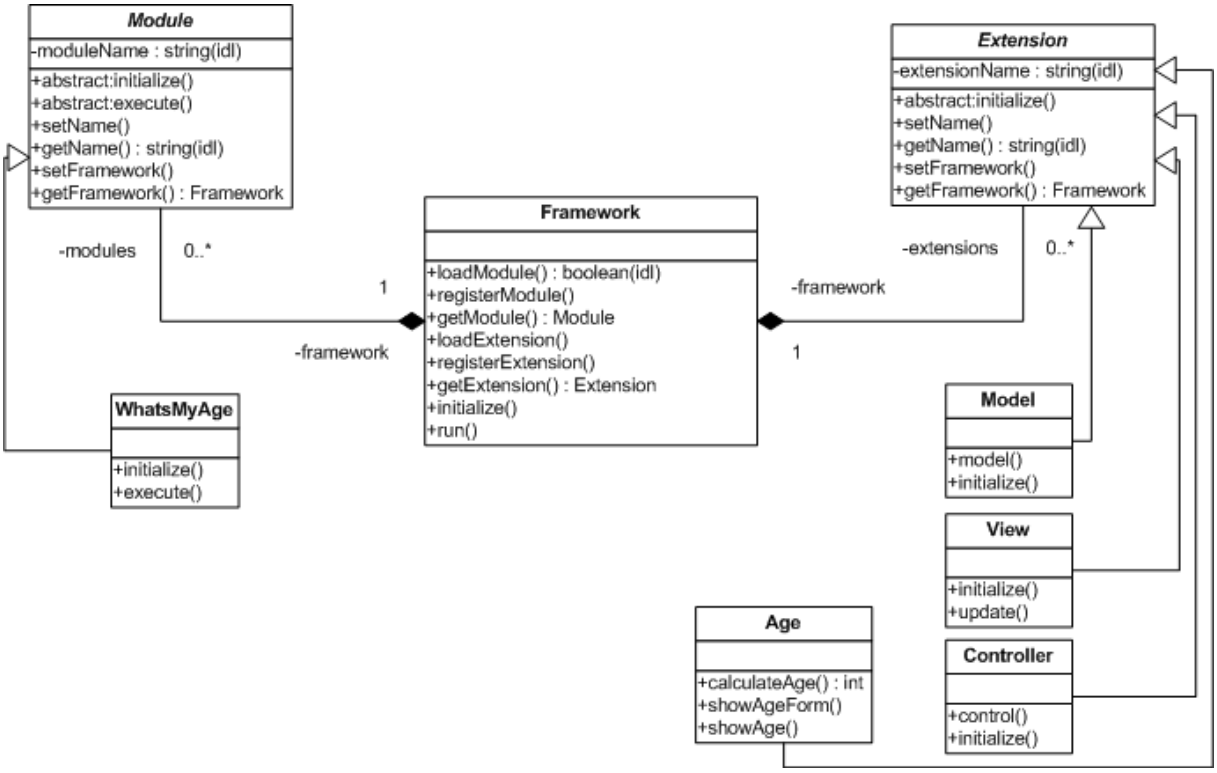


Figure appendix B-3: UML Diagram complete

B.4. IMPLEMENTING THE FRAMEWORK

The implementation of this framework did not cause a lot of trouble because the implementation of the first prototype basically had to be 'translated' from PHP into Java. Because of the object oriented paradigm this translation and applying the new design did, for the most part, go smoothly.

In Java the object oriented paradigm is much more strict compared to PHP. Therefore some parts of the first prototype could not be translated and other - more decent – implementations had to be made.

The following source code gives as an example of the easy 'translation'. First a part of the PHP *Framework* constructor that loads the basic extensions:

```
$this->load_extension("Controller");
$this->load_extension("Model");
$this->load_extension("View");
```

Translating this into Java for the second prototype it looks like the following:

```
this.loadExtension("Controller");
this.loadExtension("Model");
this.loadExtension("View");
```

Some troubles arose when translating the functions that load the modules and extensions. An extension or module class has to be dynamically instantiated within the *loadModule()* function. Doing this in Java was unknown territory. In the first prototype the following code is used:

```
public function load_module($module_name) {
	$file_name = "./".$this->modules_path."/".$module_name.".php";
	// Check if the main module file exists
	if (file_exists($file_name)) {
		//Include module
		include($file_name);
		// Register the module
		$module = new $module_name($this);
		$this->register_module($module);
		// Return module pointer
		return $module;
	} else {
		// generate exception
		echo "Module not found!<br/>";
	}
	return false;
}
```

In Java the function *load_module()* is rewritten into *loadModule()*:

The bold marked text indicates the code that dynamically creates the module class that has to be loaded. In the above examples the code for loading an extension is left aside because it uses the exact same approach.

The MVC extensions are for the most part implemented as in the first prototype, so nothing exiting here. The main difference is located in the implementation of the *View* extension. Here the Java Swing library was used to draw the graphical user interface components onto the screen of the user. The *Controller* extension still handles the users input and triggers the appropriate module, also in the prototype there is only one module; *WhatsMyAge*.

```
public boolean loadModule(String name) {
	try {
		// Dynamically create a class and a new instance.
		// Register this instance with the rest of the modules
		Class<?> c = Class.forName(name);
		Module module = (Module)c.newInstance();
		return this.registerModule(module);
	} catch (Exception e) {
		return false;
	}
}
```

B.5. SCREENSHOT

In the following screenshot the graphical user interface of the second prototype is shown. It has a simple input field for the user to specify a birth date and a button to tell the system to calculate the age. After the user presses the 'Calculate age' button the calculated age is shown in the field on the right side of the button.

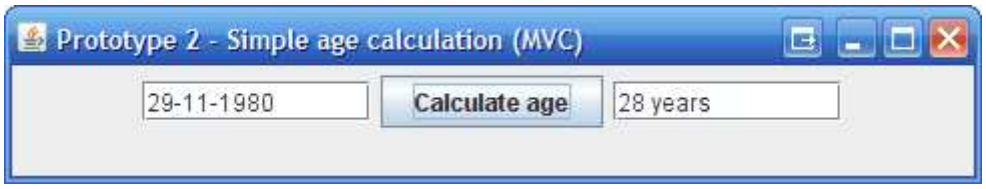


Figure appendix B-4: Screenshot of the application

B.6. DISCUSSION

This prototype's main goal was to test if the design of the first prototype could be used in another programming language. As this test was successful for in this test case in Java, it should be noted that the design of the framework is far from finished. It is even debatable is the *Framework* class is really required in this prototype because it only gathers all the modules and extensions. What can be concluded is that the basic design is evolving and thus far seems to be platform independent as long as the object oriented paradigm is used. Of course there are some differences in implementations using other programming languages, but the design should be the same in every case.

Although the design of the framework evolved, it does not resolve any of the questions that came out the first prototype. The reason for this is because the main purpose of this prototype is not to resolve these questions but to test the design in another programming language. As the questions will be handled in the following test cases one aspect did came back; the *Framework* class is far from the framework philosophy discussed in section Philosophy3.1.

APPENDIX C. FRAMEWORK DEPLOYMENT TEST

The main purpose of the first two prototypes was to design a framework with MVC support and check if this design could used in different programming languages. The design was first applied in PHP in the first prototype and an important design choice was made; the design of the framework has to be applied in two levels. During the implementation of the second prototype the main objective was to apply the first level and second level design of the first prototype. During this development the design further evolved. At this point the choices that have been made concerning the design are not yet proven. Furthermore is it not even sure if the choices are optimal and support the research objectives described in section 2.2.

In this prototype a game of Tic Tac Toe (also called “Noughts and Crosses”) has been built to show that the previous used design of the framework can be applied in different type of applications. What should become clear is that this framework proves to be useful. The game Tic Tac Toe has been chosen because it is simple and one of the most popular games in the world [8, 10].

C.1. PRINCIPLES OF THE GAME TIC TAC TOE

First the principles of the game Tic Tac Toe have to be correctly formulated before designing the framework that is able to build the Tic Tac Toe game. The game is played by two players who place their different colored or shaped game pieces on a 3x3 grid. The rules are simple: each player takes turn, each time marking an unoccupied location, with a cross 'X' or a nought 'O', until somebody wins or the grid is filled. The first player to fill a horizontal, vertical or diagonal row with three of their game pieces wins the game. There are in total 255,168 possible games [10].

Following the architecture described section 3.2, the game logic should be located in a module or in different modules. The framework should consists of extensions that provide de module functionality to build and play the Tic Tac Toe game.

To keep the prototype simple only a human and computer player exist. The human player always starts. Furthermore, this specific framework prototype can only be used to build Tic Tac Toe games, i.e. it is a 'Tic Tac Toe framework'.

C.2. DESIGNING THE PROTOTYPE

Looking back to the second prototype some improvements can be made. The *Module* and *Extension* for the most part share the same functionality, as shown in the following picture:

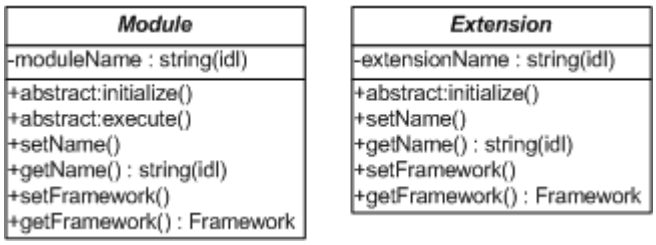


Figure appendix C-1: Module and Extension class share functionality

Because of these shared functions and the objective to optimize code reusability an abstract class *FrameworkComponent* is introduced. In this abstract class all the shared functionality is put together. The *Module* and *Extension* class still exist as abstract classes but are limited in functionality because they inherit their base functionality from the abstract *FrameworkComponent* class. This is represented in the following UML diagram:

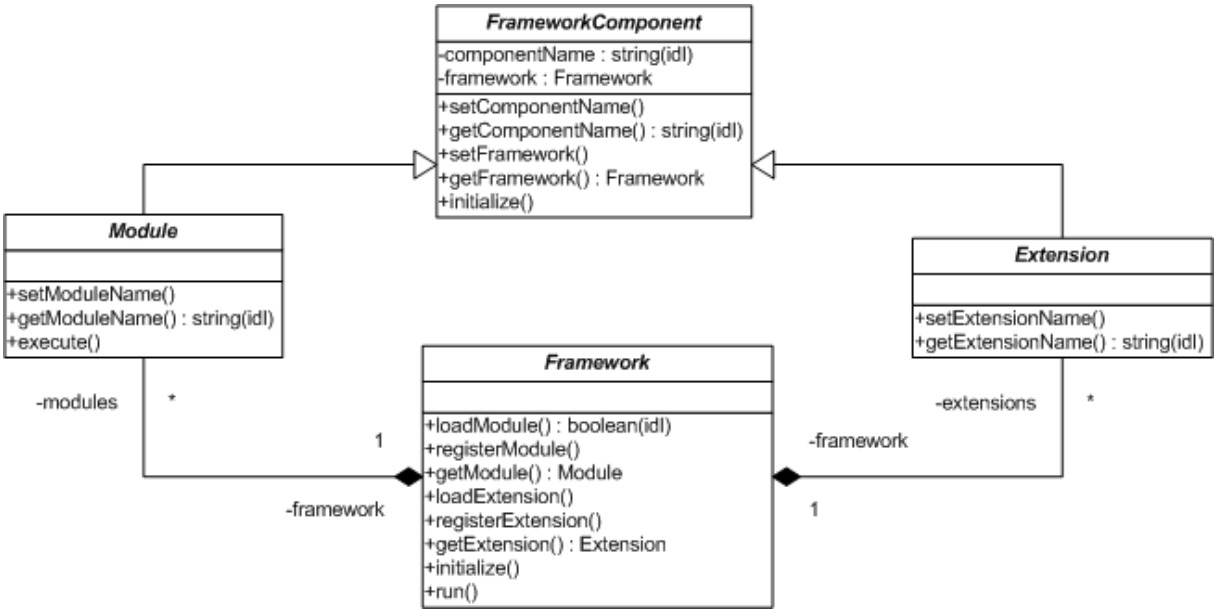


Figure appendix C-2: Framework UML - First level design

At this point the difference between a module and a extension is limited, but needed because of two reasons. At first it is most probable that the abstract classed will be extended with specific functionality, the *Module* class already has an *execute()* function that the *Extension* class does not have. Secondly the architecture clearly shows a difference with modules and extensions, this difference is important because the two have different purposes.

Other than the abstract *FrameworkComponent* no other design choices have been made compared to the second prototype.

C.3. APPLYING SECOND LEVEL DESIGN

Some interesting developments have occurred during the development of this third prototype; The *Model* extension, part of MVC, showed that a general design for different kind of applications is not very feasible. This emphasizes the approach of first and second level design. For the development of some extensions extra classes were required to support the extension classes. The question here is: are these classes extensions or not? Furthermore the boundary between the second level design and the final design containing all the components is diminishing, due to the fact that the second level design could already contain application specific parts and not only parts to facilitate the framework. This is noticeable in the following second level UML diagram:

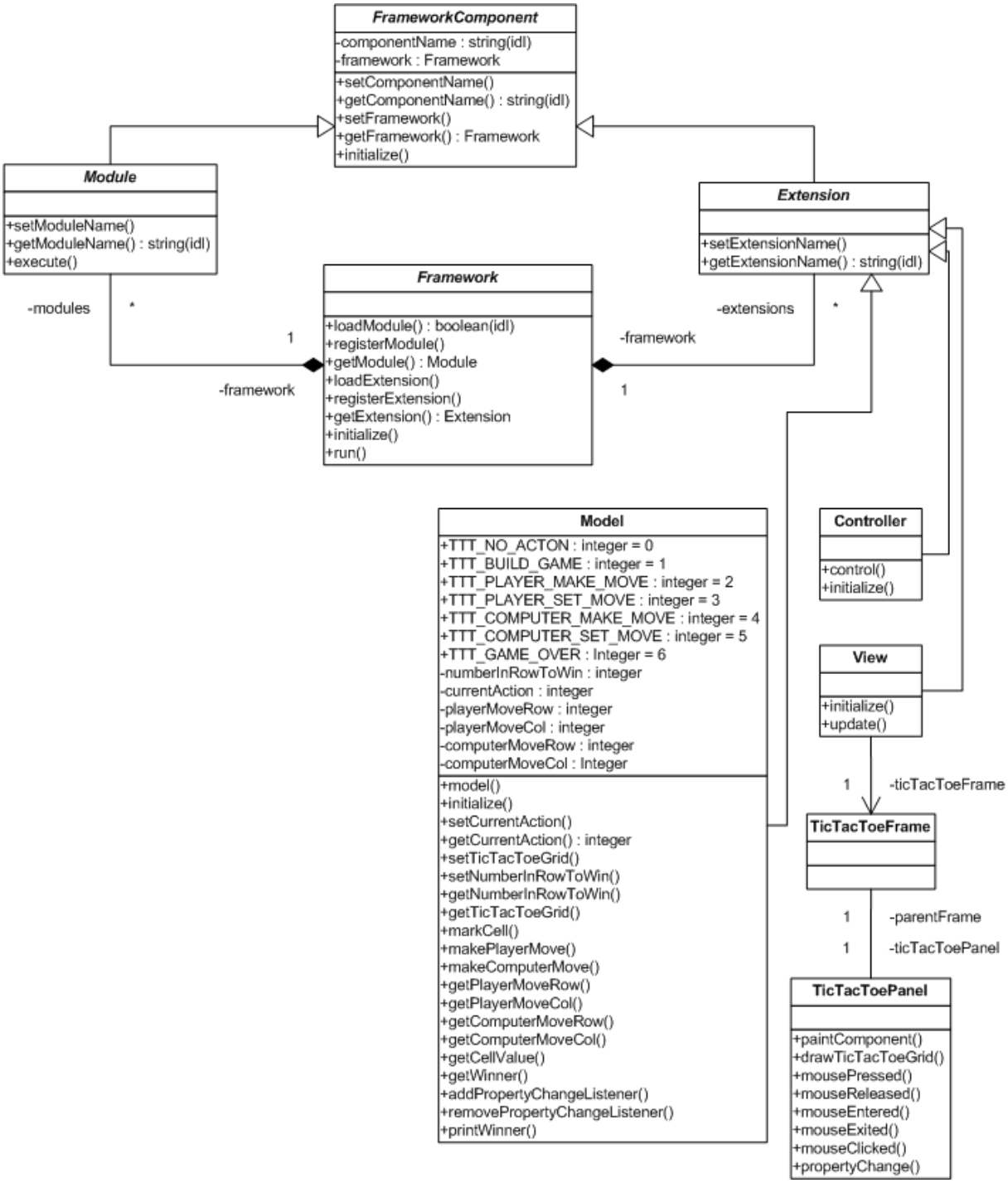


Figure appendix C-3: Second level design

In the above diagram the *Model* extension has been expanded with a lot of functionality for this TicTacToe game. The *View* extension makes use of a class called *TicTacToeFrame* which in turn makes use of a class called *TicTacToePanel*. As it is clear (by the function and class names) a lot of specific Tic Tac Toe funcionality comes back in this second level design. One may ask if it is necessary, could a more abstract approach be more suitable? On the other hand, this framework is intended only as a "Tic Tac Toe" framework, so the Tic Tac Toe functionality is designed to facilitate the Tic Tac Toe framework. Whatever the reasoning is, it is becoming clear that the

second level design not only consists of extensions and methods to facilitate the framework, but also out of application domain functionality, in this case the game Tic Tac Toe.

In the following UML diagram the complete design is represented:

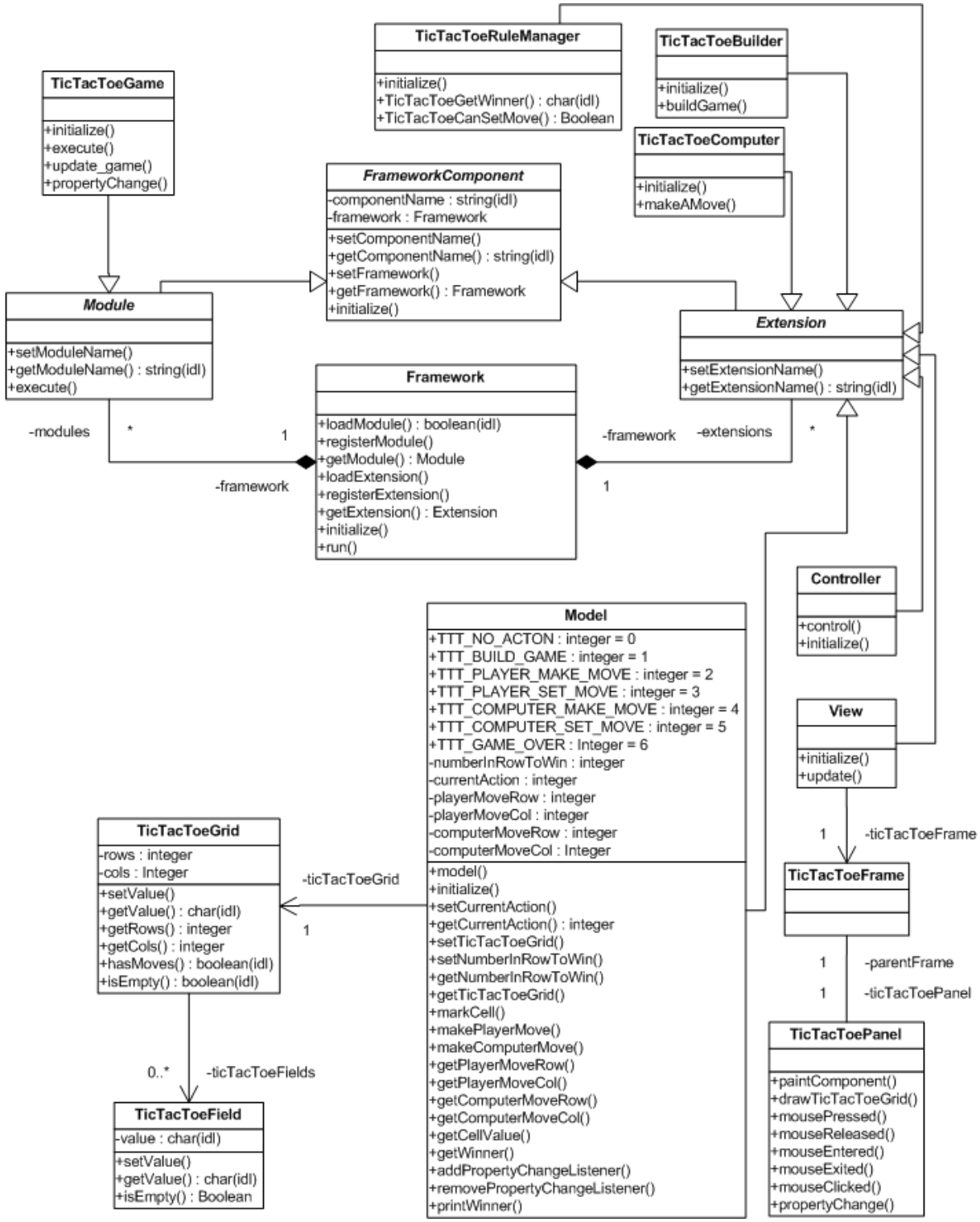


Figure appendix C-4: UML diagram complete

To elaborate on the diminishing border between the second level and final design, let's start with the *Model* extension. This class got a reference to a *TicTacToeGrid* class. One could ask why this is not visible in the second level design (Figure 7.11), it is part of the *Model* right? This is a good example where the borders between the second level and final design are diminishing; The *Model* extension is part of the second level, but parts of this class belong in the final design and at the same time the *Model* class contains final and application specific functionality.

The classes *TicTacToeGrid* and *TicTacToeField* are, like the *TicTacToeFrame* and *TicTacToePanel*, entities on their own but are not extensions. The question remains if this is a correct approach, if these kind of constructions are allowed it is unsure if the final framework is able to guarantee the objectives of section 2.3 , due to the fact that classes which are not built conform the framework design can give unpredictable results. As an example: in the first level design (figure 7.10) a clear separation exists between modules and extensions, a part of the application is either a module, the framework core or a framework extension, a simple class breaks this because it is neither one of the previous three. This example does not necessarily mean that the objective are not met, but what if a class starts acting as a module -and- as an extension? Of course it is not possible to restrict a programmer only to use modules and extensions, it is the choice of the programmer how to implement the framework. The purpose of the first level design should at first place be a strict guideline. Still there are classes in figure 7.12 that belong to the extension part of the framework but are not suited to be an actual extension (e.g. an extension class gives more overhead that might not be wanted). In this prototype no answer was found, therefore this issue will be addressed in the next prototype.

C.4. IMPLEMENTING THE FRAMEWORK

As stated before this prototype version of the framework is only a framework to build Tic Tac Toe games, and therefore the framework is limited. But what should become clear with this prototype is that building a Tic Tac Toe game using this prototype framework gives an advantage in development time, because a programmer that has the Tic Tac Toe framework only has to implement the module side. Next the implementation of the extension side is discussed to show how this is achieved by walking through every extension.

Class *Model*:

Like the previous prototypes this class is an extension representing the Model part of MVC. Unlike the previous extensions in this prototype this class holds all the data, in this case of the game Tic Tac Toe, this is clearly visible in figure 7.12. Although more functionality is put in this class it still handles all variables in memory, so no (xml) database or files are used to store/retrieve data. Once the application is closed all game data and states are lost. For the purpose of this prototype storing data is not essential and therefore left side.

Seven constants are defined to describe the different game states, in the module used as an 'action'. The *currentAction* variable is used to describe the current game state. With *numbersInRowToWin* the framework is able rule when a player wins the game, default with Tic Tac Toe is 3. The other variables *player/computerMoveRow/Col* describe the move of the player and the computer on the X-axis (col) and on the Y-axis (row).

In this class encapsulation preserved so all variables are private members and only accessible through public functions (see figure 7.12). Other functions are implemented to mark a cell on the game grid, check to see if there is a winner and to register a listener. A listener could be the *View* class, when the model is updated the *View* class as a listener is noted of this change and it can perform some required tasks like updating the graphical user interface.

Also a dependency for another extension exists. The functions *makePlayer/ComputerMove()* require the *TicTacToeRuleManager* (described later on) to check if the player or computer is allowed to set the requested move. The function *getWinner()* also requires the extension *TicTacToeRuleManager* (described later on) to determine if there is winner and who the winner is.

This *Module* class also has a reference to a *TicTacToeGrid* class where the grid of the game is stored in.

Class *TicTacToeGrid*:

This class holds the grid of the Tic Tac Toe game. It is flexible, e.g. it is not limited to the boundaries of a normal Tic Tac Toe grid of 3x3 cells. Although this class could be seen as part of the module it is still an entity on its own, not an extension. Due to the fact that it does not inherit the *Extension* class and also because the purpose of this class is to form a data structure, it is not a class that holds functionality which can be used by other extensions or modules.

Functionality is added to set or retrieve values from the grid or get the dimensions of the grid. Furthermore a function is added to check if moves are still possible, i.e. check if the grid fully filled.

A two dimensional array containing references to *TicTacToeField* objectes is used to store the grid.

Class *TicTacToeField*:

In this a simple class containing a character variable to represent an empty cell (field), a 'X'or a 'O'. It has functionality to set or retrieve the value of the character. Also a function *isEmpty()* is added to check if field is empty . This is necessary because the 'outside world' cannot know how an empty field is defined. In this case a space character is used to represent the empty field, in Java '\u0000' is used to represent a space character in a *char* variable.

Class *Controller*:

In this prototype this class is not used due to the fact that all graphical user interface actions are handled in the *View* extension. Although this somewhat demolishes the MVC pattern, no buttons or menus are used in the game' graphical user interfaces that should be handled by the controller. Only mouse clicks to determine where the player will set a move are registered, due to limited time this is registered in the View extension.

Although this does not completely complies with the MVC pattern and the first two prototype where MVC was addressed it is not a real issue for this prototype.

Class *View*:

This extension on itself is not very special. In the *initialize()* function initializes the *TicTacToeFrame* class which will handle the graphical user interface.

Class *TicTacToeFrame*:

This class extends functionality from the java *JFrame* class. *JFrame* is a component from the Java Swing library that can draw a frame/window in the graphical user interface environment which will look the same on every platform the Java application can be executed.

The *TicTacToePanel* class (described next) where the graphical user interface of the game is represented is also initialized here.

Please note that this class is not an extension.

Class *TicTacToePanel*:

To represent the graphical user interface of the Tic Tac Toe game this class is used. It extends its functionality from the Java *JPanel* class, a component from the Java Swing library that can draw graphics inside a *JFrame*.

This class registers itself as a listener with the *Model* extension. Every time the model of the game gets an update this class is notified for the update and in turn can update the graphics if this is required. This is typical MVC behavior.

In the *drawTicTacToeGrid()* function retrieves data from the *Model* extension and draws a graphical representation of the grid and its fields onto the panel. It does not matter what the dimensions of the grid are, this function automatically adepts.

The *mousePressed()* function registers a mouse click and retrieves the X and Y coordinates where the user clicked inside the frame. These coordinates are translated to the fields in the Tic Tac Toe grid. The grid dimensions are retrieved by asking the *Framework* the *Model* extension which returns the grid. Because the model is already retrieved the calculated mouse click to grid

coordinates is directly translated into the model. The mouse click itself is registered in the *TicTacToePanel* class and after translating the coordinates the click is transferred to the *Model* extension with the function *makePlayerMove()*.

Class *TicTacToeRuleManager*:

Although the game Tic Tac Toe is simple there are still some rules. One has to mark an empty location on the grid and somebody has to determine if there is a winner or a draw. For this it seemed logical to implement a rule manager, in particular a Tic Tac Toe rule manager. Because this should be framework functionality that facilitates the module Tic Tac Toe a *TicTacToeRuleManager* extension was built.

Basically it has two functions: *TicTacToeGetWinner()* and *TicTacToeCanSetMove()*. The first looks on to the grid to see if one of the two players is a winner else it returns false. For a standard 3x3 grid a player normally has to have three of their game pieces in a row to win. The second function checks if given grid coordinates if the field corresponding to the coordinates is empty. If it is empty a move can be set otherwise the move cannot be set.

Class *TicTacToeBuilder*:

Some functionality to build a new game was required. The choice was made to put this functionality in a separate extension; the *TicTacToeBuilder* extension. Given the number of rows, columns and the number of pieces in a row to win, the *buildGame()* function builds a new grid and prepares the *Model* extension for the new game.

Class *TicTacToeComputer*:

Another extension was added to the Tic Tac Toe framework that has functionality to make a move automatically. This functionality is required for the computer player. There is not a lot of intelligence in this extensions because of the time limit for this prototype. So the algorithm is kept simple, randomly select a field from the Tic Tac Toe grid, if the field is empty make the move.

Of course this extension is to show that functionality can be build in separate extensions. Another extension could have a better and more intelligent algorithm so to computer has a better strategy.

Class *TicTacToeGame*:

Other than all the previous discussed classes as extensions, this is a module containing all the “business logic”. It controls the Tic Tac Toe framework, telling it what to do at a certain time. One of the purposes of this Tic Tac Toe prototype is to demonstrate that building an application, or in this case a game, is easy if the framework can support it. In this prototype a Tic Tac Toe framework has been developed to make Tic Tac Toe games. Although limited in use, i.e. it can

only be employed for Tic Tac Toe games, the advantage is still clear in this module because the module only tells the framework what to do. So the amount of code used to create the game is limited. Furthermore, the Tic Tac Toe framework could be expanded with other extensions that allow the construction of other games. The Tic Tac Toe module does not have to be affected and can remain the same.

In the code example below comes from the function *update_game()* which is triggered when the game state changes. Every block of code represents a game state and it tells the Tic Tac Toe framework “What to do”.

```
case Model.TTT_BUILD_GAME:
	// Get builder extension from framework
TicTacToeBuilder builder =
(TicTacToeBuilder)this.getFramework().getExtension("TicTacToeBuilder");
	builder.buildGame(3, 3, 3);
// player makes the first move
Model model = (Model)this.getFramework().getExtension("Model");
	model.setCurrentAction(Model.TTT_PLAYER_MAKE_MOVE);
break;
```

```
case Model.TTT_PLAYER_MAKE_MOVE:
	// do nothing, wait for player
break;
```

```
case Model.TTT_PLAYER_SET_MOVE:
	// execute players move
	Model model = (Model)this.getFramework().getExtension("Model");
	model.markCell(model.getPlayerMoveRow(), model.getPlayerMoveCol(), 'X');

	// check if there is a winner
	winner = model.getWinner();
	if (winner != 'X' && winner != 'O') {
		// now it is the computers turn
		model.setCurrentAction(Model.TTT_COMPUTER_MAKE_MOVE);
	} else {
		// game over
		model.setCurrentAction(Model.TTT_GAME_OVER);
	}
break;
```

```
case Model.TTT_COMPUTER_MAKE_MOVE:
	TicTacToeComputer computer =
(TicTacToeComputer)this.getFramework().getExtension("TicTacToeComputer")
	// let the computer make a move
	computer.makeAMove();
break;
```

```
case Model.TTT_COMPUTER_SET_MOVE:
	// execute computers move
	Model model = (Model)this.getFramework().getExtension("Model");
```

```
        model.markCell(model.getComputerMoveRow(), model.getComputerMoveCol(),'O');
        // check if there is a winner
        winner = model.getWinner();
        if (winner != 'X' && winner != 'O') {
                // now it is the players turn again
                model.setCurrentAction(Model.TTT_PLAYER_MAKE_MOVE);
        } else {
                // game over
                model.setCurrentAction(Model.TTT_GAME_OVER);
        }
break;
```

```
case Model.TTT_GAME_OVER:
        // check if there is a winner
        Model model = (Model)this.getFramework().getExtension("Model");
        winner = model.getWinner();
        model.printWinner(winner);
break;
```

To make the actual Tic Tac Toe game using the Tic Tac Toe framework only the above had to be implemented. So the game logic is located in this module and the framework can easily be extended to support for example more games and keep this Tic Tac Toe module backwards compatible.

C.5. SCREENSHOT

In the following screenshot the simple graphical user interface of the game Tic Tac Toe is shown. The only input it accepts is a mouse click somewhere in the window.

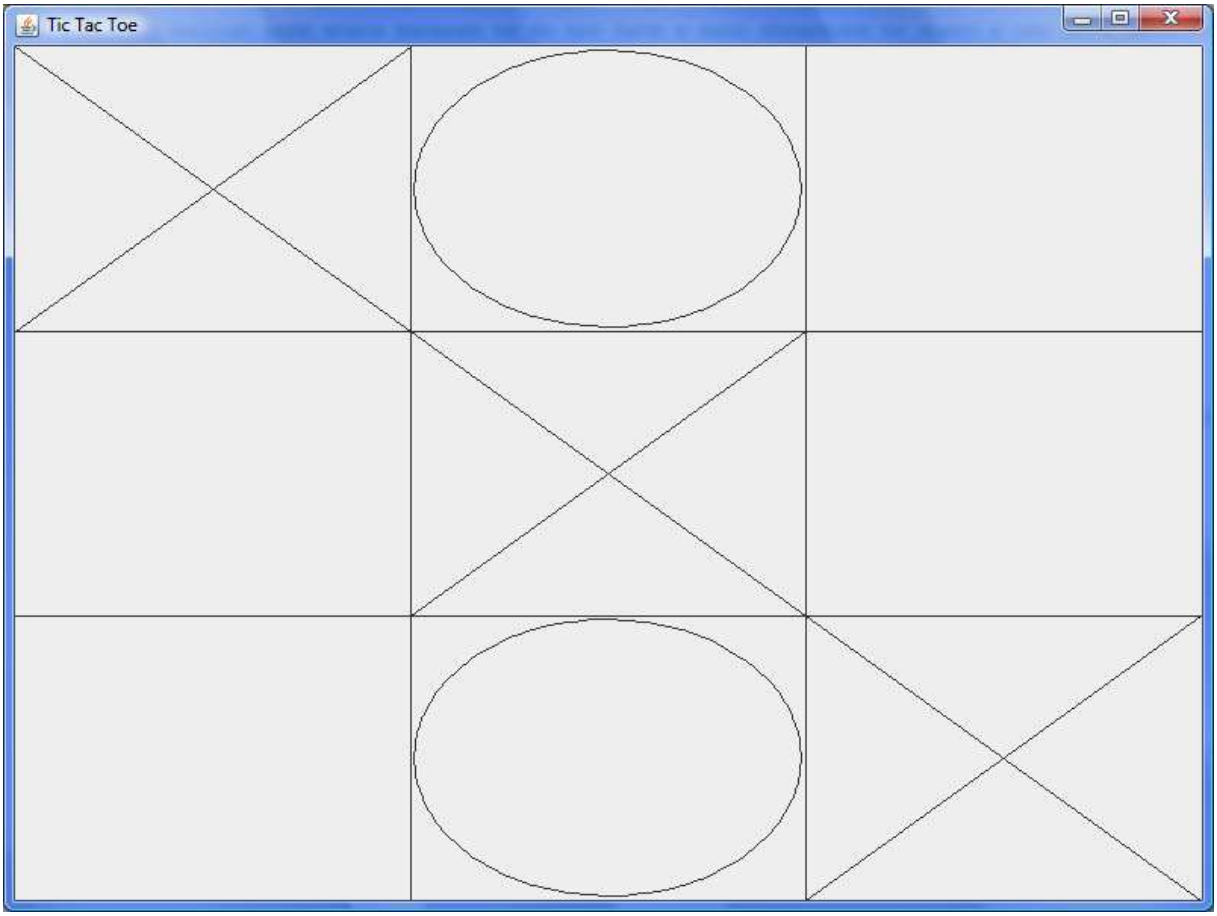


Figure appendix C-5: Screenshot of the simple graphical user interface of the game Tic Tac Toe built with the developed Tic Tac Toe framework. X has won!

C.6. OUTPUT DATA

In Figure appendix C-5, the 'X' (human player) has won the game. The Tic Tac Toe framework also outputs debug information to the console, the output of the above game has been captured and is represented here.

```
loadExtension:Model
loadExtension:View
loadExtension:Controller
Controller:constructor
loadExtension:TicTacToeBuilder
loadExtension:TicTacToeComputer
loadExtension:TicTacToeRuleManager
Module:TicTacToeGame:constructor loaded!
Model:initialize()
View:initialize()
TicTacToeFrame:constructor
Controller:initialize()
Module:TicTacToeGame:initialize
Module:TicTacToeGame:update_game(1)
TicTacToeBuilder: building new game
Module:TicTacToeGame:update_game(2)
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()
```

```
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()->cell data:---------
Module:TicTacToeGame:update_game(3)
Model->markCell(0,0,X)
TicTacToeGrid->setValue(0,0,X)
TicTacToeField->setValue(X)
TicTacToeRuleManager->TicTacToeGetWinner():no winner!
Module:TicTacToeGame:update_game(4)
TicTacToeComputer->makeAMove():calculating.... done!
Module:TicTacToeGame:update_game(5)
Model->markCell(0,1,O)
TicTacToeGrid->setValue(0,1,O)
TicTacToeField->setValue(O)
TicTacToeRuleManager->TicTacToeGetWinner():no winner!
Module:TicTacToeGame:update_game(2)
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()->cell data:XO-------
Module:TicTacToeGame:update_game(3)
Model->markCell(1,1,X)
TicTacToeGrid->setValue(1,1,X)
TicTacToeField->setValue(X)
TicTacToeRuleManager->TicTacToeGetWinner():no winner!
Module:TicTacToeGame:update_game(4)
TicTacToeComputer->makeAMove():calculating.... done!
Module:TicTacToeGame:update_game(5)
Model->markCell(2,1,O)TicTacToeGrid->setValue(2,1,O)
TicTacToeField->setValue(O)
TicTacToeRuleManager->TicTacToeGetWinner():no winner!
Module:TicTacToeGame:update_game(2)
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()->cell data:XO--X--O-
Module:TicTacToeGame:update_game(3)
Model->markCell(2,2,X)
TicTacToeGrid->setValue(2,2,X)
TicTacToeField->setValue(X)
TicTacToeRuleManager->TicTacToeGetWinner():diagional(\) win:X
Module:TicTacToeGame:update_game(6)
TicTacToeRuleManager->TicTacToeGetWinner():diagional(\) win:X
**********************************
* Game over! X has won the game :) *
**********************************
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()
View->TicTacToeFrame->TicTacToePanel:drawTicTacToeGrid()->cell data:XO--X--OX
```

C.7. DISCUSSION

During the development of this third prototype, a Tic Tac Toe framework to build Tic Tac Toe games, a lot of issues passed on. Still a lot of questions are kept unanswered and there are even new questions that need to be addressed. First let's look to the reason why this Tic Tac Toe prototype was built. As discussed in section 7.3 the main purpose of this prototype framework is to demonstrate that the design could be applied for different type of applications and that it is useful to use the framework instead of another approach. These two issues are the mainspring for this prototype.

Using the framework for different kind of applications; this is one of the objectives that drives this thesis project, described in section 2.2. In the first two prototypes frameworks were developed that could calculate an age. Although simple in design it showed total other kind of application than the current Tic Tac Toe prototype framework, a framework to build Tic Tac Toe games. Compared to the first two frameworks this is a completely different type of application, it is a simple game. So it seems that the objective is still valid. The basis – or first level design – could be the same for all kind of applications, this is more like a fundamental set of rules how the framework lower level (first level) should be implemented. At this point this first or lower level design does not state anything about the kind of framework it is going to be. This is decided by the extensions that will contribute to the framework (second and final level design). To conclude, the first level - or lowest level - design of the framework is application independent, due to the fact that the design of the first level does not tell anything about the application, it can be seen as a rule set for building a specific framework. For the designs following the first level design (second en final level designs) it is another story. Take the Tic Tac Toe framework in this prototype as an example, it is a framework to build Tic Tac Toe games, nothing more. In this case it is absolutely not useable for other kind of applications because it can only implement Tic Tac Toe games. This is not a limitation of the framework design, it is simple the choice of the programmer that implements the framework and determines what kind of applications the framework should be able to build. So this could be very limited like the Tic Tac Toe framework, but it could also be very wide, for example a framework that is able to build a wide range of board games.

For the second issue, if the framework is useful, it is not yet possible to give a satisfying answer. At this point the framework core (*Framework* class illustrated in Figure appendix C-4) does not contribute a lot and could easily be left aside. At this point the Tic Tac Toe framework only has a collection of modules and extensions and a method to retrieve a pointer to such a module or extension. Modules and extensions make the calls to functions within modules and extensions directly when a pointer is retrieved. For this prototype this approach was sufficient to show the usefulness of the framework. This is represented in the module *TicTacToeGame* as discussed in section 7.3.4. In this example the module tells the framework “What to do” when a certain state is reached, it holds the business rules (in this case game rules) of the application but knows nothing about the implementation in the extensions. So the Tic Tac Toe framework could be completely replaced by another framework that has more functionally but still be able to work the *TacTacToeGame* module. This separation between the modules and the framework (illustrated in Figure 2.6) makes this framework approach useful. The next step is to make the framework core more interactive; a module or extension asks or tells something to the framework core and it only communicates with the framework core and not directly to an extension or module. If this approach is applied the separation between modules and extensions becomes more distinct, because for example a module asks for functionality, the framework is able to look up this functionality in any way or at any location it wants. The module does not

know where the functionality is retrieved and it does not care as long as it gets a satisfying answer from the framework. This will be addressed in the next prototype. The registration of module and extension functionalities with the framework core will also be addressed.

Another issue the arose during the development of the Tic Tac Toe framework, what to do with classes that are not really extensions but are part of an extension? For example see the classes *TicTacToeGrid*, *TicTacToeField*, *TicTacToeFrame* and *TicTacToePanel* in, these belong to extensions be are not extensions. The questions remains if this could harm the objectives and ideals of the framework, i.e. it is unsure of the framework is able to guarantee the objectives of section 2.3, due to the fact that classes which are not built conform the framework design can give unpredictable results. As an example: in the first level design (figure 7.10) a clear separation exists between modules and extensions, a part of the application is either a module, the framework core or a framework extension, a simple class breaks this because it is neither one of the previous three. This example does not necessarily mean that the objective are not met, but what if a class starts acting as a module -and- as an extension? Of course it is not possible to restrict a programmer only to use modules and extensions, it is the choice of the programmer how to implement the framework. The purpose of the first level design should at first place be a strict guideline for the development of a framework.

One solution for this problem is to introduce special data type classes. These are classes that are not extensions but only hold data, e.g. like the *TicTacToeGrid* class. These special classes must conform to some sort of abstract or interface class, e.g. *ExtensionDataType*, making it possible that the framework core is able to communicate with these specials classes and thus making sure that the framework core has full control over its extensions and extension data types.

Another advantage is achieved if the extension data type approach is applied using a simple rule: a class in the framework conforms either to the framework core, an extension or an extension data type. All the used data types get their own class, e.g. an integer data type called “age” should not be an integer but a class named “Age” which holds the integer privately and conforms to the extension data type interface. This approach gives an effective method for finding functionality, e.g. find all functions that have the extension data type “Age” as input, instead of searching through all the functions that have an integer as input. More on this subject will be handed in the next prototype.

A disadvantage to the above solution is when, for example, an extension also has to extend another class, i.e. multiple inheritance is required. Not all programming languages support this feature and if a class must inherits from the *Extension* class (or another part of the framework) it cannot inherit from other classes. This will be addressed in the following case study.

APPENDIX D. DEALING WITH MULTIPLE INHERITANCE

After the development of the Tic Tac Toe prototype and looking to the issue of the *ExtensionDataType* interface another issue arose. An extension or a data type has to extend its properties from a super class, e.g. the *TicTacToeBuilder* class must extend its functionality from the *Extension* class (super class), thus making it an extension. If this extension should also extend functionality from another class multiple inheritance is required. The problem is that not all programming languages, like Java and PHP, support multiple inheritance. This framework concept should not obstruct the development of an application when multiple inheritance is required so a solution for this problem is presented here.

D.1. ASPECTS OF MULTIPLE INHERITANCE

Two important aspects play a role with multiple inheritance: parent class reuse and interface conformance. Parent class reuse means that a child class uses or 'reuses' the implementation of the parent class. Interface conformance means that a class A conforms to the interface of another class B so it is possible to use class A instead of B. In [13] a method is described how to achieve multiple inheritance in Java by using inheritance, interfaces and delegation.

Starting with a normal inheritance example a normal parent - child inheritance relation is represented in Figure appendix D-1. The *Child* class inherits (extends) from the *Parent* class. This means that the *Child* inherits all the functionality from the *Parent* class, in this case the function *foo()*. The class *Child* uses parent class reuse (the function *foo()*) and it conforms to the interface of the class *Parent*. So the class *Child* can be used as if it were the class *Parent*.

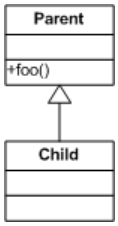


Figure appendix D-1: Inheritance example. The class *Child* inherits from the class *Parent*. This can also be described as class *Child* extends class *Parent*.

In Figure appendix D-2 multiple inheritance is demonstrated. In this case the child class named *Child* inherits from two parents or super classes called *Parent* and *OtherParent*. This means that the class *Child* uses parent class reuse and the interface conforms to the classes *Parent* and *OtherParent*. Because of this the class *Child* is able to act as a *Parent* or *OtherParent* class.

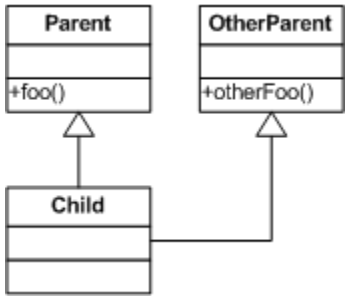


Figure appendix D-2: Multiple inheritance example

This class Child inherits the functions *foo()* and *otherFoo()* from its two parents. In this case it forms no problem, but what if the *Parent* and the *OtherParent* class both have the function *foo()*? This is one of the complexities with multiple inheritance that the developers of some programming languages limit inheritance to just one parent class.

Another issue with multiple inheritance is the so called 'diamond problem' [15]. A class is able to inherit from an ancestor through multiple paths in the inheritance graphs. The simplest form is

shown in Figure appendix D-3, it is shaped like a diamond. The diamond problem is an ambiguity that arises when the two classes *FillEllipse* and *Circle* inherit from *Ellipse* , and class *FillCircle* inherits from both *FillEllipse* and *Circle*. If a method in *FillCircle* calls a method defined in *Ellipse* (and does not override the method), and *FillEllipse* and *Circle* have overridden that method differently, then from which class does it inherit: *FillEllipse*, or *Circle*?

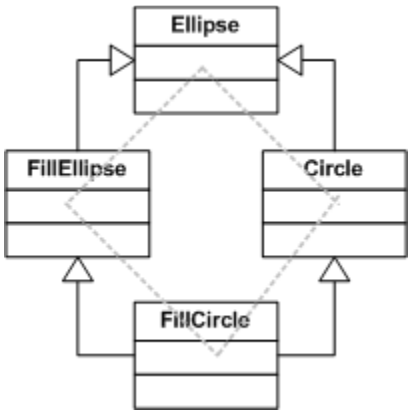


Figure appendix D-3: The Diamond Problem. A class is able to inherit from an ancestor through multiple paths in the inheritance graphs.

Different approaches exist for different programming languages, which makes it a complex issue. For this reason a lot of programming languages do not support multiple inheritance, just to keep the object structure 'simple'.

How is the problem of multiple inheritance related to the framework? Due to the fact that a lot of programming languages do not support multiple inheritance, like Java, this can induce a potential flaw in the design of the framework. In the current design an extension always has to extend the *Extension* class. This means that there is no room for another parent class if multiple in heritance is not allowed. The solution is to simulate multiple inheritance [13], i.e. simulate conformance and parent class reuse. It is not a complete solution which makes it possible to actually have multiple inheritance, but it is possible to alter the design of the framework in such a way that an extension can inherit from some other class and at the same time acts as an extension. The next sections will demonstrate how multiple inheritance can be achieved, or simulated as a better word.

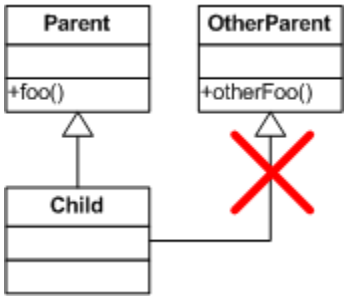


Figure appendix D-4: No multiple inheritance

D.2. PARENT CLASS REUSE BY USING DELEGATION

The first benefit of inheritance is that part of the implementation of the child class is also the implementation of the parent class. Classes do not have to be implemented as monolithic chucks of code, but can be built up piecemeal. This allows programmers to take advantage of code written in the past. This form of parent class reuse is the often cited advantage of using inheritance, but what is often under-emphasized is that inheritance is not required to reuse code [13]. Another way of reusing code is using delegation. Delegation also appears as an implementation strategy for the *Adaptor* pattern [14].

Delegation can be achieved by creating an (private) instance of a class for which code needs to be reused. Calls to functions that need to be reused can be passed through - or delegated to - the instance of the class that holds the code. This is represented in the figure right.

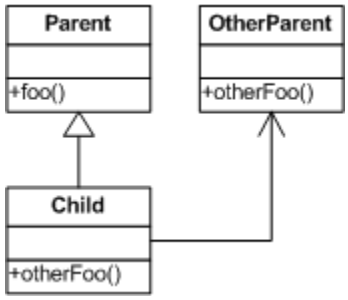


Figure appendix D-5: Delegation

The class *Child* now has an instance of the class *OtherParent* instead of extending it. The function *otherFoo()* in the class Child simply delegates all arguments to the *otherFoo()* function in the *OtherParent* class. Represented in actual Java code it will look something like this:

```
public class Child extends Parent {
	private OtherParent otherParent;
	public Child() {
		otherParent = new OtherParent(); // create new instance
	}

	// the function that needs to be reused
	public void otherFoo() {
		otherParent.otherFoo(); // this is de delegation part
	}
}
```

Achieving parent class reuse by using delegation unfortunately gives two problems. The first is clear, the amount of code required to achieve delegation can be as much as the actual code being reused. According to [13] the amount of “wrapper code” used for delegation in general be much less than the amount of code reused, so this is not a big problem. The second problem is more serious, in the given example the class *Child* is not able to act as the *OtherParent* class. Although the interfaces are the same there is no explicit relationship between *Child* and

OtherParent, this means that the class *Child* cannot be used as instances of *OtherParent*. So for the compiler the class *Child* has no interface conformance with the class *OtherParent*.

D.3. INTERFACE CONFORMANCE

The second important benefit of inheritance it interface conformance. This means that the interface of a child also includes the interface of its parent class. This means that any child instance can act as its parent, i.e. the child can be used where an instance of the parent is expected. This is possible due to the face that the child holds at least the same features as its parent.

Interface conformance can be achieved in two ways: the first is using the discussed inheritance, the second method is to use interfaces. An interface is an abstract class that only defines an interface that other classes have to implement. An interface class only holds empty methods and constant declarations and can never be directly instantiated. Another class that needs to be interface conformant with the interface class has the option to tell the programming language that is implementing the interface and thus be conformance to this interface. A relation between child and parent has been created using an interface, the child is now able to act as its parent because it conforms to the interface. The difference with inheritance is that the child class must implement all the functions defined in the parent interface class, so with interfaces no code is reused. The good thing about interfaces is that a class may implement as many interfaces as it wants. So even if the programming languages does not allow multiple inheritance the use of multiple interfaces is allowed.

D.4. SIMULATING MULTIPLE INHERITANCE

In the previous sections two methods were discussed: delegation and using interfaces. In this section they are put together to simulate multiple inheritance and thus achieving the benefits of parent class reuse and interface conformance.

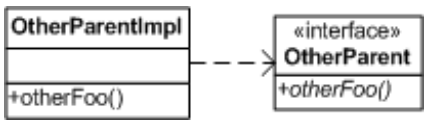


Figure appendix D-6: New OtherParent class

Suppose class *Child* wants to extend the classes *Parent* and *OtherParent*, because of the programming language the class *Child* can only extend one class, e.g. *Parent*. By using the delegation and the interface technique it possible for the *Child* class to have parent class reuse and interface conformance to the *OtherParent* class. This requires a change in the design of the *OtherParent* class as illustrated in Figure appendix D-6. The *OtherParent* class is now an interface defining only an empty *otherFoo()* function. The *OtherParentImpl* class is a new class that

implements the class *OtherParent* , this means that the implementation of the *otherFoo()* function can be found here. With this new design it is now possible to simulate parent class reuse and interface conformance. In Figure appendix D-7 the design of the new *OtherParent* class is applied to simulate multiple inheritance:

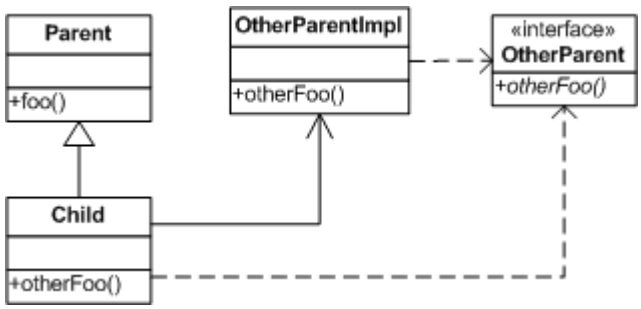


Figure appendix D-7: Simulating multiple inheritance

So what happened? First of all the class *Child* still inherits the class *Parent*, nothing changed here. *Child* now holds an instance of the *OtherParentImpl* class, this is used for delegation. In order for *Child* to conform to the interface of *OtherParent* it is implementing the interface *OtherParent*. This means that *Child* needs to implement the function *otherFoo()*, but instead of implementing the whole function over again *Child* delegates it to the instance *OtherParentImpl*. Because of this *Child* now conforms to the interface of *OtherParent* and it is reusing the code of *OtherParentImpl* and thus it is simulating multiple inheritance.

According to [13] the following rule should always be applied to ensure the simulation of multiple inheritance:

> ***To facilitate code reuse in Java, every class intended for reuse by inheritance should also have a matching interface that is used in place of the class wherever possible.***

D.5. LIMITATIONS OF THE INTERFACE-DELEGATION TECHNIQUE

In figure 7.19 multiple inheritance was achieved by a new design of the *OtherParent* class. If this it is not possible for the programmer to alter the design of a class, e.g. it is part of a closed source library or the programming language API, it is not possible to apply the technique of interface-delegation. This should not form a problem due to the fact that one inheritance is allowed and in programming languages without multiple inheritance support it is likely to assume that the API does not contain classes that should be used with multiple inheritance. Furthermore, with this framework the programmer is in full control over the classes. If the programmer follows the design layout in this thesis all the necessary steps to simulate multiple inheritance are taken care of.

D.6. USING SIMULATION OF MULTIPLE INHERITANCE

The next step is to alter the basic (also called first level) design of the framework in such a way that multiple inheritance can be simulated for the extensions and modules is required. In the Figure appendix D-8 the old design is represented.

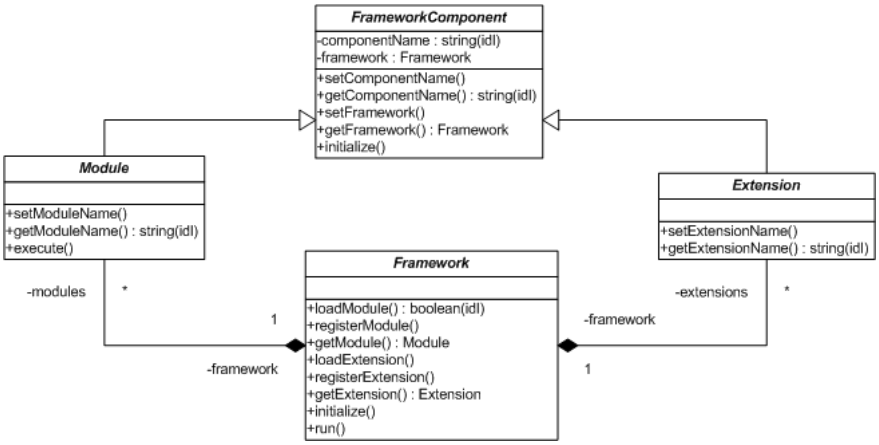


Figure appendix D-8: The 'old' basic design

All four components have to be altered, due to the fact that every class could be reused with inheritance. Thus following the rule stated in [13], all classes that can be reused an matching interface has to exist. By applying the rule the new basic framework design is represented in Figure appendix D-9.

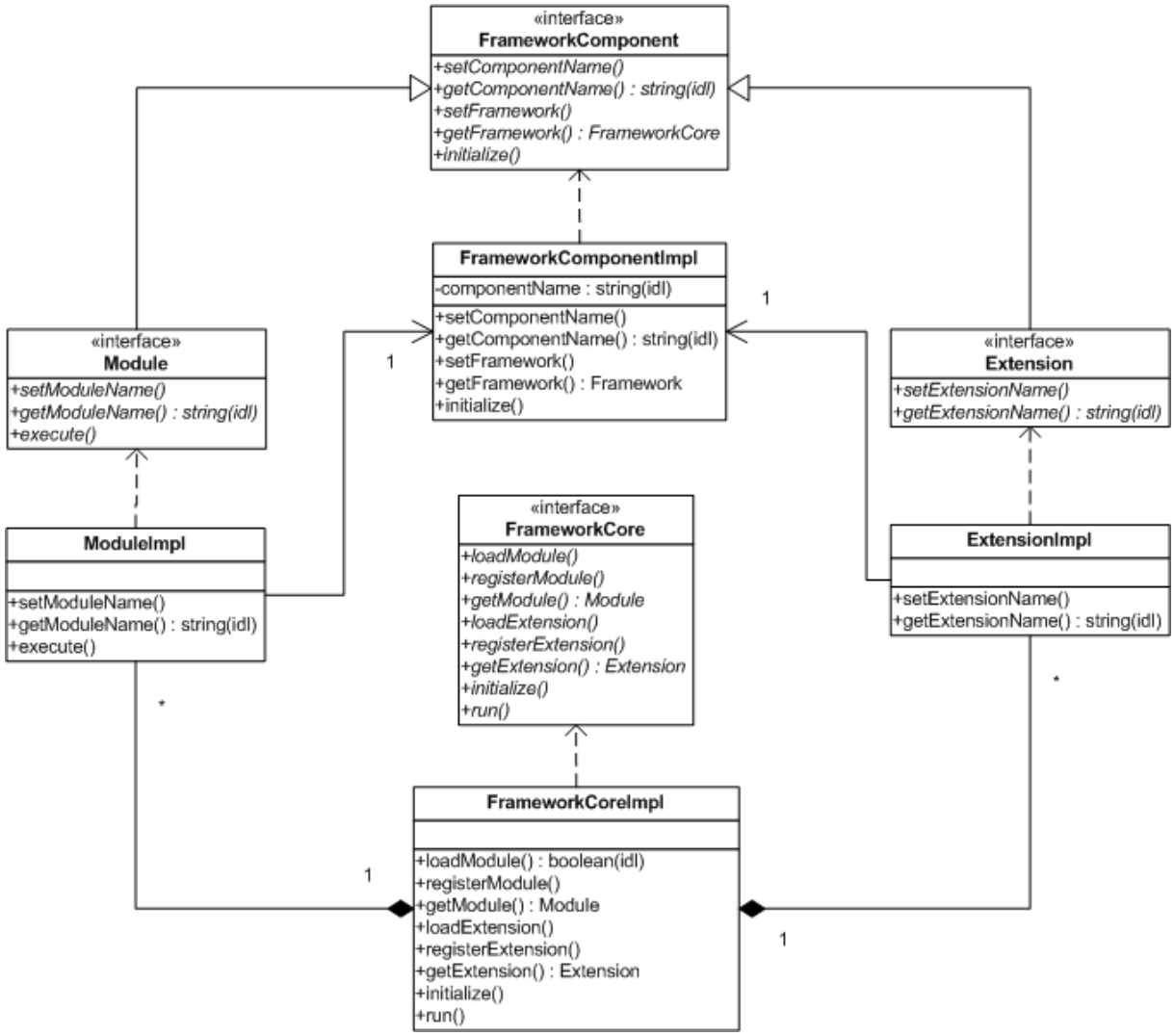


Figure appendix D-9: The new basic design that enables simulation of multiple inheritance

In the Figure appendix D-10 an extension *SomeExtension* has been added. This extension does not have to inherit from any other class, so it inherits all functionality from the *ExtensionImpl* class. Please note that for the overview other classes are left aside (grey). For the total overview see Figure appendix D-9.

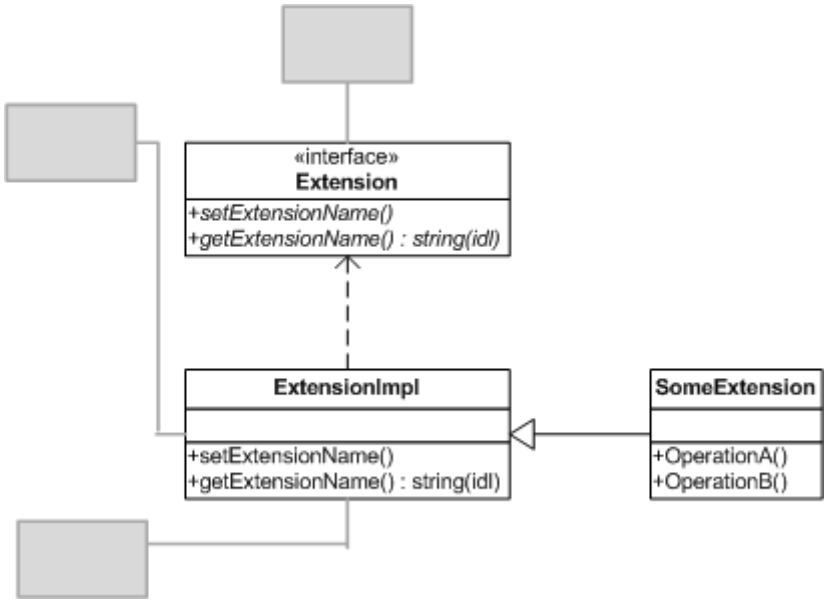


Figure appendix D-10: An extension example

In the following figure an extension *SomeExtension* has been added that also has to inherit from the Java Swing Library *JFrame* class. The programmer has no control over the *JFrame* class and because it is not possible to inherit from two classes in Java, the simulation of multiple inheritance technique can be applied here. This is illustrated in Figure appendix D-11.

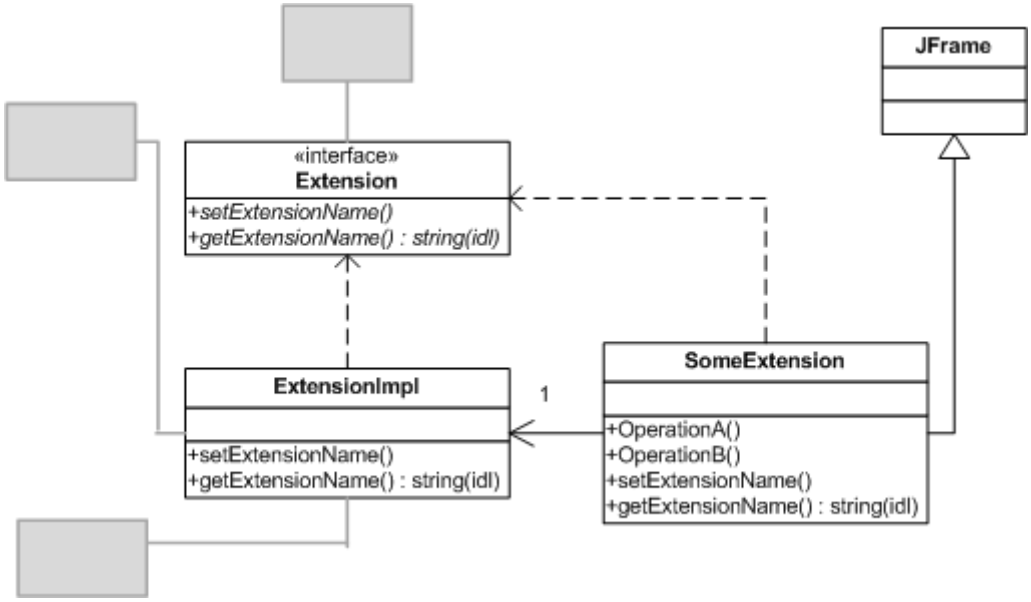


Figure appendix D-11: Simulating multiple inheritance example

The *SomeExtension* class now inherits functionality from the *JFrame* class and by applying the simulation of multiple inheritance technique the *SomeExtension* class can also acts as if it is an *Extension*. The programmer has to implement the Extension interface but delegation can be used to simplify the amount of code, i.e. operations are just passed on to the *ExtenionImpl* class.

D.7. DISCUSSION

In some cases multiple inheritance can be required. By following the design of the framework an extension always had to inherit from the *Extension* class, making the change of requiring multiple inheritance bigger. Because a lot of programming languages, like the popular Java, do not have support for multiple inheritance. It is possible to simulate multiple inheritance, but it required some changes in the design of the framework.

With inheritance two aspects are important: parent class reuse and interface conformance. Parent class reuse means that the child class has all the functionality of its parents class, it inherits this functionality. Interface conformance means that the child class has a least the same interface as the parent class and can therefore act as if it is the parent class.

Parent class reuse can be simulated by using the delegation technique. An instance of the class that needs to be reused has to be instantiated in the child class, instead of implementing all the operations over again the implementations only holds code that passes on the request to the instantiated class.

Interface conformance can be achieved by using interface classes. The class that needs to be inherit has to have a corresponding interface class. This interface ensures interface conformance. If at the same time the delegation technique is applied inheritance is simulated. With this method it is possible to simulate multiple inheritance.

There are some limitations when using multiple inheritance, e.g. protects members of the delegated class are not available. Furthermore every class that might inherit needs to have a corresponding interface if not the programmer can create it but full control of this class is required. The core Java API does not give the programming this control. These limitations do not form a problem for this design of the framework, no protected data members are used and full control over the classes is available.

APPENDIX E. EXISTING BROKER ARCHITECTURES

Until this point the framework core did not have a very important role in the whole picture. It only contained a list of modules and extensions and some functionality to register and retrieve the modules and extensions. Request from modules to extensions and vice versa did not go through the framework but directly through pointers obtained from the framework core. So the framework core did have a role but not as important as it should have (see figure 2.4 how it should be).

The framework core must have a more facilitating role, modules should not talk to the extensions directly but only talk to the framework. The framework itself, depending on the implementation, can look for a suitable extension that has the right functionality. This can be in the most simple form that the framework core has a list of all functionalities corresponding to certain extensions or very complex in such a way that the core tries to find the best functionality within a set of extensions that may not even run on the same computer. The implementation of the framework core might even have the choice of more than one extension that can be used for a particular functionality, load balancing could for instance be used in this decision making.

Looking at the behavior that the framework core should have it shows references to a middleware service broker architecture, which is found in CORBA (Common Object Request Broker Architecture) and also in web services. Before designing the core and possible reinventing the wheel related work has to be addressed.

E.1. RESEARCH QUESTIONS

> ***QE-1: The CORBA and Web Service architectures show resemblance to the current framework architecture, is the framework concept - or parts of - the same as the architecture of CORBA and/or web services?***
>
> ***QE-2: If the framework concept has (partly) the same architecture, can the architecture of CORBA and/ or web services be reused instead of reinventing the wheel?***
>
> ***QE-3: Can CORBA or a Web Service be used to replace the framework?***

E.2. CORBA

Common Object Request Broker Architecture (CORBA) is an architecture and a specification of a collection of standards describing facilities for client-server communication and interaction between distributed software objects [17, 18, 19, 20]. It has been developed and managed by the Object Management Group (OMG), a large consortium of software and hardware vendors.

The main idea behind CORBA is to let applications from different vendors implemented in different object oriented programming languages talk to each other without the other application having to know how and in what programming language it is written. To establish communication between objects an universal language is required. OMG defined the Interface Definition Language (IDL) for describing interfaces of software objects. According to the CORBA specification an interface is a description of the set of possible operations a client may request of an object [24]. An interface does not specify the internal data representation or executable code used to implement an object. In practice, an IDL interface specification may also contain declarations of types, exceptions and constants. In order to facilitate re-use and extensibility of classes, IDL supports multiple inheritances among interface definitions. IDL is independent of programming languages, and may be used to describe objects implemented using a variety of programming languages, compilers or operating systems [26].

Using the IDL on the client side (the side that is requesting a service) a stub class is automatically generated, on the server side (the side that offers the service) a skeleton class is automatically generated. The stub class is instantiated by the client and it talks to the stub object as if the implementation is located within this class. The stub performs tasks such as converting parameters and returned values into a form that allows them to be transmitted. The skeleton class, located at the server, acts as an interface between the actual object implementation and the communication layer. For the client this process is transparent, this means that the client itself is in fact not aware of the distributed nature of the system architecture.

Figure appendix E-1 shows a request being sent by a client to an object implementation. The Client is the entity that wishes to perform an operation and initiates the request on the object. The Object Implementation is the code and data that actually implements the object.

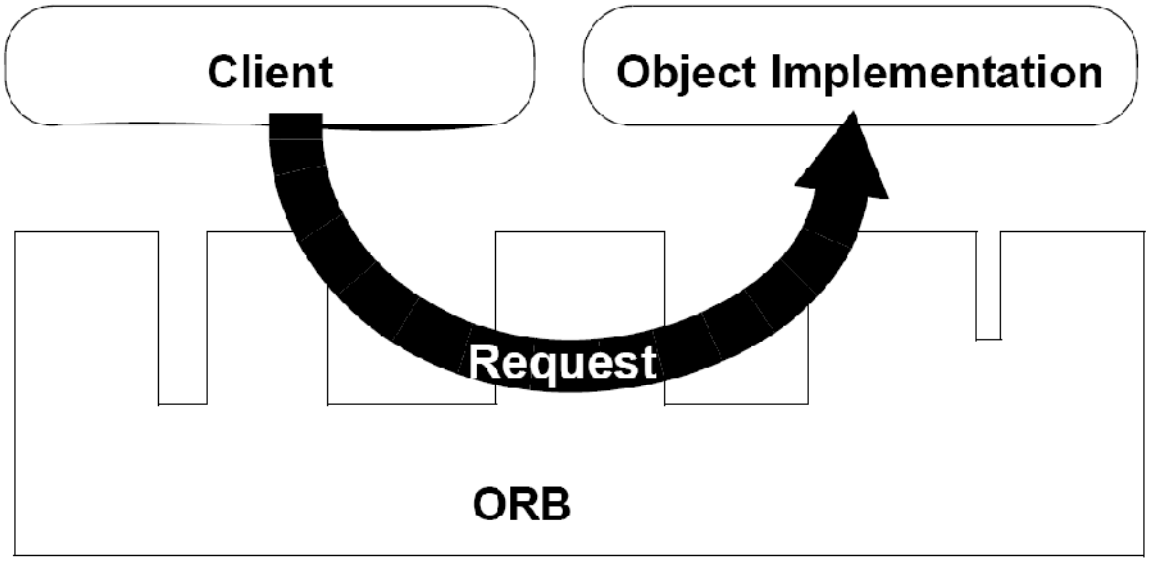


Figure appendix E-1: A request through the Object Request Broker

The communication between the client and the stub is handled through the Object Request Broker (ORB). An ORB serves as a software interconnection bus between clients and object implementations [21]. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface.

Figure appendix E-2 shows the structure of an individual ORB. The interfaces to the ORB are shown by striped boxes and the arrows indicate whether the ORB is called or performs an up-call across the interface.

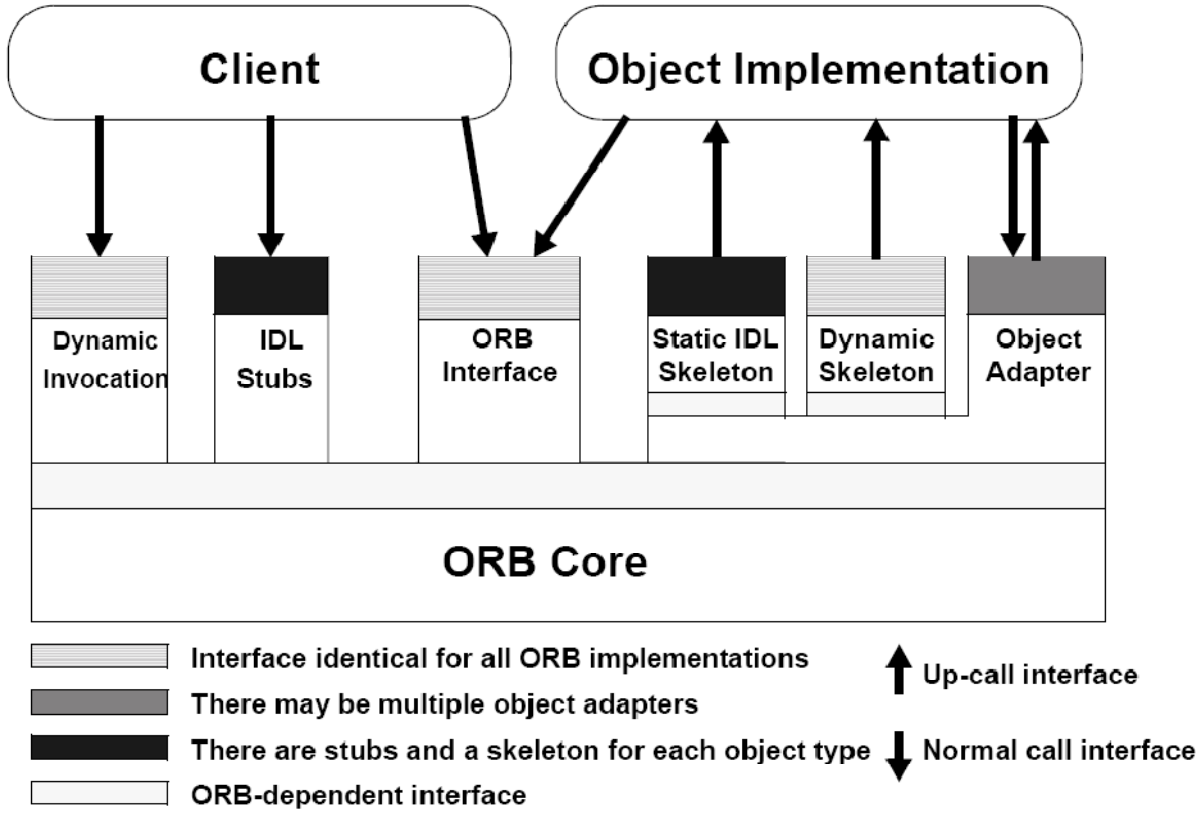


Figure appendix E-2: The Structure of Object Request Interfaces

An ORB must provide an Interface Repository. In this repository all the information of the IDL specifications of connected objects is compiled and stored. The repository can be examined by other objects that are using the ORB and try to find a certain object. This gives the advantage that an object that is requesting service from another object on the ORB can request the service without having prior knowledge of the other object's interface .

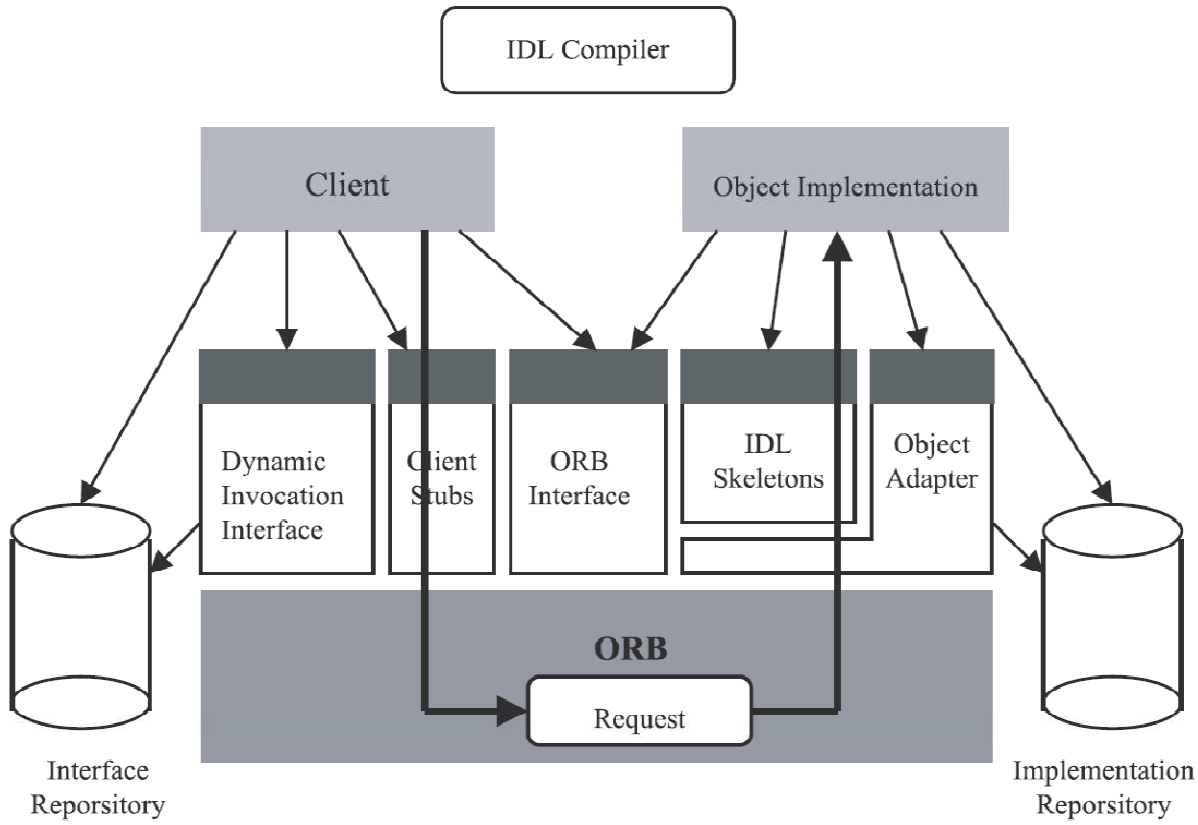


Figure appendix E-3

CORBA is structured so as to allow integration of a wide variety of object systems [7]. Several major components go into constructing CORBA.We will now briefly describe the basic concept of CORBA and its components. To invoke an operation on a remote object implementation, a client must first bind to that object. The ORB first checks whether or not the remote object implementation exists. If it does not, an instance of the object implementation is initialized. Binding to a remote object will create a local proxy for the remote object in the client program, and the requests on the local proxy will be delivered to the remote object. The object implementation replies with the results of the invocation to the client via the ORB once the object implementation completes execution of the invocation. The client may send prior requests to the same object implementation without re-establishing the communication channel. The client program can also invoke a remote object implementation via the Dynamic Invocation Interface (DII). That is, the DII allows a client to directly access the underlying request transport mechanisms provided by the ORB core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. It can query the Interface Repository so as to obtain information about a remote object implementation. Analogously, the Implementation Repository is used by the Object Adapter to find the location of an object server if the target object is not activated. [21]

E.3. WEB SERVICES

Another aspect that utilizes a broker mechanism like CORBA are Web Services. Web Services are designed as a software system to support interoperable machine to machine interaction over a network [22, 23]. Because of the similarities between CORBA this subsection does not go into details about Web Services, only the important aspects are handled here.

The official definition by the WC3 group [24]:

> ***A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.***

Web Services show a lot of similarities to CORBA, instead of using IDL (Interface Definition Language) WSDL (Web Services Description Language) is used. WSDL is an XML based language that provides a model for describing Web Services [25].

As CORBA uses IIOP (Internet Inter-ORP Protocol), Web Services use SOAP (Simple Object Access Protocol) as a transport layer. SOAP is a simple XML based protocol specification for exchanging structured information in the implementation of Web Services in computer networks. [26] It is mostly used over HTTP (Hyper Text Transfer Protocol), but can also be used over other existing protocols.

For service discovery a UDDI (Universal Description Discovery & Integration) registry used. The service requestor (client) asks the UDDI registry to lookup a certain service by a given WSDL. If an appropriate service provider (server) is found the corresponding WSDL of this provider is returned. The service requestor is now able to establish a connection and request services of the service provider. In the following diagram this is illustrated:

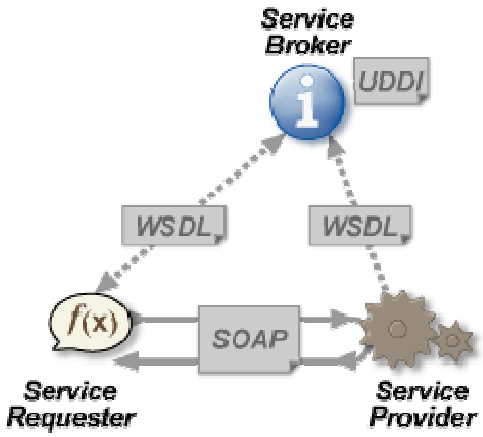


Figure appendix E-4: Web Service overview

Service providers make their services available by putting descriptions of the services into a UDDI registry. Different UDDI registries can be used. Service requestors can find service providers by issuing a WSDL request containing a service description to a specific UDDI registry.

E.4. CORBA VS. WEB SERVICES

Web Services have an advantage over CORBA; Web services use SOAP over HTTP as transport, due to the fact that HTTP requests run through port 80 they are mostly not blocked by company firewalls. CORBA uses IIOP over TCP/IP and other ports than port 80, changes of a firewall blocking IIOP request is bigger than SOAPS request. Another advantage of SOAP is that it is XML based, this means that other vendors can easily adapt to the specifications of SOAP because XML is well supported in different programming languages. Furthermore, the IDL of CORBA is not XML based like WSDL used by Web Services. These advantages might be the reason why Web Services are more popular than CORBA.

Despite the advantages and disadvantages CORBA is a more complex low level implementation platform compared to Web Services. A Web Service can be seen as middleware for middleware that would sit on top of CORBA and relegate CORBA as a lower-level implementation platform [29].

E.5. DISCUSSION

The current design of the framework seemed to have some similarities to other broker architectures. In short, the main idea: The framework consists of modules, a *Framework Core* and extensions. Modules request functionality from the *Framework Core* and the *Framework Core* tries to locate the requested functionality from a set of extensions. This architecture so far does not state how the modules, *Framework Core* or extensions communicate with each other. This could mean that a concrete implementation of the framework utilizes some form of

middleware to let the components talk to each other. In any case, the *Framework Core* shows broker like behavior, due to the fact that it has to 'localize' functionality.

Many broker kind specifications exist. Two specifications are compared in this thesis: CORBA and Web Services. They both have some sort of broker architecture. Although other specifications exist, due to time limits only these two are compared.

First of all let's look to the main idea behind CORBA: CORBA is an architecture and a specification of a collection of standards. The main objective of CORBA is to facilitate communication between distributed software objects. Distribution is also the key factor with Web Services. Although they describe it as a system that supports interoperable machine-to-machine interaction over a network, it is also distributed. Important to see here is that both technologies are designed to operate with different distributed systems over a network, this is a key difference with the framework.

To explain this key difference let's look to the framework. The main purpose of the framework is to facilitate software development of enterprise applications. Although the design does not state anything about how the components (Modules, Framework Core and Extensions) are connected, is not the goal to facilitate software development in a distributed manner. The purpose of the framework is primarily to built applications not distributed applications, this is a key difference compared to CORBA and Web Services.

How about the resemblance. The framework consists out of modules, a *Framework Core* and extensions. The modules can be seen as clients or service requestors that ask some sort of service to the Framework Core. The Framework Core can be seen as broker like the ORB that facilitates the request to the appropriate Extension. An Extension itself can be seen as a server or service provider that is able to provide service to the Framework Core and thus to the Modules.

Like CORBA and Web Services the Framework is a specification/architecture. This means that different concrete implementations may exist. As long if the specifications are followed the concrete implementation may be called a CORBA or a Web Service implementation. This rule also applies to the a concrete framework implementation. It is the authors opinion that the design of the framework itself is at a higher level compared to CORBA and Web Services. This is substantiated by two reasons: 1) the intended goal of the framework is different; the goal of the framework is to build applications, CORBA and Web Services facilitate communication between applications. 2) Due to the fact that a concrete instance of the framework could be realized using CORBA or Web Services to let the framework components communicate with each other indicates that CORBA and Web Services stand below the framework.

E.5.1. RESEARCH QUESTION QE-1

> ***QE-1: The CORBA and Web Service architectures show resemblance to the current framework architecture, is the framework concept - or parts of - the same as the architecture of CORBA and/or web services?***

The framework, CORBA and Web Services show resemblances by design but they operate at a different level and they have a different goals. The framework is for building applications, CORBA and Web Services facilitate communication between applications.

Looking at the functionality that the Framework Core should provide compared to CORBA and Web Services, some components appear to have resemblances. For the most part the ORB, the CORBA broker. The idea behind the Framework Core is that an incoming request from a module gets processed and sent to the right extension that will handle the request. The Framework Core could be very complex, e.g. an concrete instance might search for the right extension and if multiple extensions exist that could process the request the Framework Core has to sort out which one to use. This is typical service broker behavior. CORBA and Web Services both have service broker. In CORBA the actual ORB is the broker and with Web Services the broker can be found in the UDDI registries.

In the authors opinion the CORBA ORP specifications comes closest to the idea of the framework. As far as the documentation of CORBA [17, 18] the ORB specifies the ORB itself does not look automatically for services. The client gives specific information for a service by supplying IDL and the ORB is able to look in its own interface repository if the specific service is available. The Framework Core requires the same functionality but goes a little further, a model should be able to ask the framework functionality but this should not have to be specific. For example, an concrete instance of the Framework Core is implemented in such a manner that it accepts natural language as input. The text “Calculate the age of person X given the birth date 29-11-1980”. The Framework Core uses same sort of artificial intelligence to understand the request, finds suitable extensions and selects the best extension for the job. Of course this is an extreme example which will not be required for this project, but it gives a good reflection what the main idea behind the framework should be.

Web Services do not have an ORP like CORBA. They communicate using an XML format called SOAP. SOAP messages are transferred over existing protocols like HTTP. A separate interface repository exists called the UDDI registry. In this registry service providers register themselves and the functionality they offer. Service requestors can query the registry to find specific service providers and can then set up communications with the service provider. The UDDI registry does not handle the communication between the service requestor and the service provider, it only tells where who offers the service. Web Services compared to CORBA are simpler and in result

also limited. Because they do not have an ORB like CORBA they do therefore not stand as close to the framework as CORBA does.

E.5.2. RESEARCH QUESTION QE-2

> ***QE-2: If the framework concept has (partly) the same architecture, can the architecture of CORBA and/ or web services be reused instead of reinventing the wheel?***

Although there seems no evidence that there are CORBA or Web Service implementations that do what the framework should do, it should be noted that CORBA and Web Services are specifications. This means that concrete implementations may differ, e.g. one could be more complex than the other and even have more functionality, as long as the specifications are correctly implemented it may be called CORBA or a Web Service. This means for example that an CORBA ORB could be implemented in such a way that it conforms to the specifications of the framework and the *Framework Core*. It can even be turned around; the framework could be implemented in such a way that it complies to the framework specifications and to the CORBA specifications, meaning that an instance of the framework could also be a CORBA implementation and vice versa.

E.5.3. RESEARCH QUESTION QE-3

> ***QE-3: Can CORBA or a Web Service be used to replace the framework?***

Because the CORBA architecture comes closest to the framework architecture the question is if parts of the CORBA architecture can be reused. The answer is not just a simple yes or no. Because both are architectural specifications it is possible to implement both specifications as one. So the CORBA architecture could be applied to implement the framework architecture. However, putting the CORBA architecture by default in the architecture of the framework seems not the right solution. Due to the fact that CORBA is designed to facilitate communications between distributed applications and the framework is designed to built a single application. CORBA is not designed to do what the framework has to do, therefore CORBA brings too much overhead to achieve this. However, a particular instance of the framework might use CORBA to connect the components to each other.

The architecture of Web Service is less close to the framework because Web Services do not have an ORB like CORBA which looks like the Framework Core. So it seems unlikely that the Web Service architecture can be used to replace the framework architecture. However, just like CORBA it is possible to combine the architecture of the framework and the architecture of Web

Services. This could mean that for an instance of the framework the different components act as Web Services and the Framework Core is some sort of UDDI registry. Although this is possibility please note that this is not the intended design of the framework, due to the fact that this approach would probably give too much overhead.

APPENDIX F. ENTERPRISE APPLICATION EXAMPLE

The previous prototypes showed that the framework works, but the clear advantage for MASER/NextSelect developing enterprise applications is still missing. Furthermore, the design still needs proof of concept to show that the idea works. In this chapter some parts of already existing web-based applications will be simulated to show how the framework should be working in practice and it will demonstrate the proof of concept. In the Introduction subsection the applications that are built by NextSelect in cooperation with MASER Engineering will be briefly addressed so that it is clear what kind of applications the framework should be handling and how it is supposed to be deployed.

F.1. RESEARCH QUESTIONS

QF-1: The philosophy of the framework is that modules are not programmed but contain only a set of instructions (i.e. script) for the framework to execute. This could be in a simple scripting language but also in some form of natural language that the framework is able to interpreted. If this is not possible it is allowed to use the same programming language for the modules as the rest of the framework, what is allowed to implement and what is not ?

QF-2: How are modules and extensions going to communicate with each other?

QF-3: A this point nothing is said about the possibility of multiple instances of the same extension. In some circumstances this might be necessary. Is it necessary and more important: it possible?

QF-4: A module or extension has two initializations points; the first upon loading, the constructor of the module/extension is called. The second initialization is when the Framework Core calls the initialization() function of the module/extension. What are the rules of placing initialization code in either the constructor or the initialization function of a module or extension?

QF-5: Is it possible to load modules or extensions after the initialization point of the Framework Core when circumstances demand it ?

F.2. INTRODUCTION

NextSelect currently manages four enterprise web-based applications. Although these applications are different by business logic they share a common framework. The problem is that every application has its own version of this framework and when bug-fixes or new features are introduced to a certain framework all the others have to be updated also. Sometimes these updates are not even possible because the frameworks versions are too different. At the start of a new enterprise application the most recent updates framework was taken and the new application was built around the new framework. A new version of the framework was born which is further developed during the development of the new application, the old framework remained the same. This results in two different versions of the framework, maintaining the framework becomes more and more difficult as new applications are built and more framework versions exist.

Another problem with the framework is that there is no clear separation between business logic and framework core components. This means that a framework cannot be transferred to another application without consequence because it contains application specific code. This should be unacceptable but time issues in the past required easy and fast solutions.

The new framework should resolve the following issues:

1) All applications share the same framework. This does not have to be the same instance, i.e. every application could still have its own instance of the framework. A new version of the framework needs to be deployed to all applications without any problems, so that every application has the same version of the framework.
2) Business logic and framework core components are separated. This is requirement for the previous point to work. If the framework contains business logic of a certain enterprise applications this would mean that other applications get the same code. Not only can this give unpredictable situations it should not be the intention that business logic from other customers is shared between different applications from other customers.

The new framework must also make the development of enterprise applications easier. Although the architecture of the framework cannot guarantee this, due to the fact that this is dependent of the actual implementation of the Framework Core and its Extensions, this section is going to show how this achieved. This will be added to the specifications of the framework, these specifications give some limitations on how the framework should be implemented but make sure the objectives of the framework are met if a new instance is implemented.

As mentioned before the existing enterprise applications are all web-based. This means that the current framework generates HTML. In every application the structural layout is the same, only the styles (colors, fonts, etc.) are different.

Examples of these applications are shown in the following screenshots:

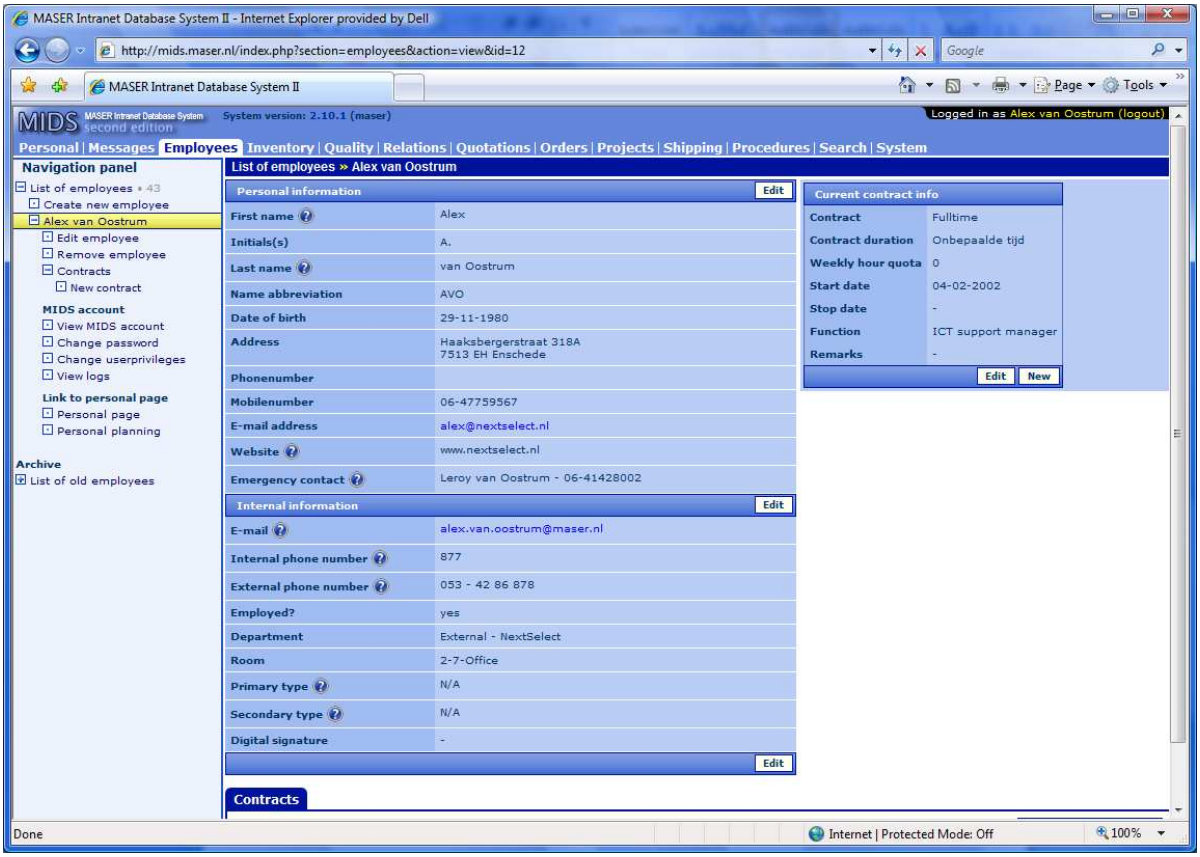


Figure appendix F-1: MASER Intranet Database Systeem (MIDS)

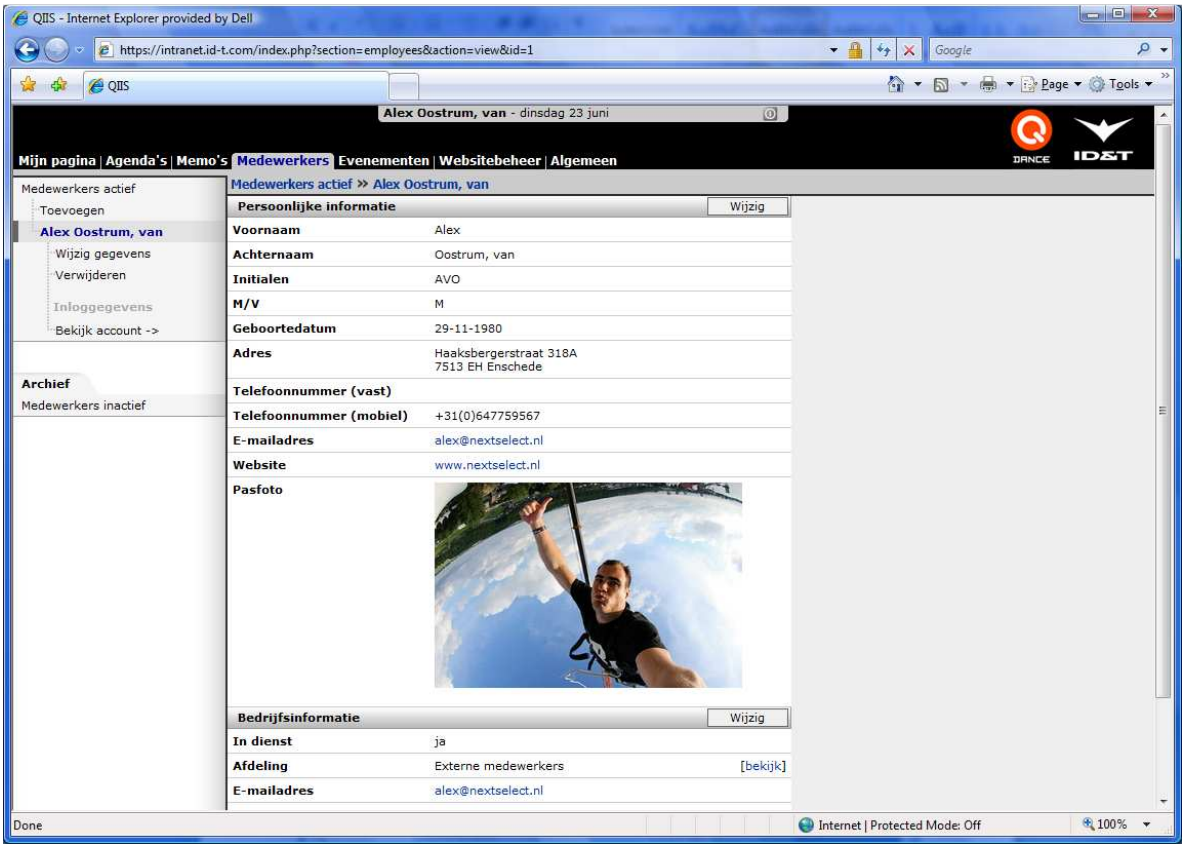


Figure appendix F-2: Q-dance and ID&T Intranet System (QIIS)

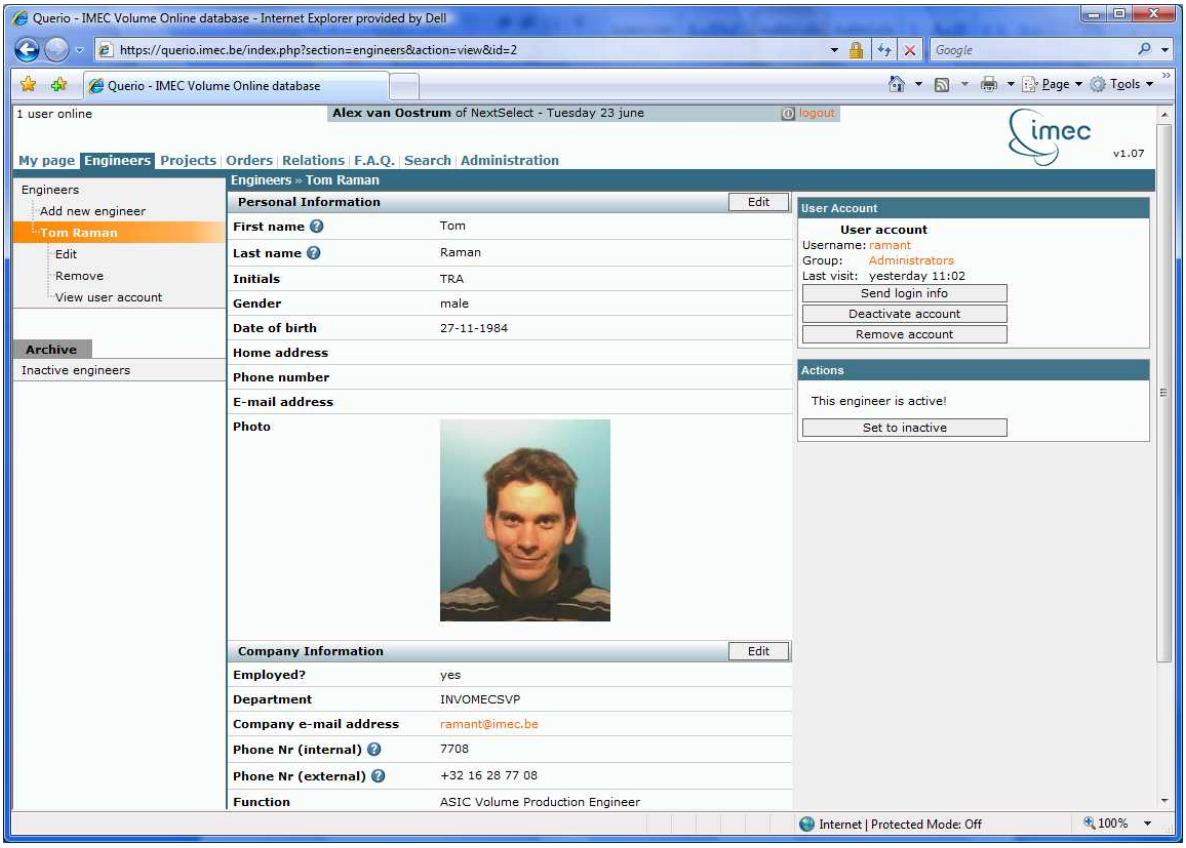


Figure appendix F-3: Querio - IMEC Volume Online database

The sequence of figures also show the order in which the applications were built. The existing framework of MIDS, the first application, was copied to be used for the second application, QIIS. Unfortunately the business logic and the framework were not completely separated so the framework had to be adapted before it could be used for the QIIS application. This resulted in two different versions of the framework. Furthermore, the framework of QIIS was updated and received new features which could not be installed easy in the MIDS framework because of the difference between the two frameworks. To make life more complicated more applications were developed in the same way, like Querio (Figure appendix F-3), taking the latest version of the framework, adapting it and upgrade it for the new application.

Different versions of the framework exist, doing almost the same but they have differences. The new framework can only resolve this if the all of the business logics is placed outside the framework and all the features required by the different applications are implemented. This might mean that a specific application will not utilize the whole framework due to the fact that an application does not need a certain feature. This is not a problem as long as the unused features will not cause unnecessary overhead for the application, i.e. the framework must have some sort of optimization.

Every application (see Figure appendix F-1, Figure appendix F-2 and Figure appendix F-3) is divided into different sections, these sections are a reflection of the business logic. Some sections, like *Employees*, are more general and used in all the applications, although sometimes with a different title. The *Employees* section can be seen as a Human Resource Module (HRM) and this chapter is going to describe what components the framework required to make the module working for the different applications.

F.3. REQUIRED EXTENSIONS AND MODULES

For the described applications (MIDS, QIIS and Querio) the following extensions are required. Please note that these extensions are designed for the kind of applications that NextSelect develops and are not standard in general, e.g. another type of application could require completely different extensions.

The extensions and modules will be briefly handled in this subsection for a global overview and handled in more details later on.

Extension	Description of extension
ConfigurationManager	This is an extension that by default needs to be loaded when the *Framework Core* is instantiated. It tells the framework which extensions also need to be loaded and sets other cross application (NextSelect applications) settings for the framework. Is also requests the framework to load the Configuration module where customer specific settings will be located.
DataLayerMySQL	Manages data storage and retrieval with an MySQL database.
SectionManager	Important for the actual web application to work. The HTTP request of the user will be processed by this extension. If corresponding modules are required this extensions will trigger the framework to load them. Module specific request will be passed on to the module.
DataHandler	Responsible for the generation, modification and representation of data. It generates forms (for adding, editing and removal of data), overview lists and views individual data records.
TemplateManager	Manages templates of layout components. Template files are located outside of the framework. Every customer has its own templates.

Module	Description of module
Configuration	Holds customer specific settings for the application. For example: database type and authorization, application language, template choice.
Employees	The HRM packages that is important for this prototype demonstration. It manages the employees in a simple way (only personal information is stored) . It is able to generate an overview list of al active and inactive employees and add -, modify, and remove employees from the system.

Keep in mind that the extensions are the same for the different application, i.e. the same framework containing the extensions is used for the different applications. Modules contain the business logic and customer specific settings, i.e. for every applications different modules are used.

The following subsections elaborate details of the extensions and modules. Functionality, communications between extensions and modules, and dependencies are specified. A dependency specifies another extension or module that is required for a certain extension or module to operate properly, i.e. if extension A has extension B as dependency it means that A will ask the *Framework Core* to load extension B.

F.4. EXTENSION: CONFIGURATIONMANAGER

The framework and its extensions might need some configuration, most important example: which extensions and modules have to be loaded and initialized when the framework initializes. Instead of putting these framework specific settings in the *Framework Core* or in different extensions it is better to put them in a central place so they can be easily adjusted. Note that some configuration settings have to be put outside this extension; the call to actually load and initialize this extension for example. This is a chicken and egg problem: an extension is not able to load itself.

Another feature of this extension is that it will try to load the *Configuration* module, which holds all the customer specific settings, upon instantiation. This means that every application using this framework prototype will need a *Configuration* Module. Note that the framework itself does not require a *Configuration* module, but the design of this application prototype made it a

requirement. Therefore it is implemented as a dependency within the *ConfigurationManager* extension. Also note that this approach is not compulsorily but a considered choice. In other implementations of the framework this approach could be different.

Dependencies:

- Module: *Configuration*
- Extension: *DataLayerMySQL*
- Exxtension: *SectionManager*
- Extension: *LayoutManager*
- Extension: *TemplateManager*

Note that the *ConfigurationManager* extension tries to load all extensions, except the *DataHandler* extension. This is because the application in every situation requires the use of these extensions. *DataHandler* is not required because this class is responsible for the generation, modification and representation of data, this is simply not always required. For example: when no user is logged on to the application only a login screen is visible and no data has to be modified. Again, this approach is based on a considered choice, in another implementation of the framework this approach could be different.

The constructor of the *ConfigurationManager* loads all the dependent extensions and modules. To load a module a request must be sent to the *Framework Core*. When the request is sent the *Framework Core* will try to load the module or extension and instantiate it. Upon instantiation of a module or extension the constructor of the module/extension is called. If the module or extension has dependencies this is the place to locate them. A dependency is easy implemented; just send a request to the framework to load the dependant module or extension.

Function initialize():
Does nothing, due to the fact that nothing has to be initialized.

Function register_setting(key : string, value : string)
Stores or updates setting identified by the given key with the given value.

Function get_setting(key : string) : string
Returns the value of a setting identified by the given key. If no settings can be found with the given key a NULL value will be returned.

F.5. EXTENSION: DATALAYERMYSQL

For this prototype data storage and retrieval is required. MySQL database has been chosen for this framework. Because this choice is made for the framework all applications have no choice

but to work with the MySQL through the *DataLayerMySQL* extension. For this prototype this is not a problem, for other applications it is considerable to add more than one data storage extension to the framework, e.g. PostgreSQL or Microsoft SQL server.

Because this is a small prototype simulation on paper, this extension will be kept simple. In normal circumstances an abstract data storage and retrieval extension would be better. Next to the abstract extension database specific extensions exist, so for other extensions or modules it is transparent to store and retrieve data without knowing which database is used. Switching from database can be done easy without have to change dependent code in other extensions / modules. Please note that although SQL standard exists not all databases utilize this correctly and some have more functionality then others which may not be adopted in the SQL standard. This could result in having to change code because the queries are not compatible when changing the database. If PHP is used as a programming language the framework 'Propel' could be for example used to solve this problem. Propel is an Object-Relational Mapping (ORM) framework for PHP5. It allows access to databases using a set of objects, providing a simple API for storing and retrieving data [30].

Dependencies:

- Extension: *ConfigurationManager*

Function initialize():
This function will ask for the login credentials and database settings from the *ConfigurationManager* extension. The login credentials are customer specific and therefore located in the *Configuration* module. Upon initialization of the *Configuration* module the module will register its settings with the *ConfigurationManager* extension. The *Configuration* module will be discussed in subsection 7.6.8. After retrieval of the configuration from the *ConfigurationManager* extension, the function *initialize_database_connection()* is called to try to establish a database connection.

Function database_connect() : boolean
Tries to establish a connection to the MySQL server using the login credentials and database settings. Returns true on success and false on failure. Reason for failure will be stored internally.

Function database_close() : boolean
Closes the database connection if a connection to this database exists. Returns true on success and false on failure. Reason for failure will be stored internally.

Function database_query(sql_query : string) boolean
Sends a given SQL query to the database. On success the result set is stored internally and the function will return true, on failure the function will return false. Reason for failure will be stored internally.

Function database_fetch_row() : array
Returns the next row from the current result set in the form of an array. For this prototype it does not return an object, this is done to keep this prototype simple. The function will return false if no result set exists or when no rows are left.

Function database_free_result() : boolean
Will free all memory associated with the current result set. On success the result set is stored internally and the function will return true, on failure the function will return false. Reason for failure will be stored internally.

F.6. EXTENSION: SECTIONMANAGER

Web based applications have a fundamental design principle, they are stateless [7]. This means that a server running a web –based application can never be certain of the current state of the client. In order for the application to work the client (a person with a browser running on a computer) sends a request via the HTTP (or HTTPS) protocol to the server that hosts the web-based application. With this request also comes session data and other information (e.g. form data) which allows the server to determine the state at the time of the request and act accordingly.

The earlier described web applications all have different sections and these sections are divided in subsections etc. Every (sub)section has actions like viewing, modifying and removing of objects. The human resource section (called 'Employee') has no subsections and the following actions:

- Employees (section)
    - overview (action, gives a list of all employees)
    - view (action, views a specific employee)
    - new (action, for adding a new employee to the database)
    - edit (action, to modify an existing employee in the database)
    - remove (action, remove an extisting employee from the database)

The main idea with this prototype is that the business logic is located in the *Employees* module which will be described later on. In this module the sections and actions have to defined telling the framework 'what to do', this means translating the business logic into framework calls. The framework must have support or the requested business logic, if not it should be added to an existing extension or new extension. It is not the philosophy of this framework that a module in general contains more code other than framework requests. In fact, the module should only contain a kind of scripting language that the framework understands, this aspect is discussed in detail in subsection 3.4.6. Due to the fact that this research focuses on the design of the framework instead of the possibility to have kind of scripting language in modules it is not a

problem to implement the module in the same language as the rest of the framework, as long as new features are not implemented in the modules.

In this prototype a module does not load automatically, but even if it did (which is the choice of the programmer that implemented the framework), somebody has to tell it what to do, otherwise the module has to figure it out on its own and this means implementing features that are not business logic which is not preferred. This *SectionManager* extension handles this.

A request from the client browser is specified in a URL (Uniform Resource Locator). URL is a type of Uniform Resource Identifier (URI) that specifies where an identified resource is available and the mechanism for retrieving it. In popular language, a URL is also referred to as a Web address [31]. The syntax looks like this:

```
resource_type://domain:port/filepathname?query_string#anchor
```

An example URL to a web page containing information about the URL:

```
http://en.wikipedia.org/wiki/Uniform_Resource_Locator
```

The following URL has been copied from the MIDS application of MASER Engineering:

```
https://maser.nl/mids/index.php?section=employees&action=edit&id=71
```

The structure of this URL is simple and therefore also used in this prototype. The actual request that the web server receives is located in the last part of the URL:

```
index.php?section=employees&action=edit&id=71
```

The file index.php can be seen as a façade for this web based application, every query has to be sent through this file. The example query above tells the section is 'employees', action is 'edit' and id number of the object is '71'. Index.php can also be seen as the `public static void main()` function in Java and C++, everything has to start somewhere, it will initiate and start-up the framework. The *ConfigurationManager* extension, which will be loaded manually will tell the framework to load this extension. Upon instantiation (i.e. constructor is called) the request of the client gets processed. The framework is told to load a module with a name derived from the request query. The *ConfigurationManager* extension contains a default section and default action that can be used if no section and/or action is specified and a specification what to do when a requested module is not found.

Not only the URL contains data that will feed the web-based application with information. The client's browser will also send extra information with HTTP headers in the request. This may include data for the web-based application like session information or form data, which will also be processed by the *SectionManager.*

In this prototype if all extensions and modules are finished initializing the framework will issue a run command to all the loaded modules, including the dynamically loaded section module. The loaded section module will ask the framework for the current action and only tell the framework 'what to do' for a specific action. It is also possible to let the *SectionManager* extension trigger the module with the action, in this way the module is completely passive and not actually 'running' anymore. The programmer of the framework has to decide how the framework has to handle request. Furthermore, it strongly depends on the kind of application, for non web-based applications, e.g. a Java application, there is no URL request and this could result in a different module/section handling. The second option seems better due to the fact the module itself gets even less non business logic code. But to keep this prototype orderly, simplicity is appreciated, the second option requires more functionality at the extension side. The first option, let the module itself derive which action has to be executed is sufficient and will not break the philosophy of the framework.

When the SectionManager is instantiated the constructor will analyze the incoming request and perform the necessary steps to let parts of the request to the appropriate modules /extensions. To keep the prototype simple this is all of the functionality it will get.

Function initialize():
It is possible that some other initialization have to be done after instantiation. This might be the case that the SectionManager has to wait for certain extensions or modules to be loaded before it can make the request. This is the suitable location to further inspect the request of the client and to take further actions. For this prototype application it was not necessary so there is no functionality located within this function.

F.7. EXTENSION: DATAHANDLER

Most important part of the existing frameworks, used for the different applications, is the so called '*TableHandler*' class. This is a tool that facilitates the management of objects, such as listing, creating, modifying, viewing and removing objects. It generates complex HTML forms with a few calls, saving the programmer a lot of time. The concept for the new framework and the philosophy to only have to 'tell' the framework what to do originated for the most part from this tool. As nice as the idea behind this tool is, the tool itself needs some serious reconstruction. The first version was built in 2002 without the use of good coding standards. When the applications needed updates the T*ableHandler* often needed updates which were implemented for the most part as ugly hacks because of time issues. This finally resulted in a tool that is great in use, but consisting of thousands of lines of (bad) code which gives a headache when maintaining it.

In this prototype a common shared part of all the existing applications will be simulated, the Employees section. To keep the prototype simple only the following actions are permitted:

- List of all employees (action is ‘list’)
- View details of an employee (action is ‘view’)
- Edit the details an employee (action is ‘edit’)
- Create a new employee (action is ‘new’)
- Remove an employee (action is ‘remove’)

The *DataHandler* extension is going to replace the *TableHandler* tool. Of course is it not possible during the scope of this research project to design a new and perfectly working tool, this requires a whole new project on its own. So only the basics to support the actions previously described will be supported.

Dependencies:

- Extension: DataLayerMySQL
- Extension: TemplateManager

Function initialize():
Does nothing, due to the fact that nothing has to be initialized.

Function set_handler(handler):
This function expects a *handler* data type, which is important because the handler data type contains all the information about the object that the *DataHandler* needs to manage. At a low level it tells which table or tables need to be used, which columns of that table need to be used and it knows references to other tables. A handler describes the data structure of an object and all references to and from other objects.

With this function the *DataHandler* receives a *handler* which it can use to manipulate the data. Without a *handler* the *DataHandler* extension is useless because it has no clue what to do.

In this prototype only one *handler* exists, the handler for the employees table. Due to the fact that every customer wants to store different information, even with something simple like an employee object, the handlers are considered as business logic and must therefore be defined in a module.

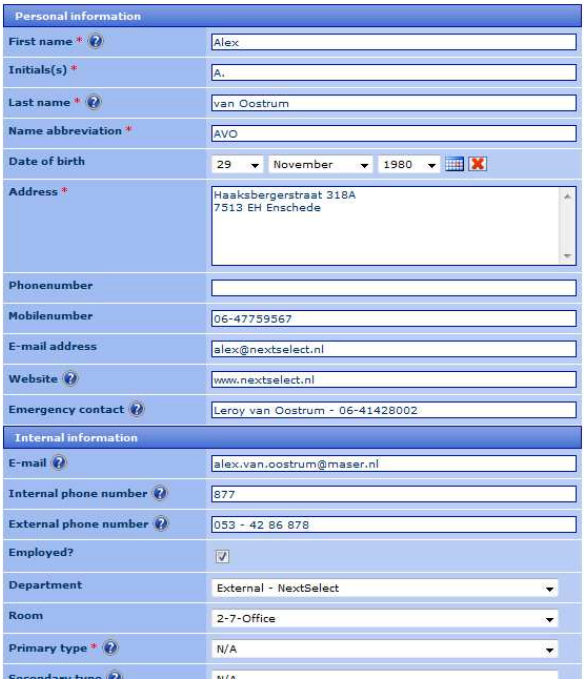


Figure appendix F-4: Example web based form

Function show_form():

Depending on the state of the *DataHandler* this function shows a form to either add a new object or to edit data of an existing object. The handler (discussed previous *in set_handler()* function) forms the basis of the form. It tells which fields are visible and what form types (text field, checkbox, radio buttons, etc) that have to be used for the form. An example of such a form is shown in the figure on the right.

This form is completely automatically generated. Catching the post, checking for errors and storing data is handled by the *DataHandler* extension. So the programmer that implements business logic in a module can focus on the business logic instead of tedious programming issues.

For the HTML representation the *TemplateManager* extension is used. With this extension no HTML or any kind of formatting code has to be used in the DataHandler extension. The *TemplateManager* can be seen as a View of the Module View Control pattern. If instead of HTML XML is preferred this can be done by changing the *TemplateManager*.

Function check_form():

A form that is posted has to be checked before it can be inserted into the database (or any other data storage device). Within the handler can be specified which fields are required and formatting rules. It depends on the type of field what kind of rule can be specified. Also checks if data is valid, like an e-mail address, can be specified. These checks are performed by other extensions. In this prototype they are left aside.

Function submit_form():

When a form is passes the checks by the *check_form()* function the data can be inserted into the database (or any other data storage device). Depending on the state of the *DataHandler* extension it knows if the submitted data is new (e.g. adding a new employee) or if it needs to replace data (e.g. modifying an existing employee).

Function show_list():

Shows al records given a *handler*. In this *handler* are also rules defined for listing the objects, e.g. only show active employees and which columns to show. An example of such a list is depicted in the following figure:


Filter on: First name Filter value: Filter Reset Filter

Create new employee

First name ▼	Last name	Department	Room	Internal phone number	External phone number	Employed?
Alex	van Oostrum	External - NextSelect	2-7-Office	877	053 - 42 86 878	yes
Annemiek	Nijhuis	Secretary	1-13-Office	697		yes
Astrid	de Vlugt	Secretary	1-13-Office	680		yes
Christian	Bos	Physical Analysis	1-10-Office	844		yes
Clemens	Wargers	Physical Analysis	1-11-Office	693		yes
Daan	Scheerens	External - NextSelect	2-6-Office	878	053 - 42 86 878	yes
Dagmar	Mebius	Reliability Test	3-10-Office	843		yes
Daniël	Arslan	Physical Analysis	-	851	053 - 428 68 51	yes
Danielle	Hoogstede	Physical Analysis	2-9-Office	699		yes
Dennis	Ticheler	Reliability Test	3-10-Office	800		yes
Edwin	Obdeijn	Reliability Test	4-6-Office	679		yes
Edwin	Jellema	Reliability Test	4-1-Office			yes
Erik	Heijt	Reliability Test	3-10-Office	678		yes
Ewald	Reinders	Physical Analysis	1-11-Office	687		yes
Ferdi	Meijer	Physical Analysis	2-7-Office	686		yes
Fred	ter Borg	Reliability Test	3-11-Office	848	053 - 42 86 848	yes
Gerard	Klomphaar	Reliability Test	-			yes
Gilbert	Rikkink	Physical Analysis	1-11-Office	685		yes
Hans	Kemper	Management	1-14-Office	682		yes
Hendrik-Jan	Mutsaers	Physical Analysis	2-10-Office	846		yes
Jolien	Nijhuis	Secretary	1-13-Office	680		yes

Figure appendix F-5: Example listing

In the figure above also a ‘filter’ (at the top) is shown. This extra functionality that can be added to the *DataHandler* extension. In this prototype this is left side.

Function show_object:

The last actions that will be demonstrated with this prototype is the viewing of a specific object. Again with the given a *handler* a specific record will be retrieved from the database and the data defined in the *handler* will be shown on the screen of the user.

Other functions:

Of course this extension requires more functions than the described above. The less important are intentional left side from this document to keep the focus on the important aspects clear.

F.8. EXTENSION: TEMPLATEMANAGER

Using templates for a web-bases application might be handy. If templates are used it means that screen data goes through the template. A specific piece of code that wants to output something on the screen of the user does not have to take care or know anything of the corresponding HTML code. The advantage of using templates is that application logic (not business logic) and content are being separated from presentation.

Because this prototype does not require a HTML framework with navigation panels, shown in figures Figure appendix F-1, Figure appendix F-2 and Figure appendix F-3 on the left side, it has limited functionality.

Templates are stored in files and are part of the business logic domain, due to the fact that every customer might want to have its own style , layout, etc. The *TemplateManager* extension needs to know where it can find the template files. The best location for this specification is in the *Configuration* module. If this location is always the same it can also be set in the *ConfigurationManager* extension. In any case this extension will retrieve the location of the templates through the *Configuration* extension.

Dependencies:

- Extension: Configuration

Function initialize():
Retrieves the location of the template files from the *ConfigurationManager* extension. This location could be specified in the *Configuration* module, when this module is loaded it stores al its configuration in the *ConfigurationManager* extension (see subsection 7.6.4 for more details about the configuration).

Function display_template(template_name, template_variables):

Generates HTML output based on an existing template and passed on template variables. The template file is a simple PHP script that contains HTML. Within this HTML variables are placed that can be filled with content defined within *templace_variables*, this is a data type that enables the programmer to assign a set of variables that contain application logic output that needs to placed within the template. In the following diagrams a simple template example is given. On the left the actual template file with two variables. On the right is the generated output when the two variables are filled with application content.

Template file:

```
<html>
<head>
	<title>User Info</title>
</head>
<body>
	<p>User Information:</p>
	Name: {$name}<br />
	Address: {$address}<br />
</body>
</html>
```

Output to screen:

```
<html>
<head>
	<title>User Info</title>
</head>
<body>
	<p>User Information:</p>
	Name: Alex van Oostrum<br />
	Address: Haaksbergerstraat, 318A, Enschede <br />
</body>
</html>
```

The advantage of using templates it that application logic and presentation are clearly separated. If this implementation method is applied correctly it resolves one of the problems with the current software built by NextSelect and MASER Engineering, discussed in section 2.1.

The location of the templates has to be retrieved from the *ConfigurationManager* extension. The name of the template that is given through the *template_name* variable, represents the filename of the template without extension.

F.9. MODULE: CONFIGURATION

This module holds the customer specific configuration. This can be the database connection settings but also the language settings of the application. Every setting that can be adjusted conform customer specifications has to be located here.

In this prototype the *Configuration* module will be loaded by the *Configuration* extension (this extension is always loaded, see subsection 7.6.4 for more information). There is a implementation difference with the *Configuration* module and the other extensions. Due to the fact that upon loading of the extension/module it is unsure if the depended modules/extensions are loaded, initialization of a module or extension takes place in the *initialization()* function and not in the constructor. This *Configuration* module forms an exception to this rule because some settings located here may be required when an extension/module is initialized, i.e. before the actual initialization call. When the configuration settings are placed in the *initialization()* function

it is not guaranteed that the initialization of this module is called before the initialization of an extension/module that needs a setting. Therefore the settings are placed in the constructor, ensuring that the settings are initialized upon loading of the module and thus ensuring the settings are available when modules/extensions are initializing.

Furthermore, the *Configuration* module depends on the *ConfigurationManager* because the *ConfigurationManager* stores all the settings. Initialization of a module or extension, like this *Configuration* module, in the constructor is normally not advised because it is unsure if depended modules or extensions are already loaded, e.g. the *Configuration* module uses functionality of the *ConfigurationManager* without knowing if the ConfigurationManager actually exists.The *Configuration* module also has an exception to this rule, due to the fact it is known (actually programmed in the source code) that the *ConfigurationManager* extension loads the *Configration* module, it is therefore ensured that *ConfigurationManager* is loaded.

Dependencies:

Extension: *ConfigurationManager*

Module: Employees

This is the location for the business logic and this is also the location where the advantage of the framework becomes clear. The actions previously discussed in subsection 7.6.6 are 'implemented' here. The implementation of these actions are in fact very simple; depending on the action the module will issue a request to the framework that will look like this:

```
$employeeTestFramework->show_form();
```

By this request the *Framework Core* would issue the *DataHandler* to execute *show_form()* which will draw a form to add or edit an existing employee. For all the other actions similar actions have to be taken (described later on). The clear advantage here is that implementing a module does not take a lot of effort. Of course the handler needs to be defined which tells the *DataHandler* the elements of an employee object, e.g. name, address telephone number etc., and a real application some more requests might have to be made. But the advantage should be clear: implementing a module is simple and should not take a lot of time. This depends however on the actual implementation of the framework , if this implementation is according to the philosophy it should not be an issue.

Function initialize():
Here a *handler* object is constructed that contains all the information about the employee object that needs to be handled by the *DataHandler* extension, e.g. a name, address, telephone number and department. It feeds the *DataHandler* extension with information on how to present or modify a certain object, in this case an employee.

Function execute():

After the Framework Core is finished issuing initialization calls to all the modules and extensions, it will call the *execute()* function of all loaded modules. In this function is de code located where the Employee module will tell the framework 'what to do' and when to do it. The advantage of using the framework can be found in the following source code example:

```
$employeeTestFramework->set_handler($employeeHandler);

switch ($current_action):
	case 'list': $employeeTestFramework->show_list(); break;
	case 'view': $employeeTestFramework->show_object(); break;
	case 'edit': $employeeTestFramework->show_form('edit'); break;
	case 'new': $employeeTestFramework->show_form('new'); break;

	// Just an example, remove_object is not actually implemented in this framework
	case 'remove': $employeeTestFramework->remove_object(); break;
}
```

Telling the framework what to do in a few simple steps means that that the framework itself contains all the functionality required to actually do what it is told to do. This is important otherwise functionality could be located within the modules which takes down the philosophy of the framework and, also important, it decreases code reusability due to the fact that an implemented feature within a module cannot be used for other customers.

F.10. UML REPRESENTATION OF PROTOTYPE

To make the prototype overview clear it is represented in the following UML diagrams. The first diagram represents the basic framework design as shown in Figure appendix D-9. It is exactly the same diagram, only the components are placed in a more efficient way to save space.

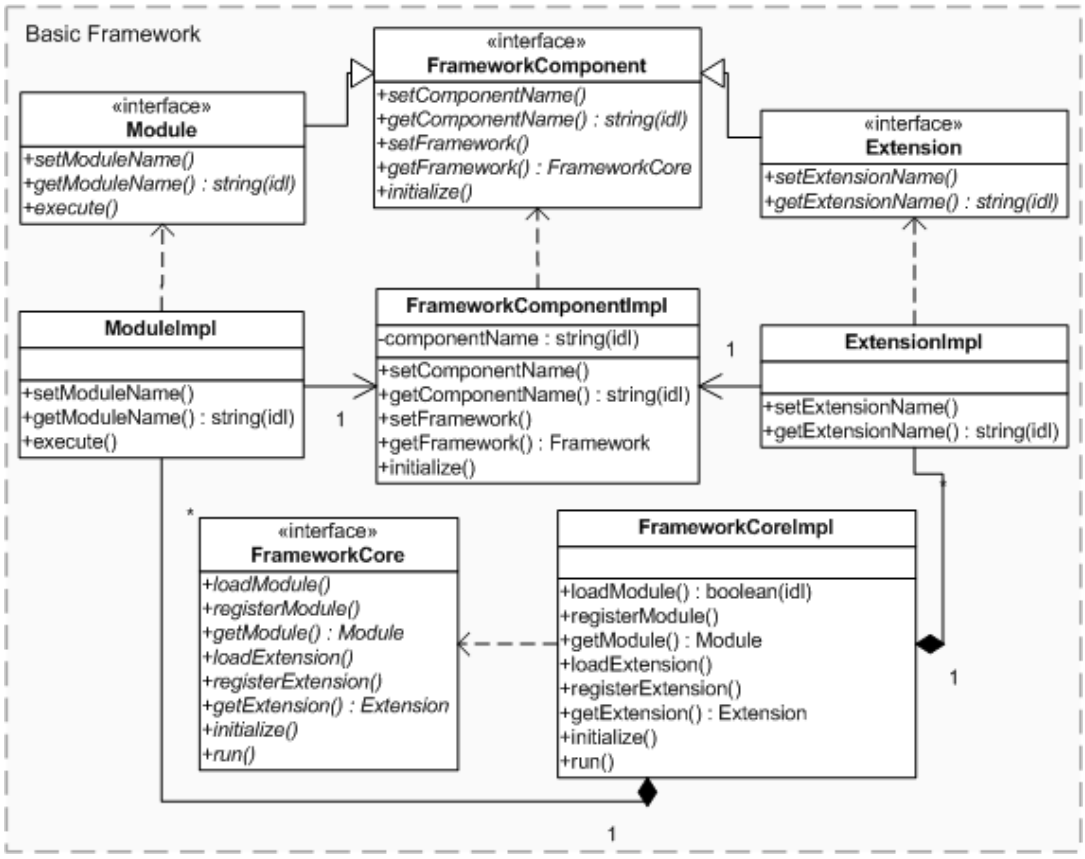


Figure appendix F-6: Basic framework design in UML. It is a copy of the UML diagram presented in Figure appendix D-9, only the components are placed in a compact layout to save space in the document and following UML diagram.

The bordered box represents the basic framework. In the following diagram the prototype is represented, the bordered box still represents the basic framework but only the connected components are shown to save space and keep the diagram orderly.

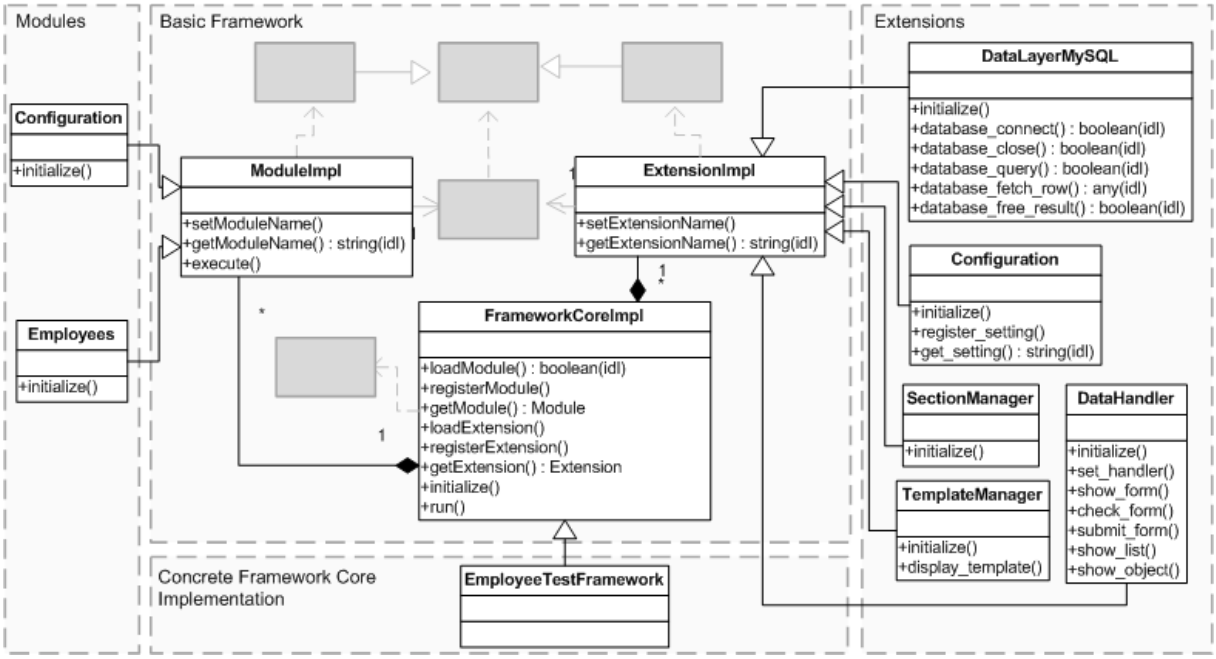


Figure appendix F-7: Complete UML diagram of application with framework

F.11. INTERNAL COMMUNICATIONS

At this moment the basic framework is a design of components, what is still missing are specifications or rules how the internal components (modules and extensions) communicate with each other. This can be resolved in different ways. Some of these have been discussed already within this document. In the first prototypes a direct approach was used, the Framework Core receives a request for a certain extension or module and by looking in its internal list of extensions and modules it returned a pointer to a specific extension or module. The module or extension requesting the pointer had direct access to the requested module or extension.

As mentioned before, the philosophy of the framework is that programming within modules is limited. The business logic of the customer has to be translated into 'simple' calls for the framework to execute with the purpose of saving time when a new application has to be built. So a module contains sort of a script that configures the framework instead of real functionality. This means that all functionality is located within the extensions. Not only modules have to communicate with extensions through the Framework Core , extensions also have to communicate with each other, due to the fact that code reusability has to be maximized (see section 2.3) functionality required in one extension might exist in another extension and must therefore be reused.

As good as the philosophy of the framework might be, there is a problem; the basic design of the framework does not state anything about the implementation of the modules, extensions and the way they have to communicate with each other. This means that the philosophy is not guaranteed by the basic design (illustrated in Figure appendix F-7). The matter is complicated due to the fact that different methods are possible to ensure the philosophy . To resolve this problem the basic design needs to be accompanied by a set of rules. If the programmer follows the basic design and the specifications of the rules when implementing the framework, the implemented framework complies to the philosophy.

At this point the rules are still unclear. In the first prototypes a very simple method was used by means of pointer (to module or extension) pass on. In this prototype another approach is used and this should give more insights on how the rules must drawn. Implementation of this new approach is simple. The *Framework Core* holds a simple data structure, e.g. an array, which has a reference to all the functionality of the extensions and objects. When a component (module or extension) is loaded and instantiated, the Framework Core issues a request to component for all its functionality. This is standard functionality provided by the PHP engine by use of the *get_class_methods()* function [34].

When all the components are loaded and instantiated the *Framework Core* has a list of all components and their functionality. The *Framework Core* gets a special 'magical' function *__call()* (illustrated in Figure appendix F-8) which is triggered when invoking inaccessible methods in an object context [33]. This means that when an component asks functionality of the

*Framework Core, w*hich is not actually located within the *Framework Core,* the request is automatically passed on to the *__call()* function.

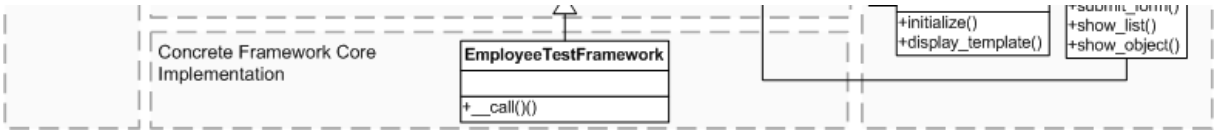


Figure appendix F-8: Added __call() to the concrete Framework Core

To keep the prototype simple the *__call()* function does not have a lot of intelligence. It will look up the requested functionality within the internal function list and pass the request with optional parameters to the concerning component. In this way modules never have direct contact with the extension that the request is passed on.

This approach limits functionality for components because all communication travels through the Framework Core. For modules this is perfect because it follows the philosophy how the framework should work. For extensions however it might be another case. First of all, using the approach of only asking the framework functionality instead of direct communication with the components creates overhead; the appropriate component has to be located first and then the optional parameters has to be passed through. For a module this is not a problem because it is likely that the amount of request is limited, configuring and telling the framework what to do takes between 20 to 50 calls depending on the request. This is based by looking at the existing framework and counting the requests as if they would be implemented in this framework. For extensions this is a completely different story, only to generate a simple form to edit employee data could easily generate thousands if not tens of thousands requests. One might ask if it is necessary for the extensions to communicate with the overhead that might slow down the application considerably. Due to this fact it is allowed for extensions to have direct communication with other extensions. They can request a pointer from the *Framework Core* and store it in their own data structure, just like in the first prototypes. The best point to request the pointer of another extension is when the *initialization()* function is called. At that point all dependencies have be loaded and instantiated, and the requesting extension can store the pointer so it does not have to ask for it twice.

There is not much intelligence in the *__call()* function that is located in the *Framework Core* class. It is rather easy to make it more intelligent by using the *ExtensionDataType* to search for specific functionality with specific input or output parameters. This is left outside this prototype due to time issues and the fact that making the internal communication more intelligent does not mean that the philosophy is better implemented. Furthermore, the function get_class_methods() that returns a list of all functionality within a class only returns function names, not the parameters and return values. This means that another method for registering functionality with the *Framework Core* has to be implemented. This also holds for programming

languages that have no support to list class functionality. The *ExtensionDataType* object is further elaborated in section 4.3.

F.12. DISCUSSION

The development of this prototype needed to show a clear advantage of using the framework and that it actually will work for MASER and NextSelect. First, let's look back to the main problems of the existing framework; business logic and framework core components are not separated, resulting in a framework that has customer specific implementations. When enterprise applications had to be built for new customers the existing framework was copied for every new application and the customer specific implementations within the framework were replaced or in some cases reused. The copied existing framework that needed to be used for another customer actually became a new version of the framework. Due to the fact that the business logic contained within the existing framework needed to be replaced it was not compatible with its predecessor framework and thus making the new version a different version.

As time progressed more enterprise applications were built resulting in more different versions of the frameworks, frameworks that by essence are the same but due to customer specific implementations different. Maintaining the software becomes harder and harder, e.g. if a bug is found in one of the existing frameworks changes are it also exists in the other frameworks. The questions is: do the other frameworks have the same bug, does the bug need to fixed and can it be fixed with the same solution?

The architecture of the new framework demands that business logic and framework components are separated. In this prototype this is achieved by only implementing business logic in modules, i.e. the *Configuration* and *Employee* module. All the functionality to facilitate the modules is located in the framework extensions, which do not contain any business logic. This prototype framework is simple, it has only functionality to do some operations, like adding a new employee or modifying an existing employee. Now suppose a new application has to be build with this framework for a different customer. The customer in question might want to store different or more data of an employee. To achieve this the same framework can be used without having to change anything of the prototype framework. Only the business logic of this new customer has to be implemented resulting in a new *Configuration* and *Employee* module. The framework itself stays the same and is even interchangeable with the previous application.

The *Framework Core* and the functionality located within the extensions determine for what kind of applications the framework can be deployed. This prototype framework was built for simple employee management applications only, but it has been built in such a way that it could be deployed for different customers with different business logic. This is achieved by using the handler data type which gives the programmer of the application the ability to customize an employee object and thus making it possible to implement a new application for another

customer that wants to handle different employee data. In this case a new handler has to be implemented in the *Employee* module, describing the employee of the new customer. This is exactly what MASER/NextSelect need. The fact that their required framework is a lot bigger and required a lot more extensions is irrelevant because the idea stays the same.

When a new framework has to be built, something that will happen for MASER/NextSelect, the design and implementation have to take in consideration that the extensions are tools. Tools to facilitate the needs of modules and other extensions. As stated before, the implementation of the framework and its extensions will determine the scope of applications it can be deployed for, e.g. if NextSelect finishes the implementation of a new framework for development of business applications it cannot be deployed for, for example, spread sheet applications. The framework has to be deployable for different customers, so the extensions and the core must absolutely contain no business logic. Otherwise the framework roles back into the old situation where customers had their own custom version of the framework. An extension might be expanded with new functionality or new extensions might be added when a (new) customers asks for new functionality resulting in a new version of the framework. The new version must be deployable for other existing customers, i.e. their framework gets an upgrade, even if they are not going to use the functionality. If this rule is applied properly all customers keep working with the same framework and only one framework has to be maintained.

Please note that it is possible for a customer to request functionality that no other customer is going to use. The added functionality in either an existing extension or a new extension might look as if a part of the business logic is implemented in the framework, due to the fact that it seems a custom implementation for a specific customer. This is not true, a long as the added functionality is implemented as a tool showing no business logic of the customer. Even if it is going to be used only by one customer it needs to be implemented as if it could be used for more customers. So, like the employee example, the added functionality has to receive the business logic before it can facilitate the application. This is keeping the business logic and framework core components separated and thus ensuring that the framework remains deployable for different customers and their applications.

F.12.1. RESEARCH QUESTION QF-1

> ***The philosophy of the framework is that modules are not programmed but contain only a set of instructions (i.e. script) for the framework to execute. This could be in a simple scripting language but also in some form of natural language that the framework is able to interpreted. If this is not possible it is allowed to use the same programming language for the modules as the rest of the framework, what is allowed to implement and what is not ?***

As extensions are implemented as tools, modules are not suppose to have actual implementations according to the philosophy of the framework. A module should be like a script telling the framework what to do in a certain point in time. In the prototype the modules were implemented in the same programming language as the rest of the framework. Using the programming language is allowed because the module can still be implemented as a script even though the script is in the same programming language. Important here is to make sure no functionality for the application is added to the module. Statements like 'if', 'else', 'switch' are allowed, even the implementation of functions are allowed. As long as no new functionality is added the programmer is free to use the tools the programming language provides.

An implementation in a module can be seen as new functionality if the implementation tries to manipulate data. The implementation that offers the functionality for data manipulation needs to be located in an extension not a module.

A programmer might not see the problem of adding functionality to a module, if the application works it is fine. The problem with adding functionality to a module is the fact that it will tear down the objective of maximizing code reusability. If functionality is implemented in a module it means that it will only be available for one customer/application. If another customer requires the same functionality it has to be implemented over again, resulting in two likewise implementations. This is in conflict with the philosophy of the framework.

F.12.2. RESEARCH QUESTION QF-2

> ***How are modules and extensions going to communicate with each other?***

Modules and in particular extensions have to register their functionality with the *Framework Core*. Following the philosophy of the framework modules will not communicate directly with extensions. They have to send the request through the framework, i.e. telling the framework what to do. The *Framework Core* has to figure out how the request has to be handled and delivered to the appropriate extensions. How this is implemented in the *Framework Core* depends on the implementation of the framework and what the programmer expects of the framework. In this prototype an internal data structure containing all the functionality and reference to the extensions and modules was used. When a request of a module is received the *Framework Core* looks up in which extension the functionality can be found.

In a more advanced implementation of the *Framework Core* it might be possible, for example, that functionality is not directly asked by name, like in this prototype, but only by input and out parameters. Using only the *ExtensionDataType* object as input and output parameters, it is relatively easy to search for functionality; the *Framework Core* iterates through its own internal repository for an extension that offers a function with the same input and output *ExtentionDataType* parameters. In the framework that NextSelect/MASER are going to use this

is not really desired due to the fact that it is not sure which extension and functionality are going to be used and this can give unpredictable results. However, in another type of application with a different framework it might be desirable to let the *Framework Core* search for the best functionality. For example: a framework that is using web services that are located on the internet as extensions. A module requesting functionally will trigger the *Framework Core* to find an appropriate extension that offers the functionality. Because the extensions are implemented as web services it is not guaranteed that they are always available, e.g. an extensions might be unreachable because they could be offline so the *Framework Core* has to try to locate an extension that is online with the requested functionality.

Another example is that modules are built with some kind of scripting language that the *Framework Core* is able to interpret. This is actually the ultimate goal of the framework philosophy, to provide a framework that is able to be told what to do in a natural scripting language. The interpreter could be located in the *Framework Core,* a better solution is to add an extension providing functionality to interpret a particular language. This allows to add more language interpreters if required and keeps the *Framework Core* clean.

For the extension it is a different story. Just by looking at the functionality of the existing frameworks of MASER/NextSelect it is clear that rebuilding a new framework conform specifications of the new framework design will generate a lot of extensions. These extensions not only have to facilitate the modules, but also other extensions. In this prototype the *DataHandler* extension uses the *TemplateManager* extension to generate HTML forms and the *DataLayerMySQL* extension for retrieval and/or storage of employee data. Extensions, like the *DataHandler* extension, are able to generate thousands, if not tens of thousands requests to other extensions just to handle one request of a module. If extensions have to ask the *Framework Core* for a functionality, just like the modules have to do, it creates a lot of overhead that is not necessary. It is therefore allowed that extensions have direct communication with other extensions without having to communicate through the *Framework Core*. Extensions can still use the *Framework Core* to find other extensions, check dependencies, or even finding functionality like it could do for a module.

To conclude, the actual communication between modules and extensions depends on the implementation of the framework. Module have to communicate though the *Framework Core* for requesting functionality, this can be done in the same programming language or in a more advanced (scripting) language that the *Framework Core* is able to interpret by itself or by using interpret extensions. Extensions are allowed to communicate directly to prevent overhead.

F.12.3. RESEARCH QUESTION QF-3

> ***QF-3: A this point nothing is said about the possibility of multiple instances of the same extension. In some circumstances this might be necessary. Is it necessary and more important: it possible?***

Multiple instances of an extension, in this and the previous prototypes it seems completely useless to make multiple instances of an extension. Due to the fact that an extension has to be seen as a tool that offers functionality. Having multiple instances of the same tool does not give an advantage. Furthermore, the way the *Framework Core* in this prototype communicates the request of modules to the extensions makes it impossible to utilize all the extensions, the core will only use the first it will find.

There is, however, a situation thinkable when multiple instances are useful. Consider the previous discussed example (in A7-5) where multiple web services are used as extensions. Every web service extension could hold the same functionality and it up to the Framework Core to decide which one to use. This could be based on a simple load-balancing algorithm or on performance statistics based on previous requests. In any case, multiple instances are possible.

Although it seems unlikely that an enterprise application developed by MASER/NextSelect will have multiple instances of the same extension, it is allowed.

An important rule here is that the extensions may not store any business logic data with the purpose to use it in the future if it is possible to have multiple instances of the extension. If an extension exists in the application with multiple instances and it contains business logic data it is not known which extension hold which data and this can give unpredictable results and is therefore not desirable.

F.12.4. RESEARCH QUESTION QF-4

> ***A module or extension has two initializations points; the first upon loading, the constructor of the module/extension is called. The second initialization is when the Framework Core calls the initialization() function of the module/extension. What are the rules of placing initialization code in either the constructor or the initialization function of a module or extension?***

A module or extensions has two initializations points. The first, because it is an object (from the object oriented paradigm), is the constructor of the object. When the module or extension gets instantiated the constructor is automatically called. The second initialization point is when the *Framework Core* has finished loading all modules and extensions, after this the Framework Core will call for all modules and extensions the *initialization()* function.

An module or extension might depend on another module or extension for initialization. But at constructor initialization, the first point, it is not guaranteed that the dependency is already loaded. This means that calls to the framework might not be executed, simply because the functionality is not loaded. At the second initialization point this is guaranteed.

The programmer of the framework (and enterprise application) has to take the above into consideration when implementing the framework components. Requesting the *Framework Core* to load a module or extension is best placed at constructor initialization (the first), due to the fact that this ensures all required modules and extensions of the application are loaded before *the Framework Core* will start the second initialization.

Calls to other extensions, requesting functionality or 'telling' the framework what to do, is best placed in or after the second initialization point, due to the fact that the module or extension placing the request is ensured that the dependency is loaded and available. There is however an exception to this rule, one that is also demonstrated in this prototype framework. In some cases, like the *Configuration* module (see subsection 7.6.8), asking for functionality is required at constructor initialization. This is allowed as long as the programmer can ensure that the dependency is loaded at the time the constructor is called. In the *Configuration* module example this is true, due to the fact that the dependency required for the *Configuration* module is actually responsible for loading the *Configuration* module. If the dependency loads the module, the module is ensured that the dependency is available. Because this sequence is programmed it is also ensured that this sequence will not change. As long as the programmer can ensure that the dependencies are met it is allowed to request functionality at constructor initialization.

To illustrate the above example, the following sequences needs to be implemented:

- Framework Core -> load extension [A]
- [A] -> load module [B]
- [B] -> requests [A]

Because this is implemented in the code the programmer knows that extension [A] exists at the time module [B] is loaded, due to the fact that [A] loads [B]. It becomes different when modules or extensions are loaded dynamically or the sequence of loading is not ensured. The programmer of the framework has to deal with this issue and take into considerations at what time a certain dependency is required and if the dependency can be guaranteed.

Another situation that should be avoided is when two extensions depend on each other during the second initialization point to initialize. For extension A to be able to initialize it requires functionality offered by extension B. Extension B in turn requires functionality offered by extension A to be able to initialize. The functionality of the two extensions will only work if the extension is initialized. If this occurs the application cannot be executed and will stop because of

an dependency problem. This can be seen as a programming and design flaw. It is not a problem for modules, due to the fact the modules are not allowed to contain functionality.

F.12.5. RESEARCH QUESTIONS QF-5

> ***QF-5: Is it possible to load modules or extensions after the initialization point of the Framework Core when circumstances demand it ?***

At this point all the extensions and modules are loaded before the *Framework Core* calls all the *initialization()* functions of all the loaded modules and extensions. This is done to ensure that the dependencies are loaded when a module or extensions starts its second initialization point. What if the application wants to load an extension or module in a later moment in time, when the application is running?

For a web based application, like this prototype built in PHP, this most probably will never occur, due to the fact that the actual application does not keep running. The user triggers a request through its web browser which in turn sends the request to the web server that runs the prototype application. Within the server a new PHP process is started which will run the prototype application. Compared to normal application processes the process keeps running until the user closes the application, with this web based application the process is finished when the request output is sent back to the client's web browser. This means that the prototype is started and closed immediately, this also has to be done as fast as possible because the user does not want to wait too long for his request. All the NextSelect applications share this web based model. It seems unlikely that after all the required modules and extensions upon initialization of the application are loaded, another extension or module has to be loaded during execution of the application.

For applications that keep running it is highly likely that the modules and extensions that are loaded at runtime and not only at the initialization points of the application. The concept of the PHP web based application is that the request is analyzed and the required extensions and modules are loaded. Extensions and modules that are not required are not loaded. There is no reason to load another extension or module during runtime of the application when the runtime period is instant compared to a normal application. Because a normal application keeps on running it is not clear at the initialization point which modules and extensions are going to be used during the course of the application. One option is to load them all but this is not very efficient, especially when the framework is relatively large, it will consume memory and it has a negative effect on the framework's performance. If the framework has a lot of extensions and a lot of modules exist it is better to load the modules and extensions dynamically at runtime. This means that when a request is sent to the framework the Framework Core has to check if the

appropriate module or extension is already loaded, if not it will try to load it. After a certain point of time the module or extension can be unloaded to free memory and other resources.

Dynamically loading modules or extensions at runtime means that the *Framework Core* has to know the functionality of the modules or extensions before they are loaded. The programmer has to take into account that the *Framework Core* has to know functionality of all the modules and extensions in order to use them. One method is to load all modules and extensions when the application is started so that they can register their functionality with the *Framework Core*, just like in this prototype except without calling the second initialization point. The modules and extensions not directly required can be unloaded to free memory and resources.

It is the choice of the programmer to load modules or extensions at runtime of the application. The programmer has to take in consideration what the best solution for the framework will be. This depends on the type of application where the framework is deployed for, the number of modules and extensions, and if optimizations are required. Also note that when a module or extension is loaded at runtime the *Framework Core* will not automatically call the second initialization point. This either has to be done manually or the *Framework Core* has to be adapted.