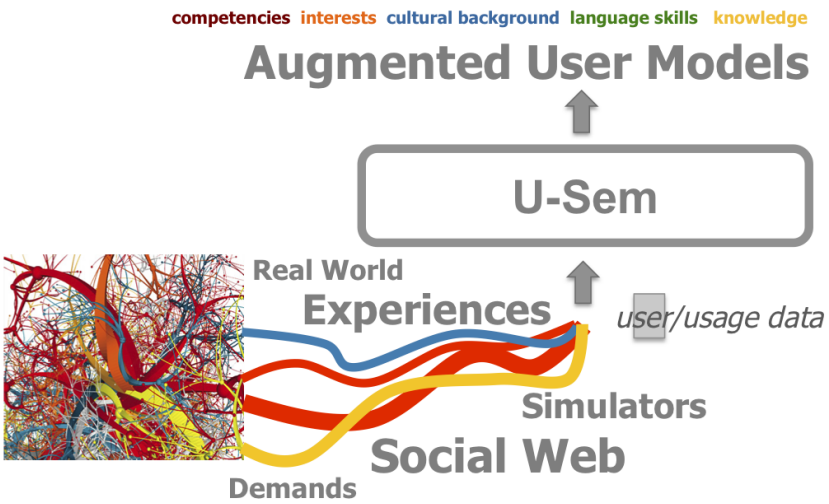
U-Sem, a framework for augmented user and context modelling

Master's Thesis



Borislav Todorov

U-Sem, a framework for augmented user and context modelling

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Borislav Todorov
born in BIRTHPLACE

Web Information Systems
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
http://wis.ewi.tudelft.nl

U-Sem, a framework for augmented user and context modelling

Author: Borislav Todorov
Student id: 123456789
Email: any@email.com

Abstract

This document describes the standard thesis style for the Software Engineering department at Delft University of Technology. The document and it's source are an example of the use of the standard LaTeX style file. In addition the final appendix to this document contains a number of requirements and guidelines for writing a Software Engineering MSc thesis.

Your thesis should either employ this style or follow it closely.

Thesis Committee:

Chair:	Prof. dr. ir. A.B.C. Een, Faculty EEMCS, TUDelft
University supervisor:	Dr. ir. A.B.C. Twee, Faculty EEMCS, TUDelft
Committee Member:	Ir. A.B.C. Drie, Faculty EEMCS, TUDelft

Preface

A place to put some remarks of a personal nature.

Borislav Todorov
Delft, the Netherlands
February 17, 2013

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Research questions

The main research question is the following:

How to facilitate the work of scientists and organizations interested in user modeling and analysis?

The following sub-questions articulate the problem:

1. **How to enable users to add custom functionality and change system's behaviour?** (plug-ins)
2. **Is it possible to develop a component that enables users to store and retrieve information in arbitrary data formats?** (Universal datastore)
3. **How to enable users to maintain their results up-to-date?** (Scheduling)
4. **How to enable multiple users to use the system simultaneously without interfering with each other and keeping their work and results protected?** (Multi User and privacy)
5. **How can real-world use cases benefit from this system?**

1.1.2 Contributions

1.1.3 Organization of the work

There are many possible ways to approach that. One can use the waterflow model, incremental, ...

In this work we will use the incremental model because of the many advantages it brings. We design and implement one feature at a time, providing a working system after each iteration.

The last thing that has to be considered before launching the next phases of the project is to decide on the order in which the features will be implemented. This issue is addressed in the next section.

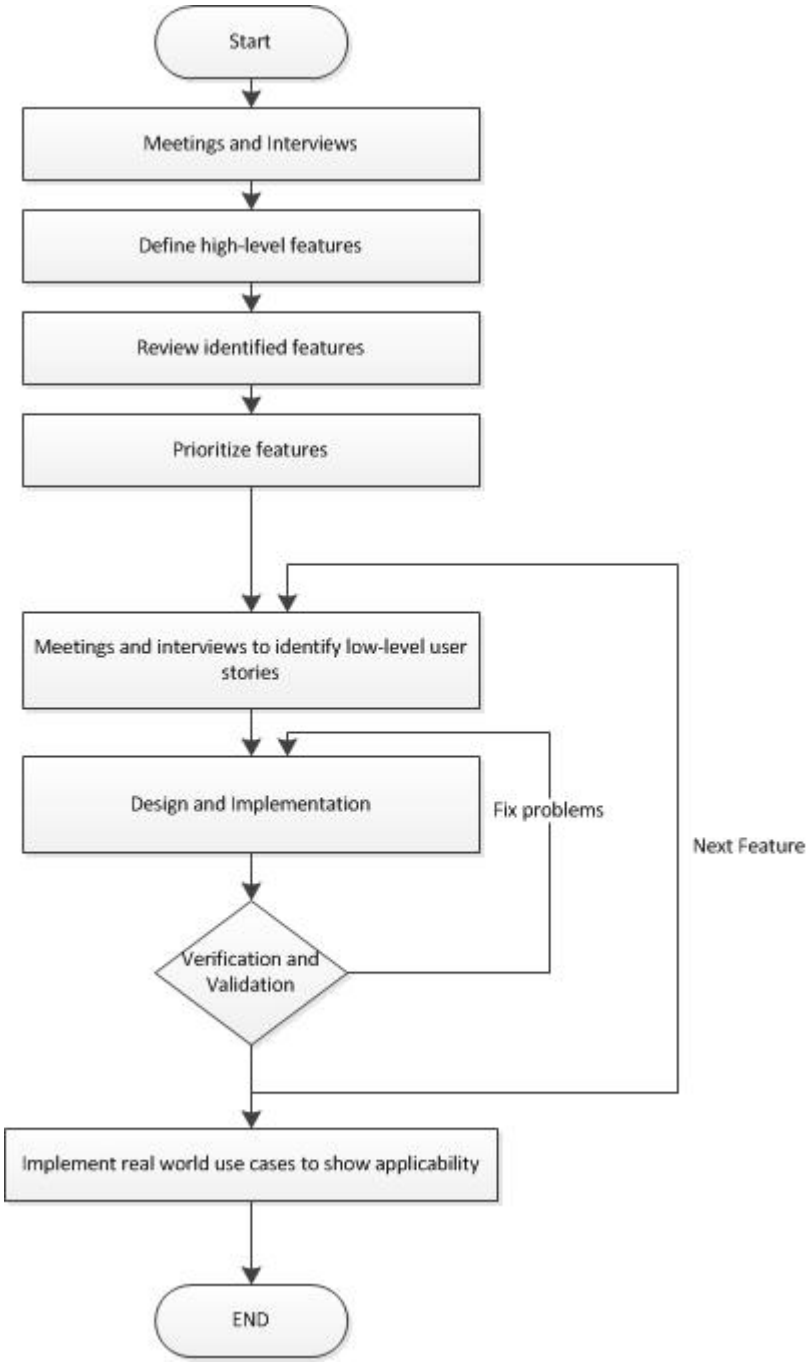


Figure 1.1: Flow chart defining the sequence of actions taken in order to complete the project.

1.1.4 Organization of the thesis

Chapter 2

System Requirements

This chapter describes the process of identifying the main needs of the stakeholders and defining the scope of the project. **TODO**

2.1 Requirements gathering

In this section we aim to elicit the needs of the users and structure them into high-level system features that will later be designed and implemented.

2.1.1 Stakeholders

Stakeholders are the people that are have some kind interest in the project. They are the ones that will be affected by the project and thus they are the people that will be the source for the characteristics of the system we have to build. We identified the following as the main stakeholders of the project:

- Scientists - people that develop algorithms and approaches for user modeling and analysis and allow other scientits and users to access them.
- ImReal - system that want to use the services provided by the scientists and has its own requirements
- RDFGears - provide the workflow engine and want to reuse some of the things.

2.1.2 Interviews

Once we had identified the stakeholders of the project we started to think how to extract the requirements from them. Literature suggest a wide variety of possible approaches []. However, there is one technique that proves to be really effective and it is the semi structured interviews [1]. In order to perform it we prepared some important questions that we wanted to know and then we did discussions **TODO**

2.1.3 Identified features

We analysed carefully all the raw information that we gather from the interviews and we identified several high-level features. We presented them to the stakeholders and

after some discussions we ended up with the following final features that the system should provide:

- **Multiuser support and Access control** The system should allow access to multiple users simultaneously. It should also provide access control mechanism that deals with the following issues:
  User types and authorization - What anonymous users are able to do and what registered user are able to do?
  Information privacy - Users should be able to protect private or sensitive information.
  Sharing and collaboration - How can users work together and reuse each other's work?

- **Plug-in environment** The system should enable scientists to extend it by plugging in custom logic such as RDFGears functions and other functional components. Users should be able to manage(add/update/remove) this custom logic runtime(without restarting the system). This process should not affect the work of other users.

- **Scheduling** The system should provide functionality that enables users to schedule and monitor the execution of workflows.

  TODO: What should be made clear is what events are available in scheduling. For example, configure to run on a specified date/time/interval. Or based on the completion of another component or workflow. Or based on the outcome (true/false) of a component, such that you can trigger it based on whether or not you found something in a crawl.

- **Universal data storage**

  Many of the workflows need to store various types of data(e.g. intermediate and final results). Therefore, the system should provide a mechanism that enables storage and retrieval of information in arbitrary data formats.

- **Integration with Hadoop** The amounts of information that have to be processed in the system can sometimes be huge. It is critical that this information is processed efficiently and Hadoop is often the solution for that. Therefore, the system should provide functionality that enables easy integration with Hadoop.

2.2 Feature prioritization

Having already defined the requirements we have to define the order in which we are going to design and implement them. Requirements(features) prioritization is the process of determining the order in which candidate requirements or features of a software product should be implemented. The criteria to order the requirements can vary a lot, for example one can consider the requirements' importance, value, cost, risks or views of stakeholders etc. Scientists and other systems(ImReal) that depend on U-Sem already need the functionality and thus we are mainly concentrated to deliver the most essential functionality as early as possible. Having this in mind, the criteria

we choose for the prioritization is the value(importance) of each functionality as well as the time required for implementation(cost).

Requirements prioritization is a relatively old research topic and there are numerous approaches that are available [4]. The most popular include Quality Function Deployment (QFD) the Analytical Hierarchy Process(AHP), the cost-value approach proposed by Karlsson, Wiegers' method, as well as a variety of industrial practices such as team voting, etc. However, literature also suggests that there is no perfect solution for this problem and the applicability of each approach depends heavily on the particular situation it is used.

For our project, we choose the Karlsson's Cost-Value approach that makes use of the analytic hierarchy process [2]. This decision was based on several factors. Firstly, it uses the exact criteria that we are interested in(value and cost). Secondly, it is especially suitable for prioritizing a small number of requirements[2], which is our case. And last but not least, it is a proven and widely used [3].

2.2.1 Requirements prioritization using the Cost-Value Approach

In this section we will describe step by step the process of prioritizing requirements using the Cost-Value approach. The process consists of three distinct steps. The following sub-sections will address these steps.

Value assessment

In the Requirements gathering section we identified 5 high-level requirements(features) that cover the main functionality of the system. In this step we are using the AHP's pairwise comparison method in order to assess the relative value of the candidate requirements. We asked a group of four experienced project members to represent customers views. We instructed them on the process and asked them to perform pairwise comparisons of the candidate requirements based on their value(importance). For the comparison criteria we used the one defined in []. Fig.. Apendix 1 shows the form that they were asked to fill.

We let the participants to work alone, defining their own pace. We also allowed them to choose the order of the pair's comparison. Discussions were also allowed. When all participant finished the pairwise comparison, we started to calculated the value distribution. However, first we calculated the the consistency indices of the pairwise comparisons. According to [**?**] values lower than 0.10 are considered acceptable and according to [2] values around .12 are commonly achieved in the industry and can also be considered acceptable. The calculation showed that two of the participants has indeces higher than .23 which indicates serious inconsistencies. Therefore, we asked them to revise their answers and the results were around 0.12 **TODO**

	Stakeholder 1	Stakeholder 2	Stakeholder 3	Stakeholder 4
Consistency ratio	0.04	0.23	0.13	0.26

Table 2.1: The initial consistency ratios for each of the stakeholders.

	Stakeholder 1	Stakeholder 2	Stakeholder 3	Stakeholder 4
Consistency ratio	0.04	0.12	0.13	0.11

Table 2.2: Consistency ratios for each of the stakeholders after refinement.

Once we had achieved satisfying results we calculated the distributions. We outlined the candidate requirements in a diagram and presented the results to the project members. Each requirement's determined value is relative and based on a ratio scale. Therefore, a requirement whose value is calculated as 0.20 is twice as valuable as a requirement with a value of 0.10. Additionally, the sum of the values for all requirements equals 1. This means that a requirement with a value of 0.10 provides 10 percent of the value of all the requirements. You can see that the "plug-in environment" requirements is the most valuable. It is followed by the "data-store" and "Access control". At the bottom, providing considerably less value, are the "scheduling" and "hadoop" requirements.

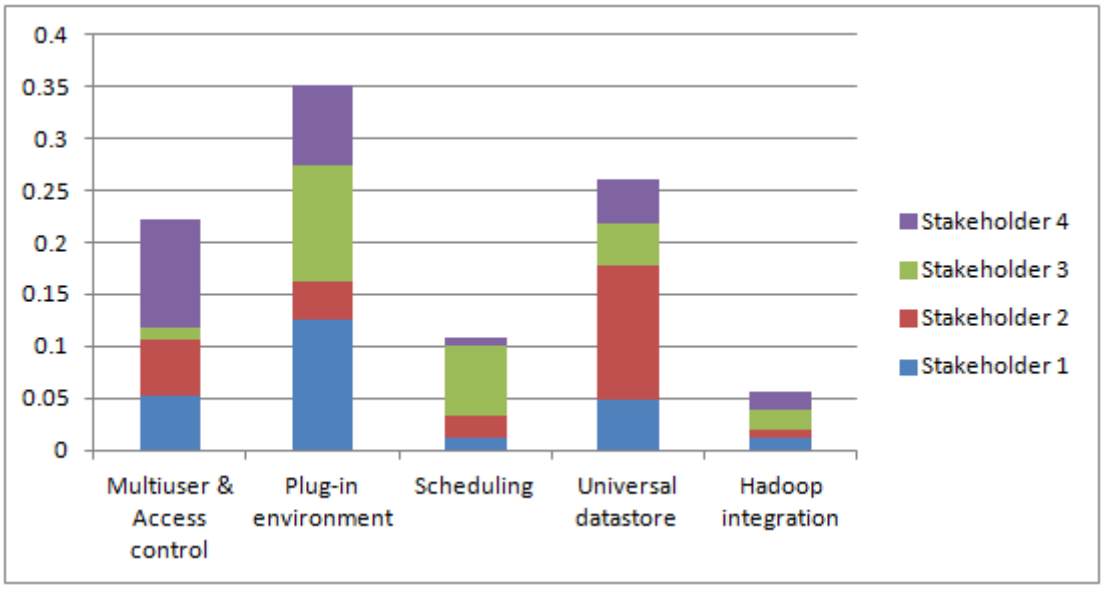


Figure 2.1: The value distribution of the 5 requirements in the U-Sem project.

Cost assessment

Value on its own is not enough to estimate the priority of the requirements. The problem is that a requirement with high value may also be costly to implement and also take a lot of time. Our aim is to provide the most value as quick as possible. Thus, requirements that provide a bit less value but on the other hand are easy and quicker to implement might be the better choice. In this step, we measure the distribution of cost between the requirements. In order to do that, we asked the software engineer responsible to build the system to perform AHP's pairwise comparison to estimate the cost of implementing each of the candidate requirements. The process absolutely the

same as the one described in the previous section. However, this time the requirements are compared based on their cost rather than their value.

Once the comparison was finished, we measured the consistency index of the answers. The result was **0.029** which is completely acceptable and there was no need for further refinement.

Finally, we used the AHP technique to calculate the cost distribution of the requirements. We outlined the results in a digram FIG. Again, the determined values are relative and based on a ratio scale.

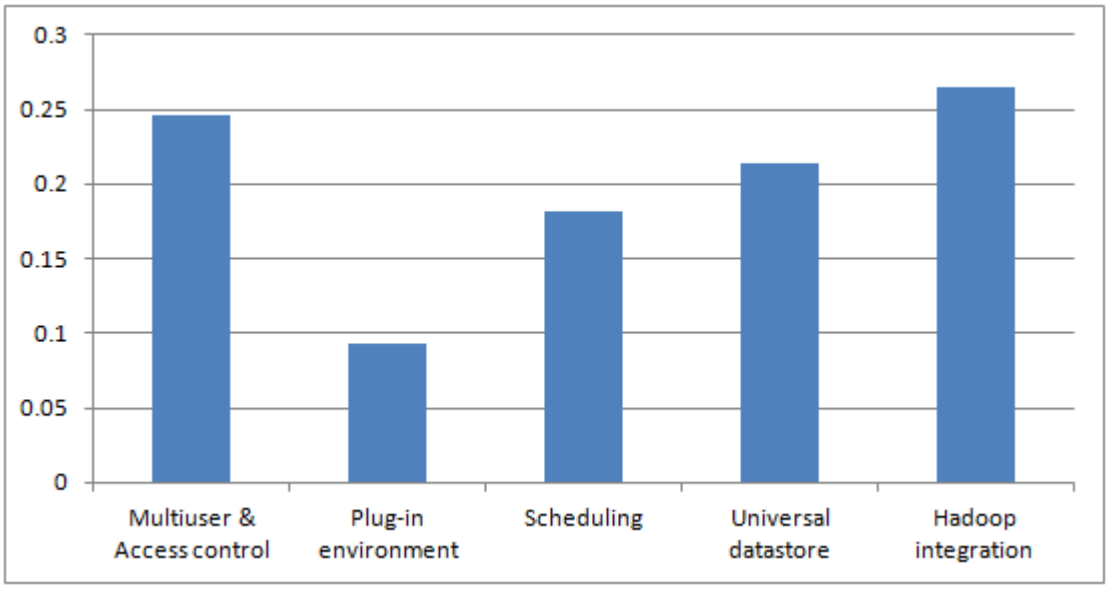


Figure 2.2: The cost distribution of the 5 requirements in the U-Sem project.

Cost-Value analysis

Once we have the value and cost value distribution of the requirements we can establish the order in which to implement the requirements. The decision is based on the value/cost ratio of each requirement. Figure[] shows the results. As you can see, the the plug-in environment is the definite winner and the hadoop integration provides very little benefit compared to the cost to implement it.

Discussion

Our observations about the stakeholders which were asked to perform the pairwise comparisons found the method intuitive and easy to understand. However, they also found performing the pairwise comparison to be a bit tedious. They sometimes got distracted and gave inconsistent results which was indicated by the consistency check. However, this was easily fixed by revising the answers and we reached a level of consistency that is considered acceptable.

Another disadvantage of this approach that affected our work is the fact that the method takes no account of interdependencies between requirements. However, this

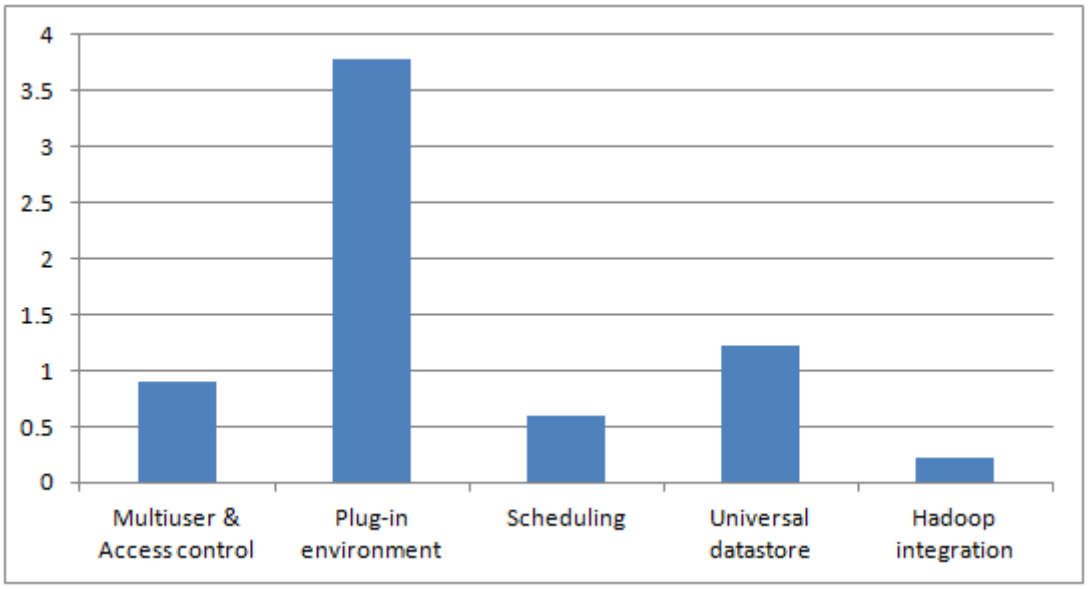


Figure 2.3: This diagram shows the comparison of the value/cost ratio of the requirements.

issue did not affect the project significantly because the project high-level features are loosely coupled and the only one that required some attentions was the "access control". **TODO**

Bibliography

[1] Oscar Dieste, Natalia Juristo, and Forrest Shull, Understanding the Customer: What Do We Know about Requirements Elicitation?

[2] Karlsson, J., Ryan, K. (1997). A Cost-Value Approach for Prioritizing Requirements, IEEE Software September/October 1997, 67-74.

[3] J Karlsson, C Wohlin, B Regnell - Information and Software Technology, 1998 - Elsevier

[4] Frank Moisiadis, THE FUNDAMENTALS OF PRIORITISING REQUIREMENTS

Chapter 3

Dynamic Component Model

This chapter proposes a solution to the problem of extending U-Sem by adding custom functionality. Section 3.1 discusses the problem in details and describes all functional and non-functional requirements that a successful solution must satisfy. Section 3.2 discusses that state of the art approaches and technologies that can contribute to solving the problem. Section 3.3 describes the architecture of U-Sem that we propose in order to solve the problem. Section 3.4 briefly discusses the simple implementation that we provide in order to be able to verify the capabilities of the proposed architecture. Section 3.5 discusses and verifies whether the proposed solution satisfies all of the requirements. Finally, in section 3.6 we discuss the limitations of the proposed design and suggest aspects in which the design can be improved in the future.

3.1 Problem definition

During the initial interviews with scientists that are going to potentially use the system, we reviled that the nature of their work is very dynamic. In their day to day work they are expected to constantly improve and come up with new algorithms and approaches for user modelling. As a result, they are continuously producing new software code that implements these algorithms. After each production cycle, the program code has to be deployed into U-Sem so that it is available for testing, demonstration and evaluation purposes.

We also performed additional interviews with the scientist in order to reveal how this process is currently done, what are the problems they face with it and what are their expectations for the future system. Scientists reported that they have on their disposal only the capabilities of the workflow engine(RDFGears). However, it provides no functionality that enables to plug in custom logic on demand into the system and as a result, scientists are forced to "hardcode" their logic into the source code of the workflow engine. In this way, the software code implementing the algorithms becomes part of the workflow engine. We believe that this approach is error prone and brings a lot of discomfort to the scientists working with the system. The most important disadvantages of this approach include:

- Adding new or modifying existing functionality requires a lot of time and knowledge since in order to do that one has to alter the source code of the workflow

engine and basically, release a new version of it. This process requires advanced knowledge about each phase of the release process: checking out the source code from the software repository, putting the new source code in the appropriate place, building the system and finally, deploying it to the web server. Most of the time, all this knowledge is not required for the daily work of scientists and learning it may create a serious overhead and discomfort.

- In order to add/modify functionality one has to stop the web server where the system is deployed, replace the deployment entities of the system and start the server again. The problem with this approach is that during the time the server is down all other running services are unavailable. This is a major problem for everyone that is using the system during that time.

- Another major disadvantage is that, as a result of all the additional knowledge required, the training period for new scientists is significantly increased. This may easily cause project delays and missed deadlines.

- Multiple scientists adding/modifying functionality simultaneously may result in loss of functionality. Figure 3.1 illustrates the problematic scenario. As stated earlier, in order to add new functionality, scientists must first check out the source code of the system, make the changes and deploy the new version on the web server. However, if two scientist perform this process simultaneously then the new functionality provided by the first scientist will be lost when the second one deploys his version.

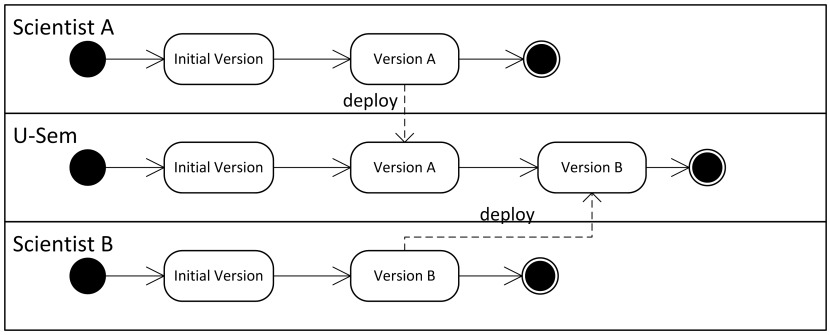


Figure 3.1: State diagram illustrating the scenario where two scientists extend U-Sem simultaneously and the changes made by Scientist A are lost.

- And last but not least, it is hard to verify what is the exact state of the system at any particular moment. Unless documented exclusively, it is not clear what additional functionality is added to the system. This problem becomes more serious when there are more people working on the project simultaneously and it is hard to track the changes in the system.

This approach also introduces one big disadvantage from software engineering perspective. The problem lies in the poor modularization of the system. In software engineering, modularization is considered as a key property for improving extensibility, comprehensibility, and reusability in software projects [4]. The most important

aspect of a successful modular system is its information hiding capabilities [17]. In our case, scientists can only rely on the modularization functionality provided by the Java language. However, its information hiding principles are only applied on class level, but not to the level of packages and JAR files. For example, it is not possible to restrict access to certain public classes defined in a package. The absence of such visibility control can easily lead to highly coupled, "spaghetti-like" systems [18]. The consequences of this will become more and more clear with the time when the system grows in size, complexity and the number of engineers working on it increases. The most probable consequences include high development costs, low productivity, unmanageable software quality and high risk to move to new technology [6].

Having all these considerations in mind, we devised a complete set of requirements that presents the functional scenarios(functional requirements) and system qualities(non-functional requirements) that the proposed architecture has to provide. These requirements are also referred to into the evaluation section where we discuss how and to what extend the architecture satisfies each of them.

3.1.1 Functional Scenarios

In this section we formally identify the functional requirements which define the main interactions between the scientists and the system. Each scenario is marked with a code at the beginning which is used for easier identification during the verification and evaluation phase.

- **UC1 - Create custom functionality for U-Sem** - This is the main scenario regarding the feature addressed in this chapter. Scientists has to be able to extend U-Sem by adding custom functionality on demand. They have to be able to compose the custom functionality independently from the system, add the produced functionality to U-Sem while the system is running and/or if desired, share it with other scientists.
- **UC2 - Use functionality shared by other scientists** - Scientists has to be able to reuse custom functionality that is previously shared by other scientist.
- **UC3 - Manage loaded functionality** - Users has to able to manage all functionality already added to the system. This includes, firstly, that they have to be able to view a list of all added functionality. And secondly, they have to be able to remove any of the functionality from the provided list.

3.1.2 Non-functional requirements

This section identifies the main quality scenarios that a successful architecture has to accommodate.

- *Availability* - It is expected that in future the system will be used by many users and therefore, any down time of the system is unacceptable. On one hand, performing any of the functional scenarios defined in the previous section should not cause any kind of unavailability of the system. On the other hands, the system has to be able to withstand failures of the computing units. There two types of failures that the system has to be able to deal with:

  - *Hardware failures* are one of the main problems that may cause the entire system to fail. They might be caused by processing unit failure, network failure and electricity supply failure. In case of one or several failures of any type, the system has to be able to continue to operate normally. In the worst case, a decrease in performance is acceptable.
  - *Software failures* are also a common cause for system failures. In case of software failure in a part of the system, the system has to continue to operate normally. Again, in the worst case, a decrease in performance is acceptable.

- *Scalability* - the architecture should enable scalability and should not affect the scalability capabilities of the workflow engine. The system has to be designed in such a way that allows to easily add new hardware processing units when it is required by increase in the number of users using the system.

- *Isolation* - A scientist should not be affected by the work of the others. The only way of interaction between scientists has to be achieved through the sharing mechanism. Moreover, scientists should not be affected by any future changes to the reused components.

- *Security* - This is also very important requirement since the system executes custom code and thus, is vulnerable to deliberate or unintentional exploitation of vulnerabilities. Therefore, the system should provide mechanism that enables administrators to enforce different restrictions on the executed custom code. For example, the administrators might want to forbid access to the file system or the network.

  Additionally, although not critical, sometimes scientist might want to be insured that their custom functionality is protected and it cannot be accessed against their will. Therefore, the system has to provide secure transportation and access mechanism for the custom functionality.

  Finally, all installed functionality should be backuped. In case of failure of a storage device, the system should provide a quick and reliable method for recovering of all the data.

- *Modularization* - As we discussed in the previous section, modularization of components is crucial for the future of the system. Therefore, the architecture has to provide facilities that support and enforce strong modularization between components.

3.2 Approach

This section discusses the currently available approaches and technologies that might be used in order to design a system that fulfils the requirements specified in the previous section. After a detailed investigation of the requirements, we reached the conclusion that the core of the problem lies, firstly, in the poor separation of concerns(modularization of the functionality) and secondly, in the impossibility to manage(add, replace, remove) these concerns while the system is in operation. In order

to solve these problems, we investigated the scientific literature to find what are the available approaches and technologies that can help to overcome these problems. Our research relieved that the topic about modularization of software systems is widely discussed and there is even a sub field in software engineering which addresses the problem of building systems out of different components [5]. This sub field is called *Component-based software engineering*.

In the next subsection we discuss the basic idea and the advantages that this approach brings. We also discus what features a system needs to provide in order to enable Component-based software engineering. This is known as component model. At the end, we also discuss the state of the art technologies that provide support for Component-based software engineering, enable dynamic management of the components and are useful in the context of U-Sem.

3.2.1 Component-based software engineering

Component-based software engineering is based on the idea to construct software systems by selecting appropriate off-the-shelf components and then, assemble them together with a well-defined software architecture [1]. This software development approach improves on the traditional approach(building application as a single entity) since applications no longer have to be implemented from scratch. Each component can be developed by different developers using different IDEs, languages and different platforms. This can be shown in Figure 3.2, where components can be checked out from a component repository, and assembled into the desired software system. This completely complies with the idea behind U-Sem where each scientists is responsible to build only a peace of the system which provides certain service.

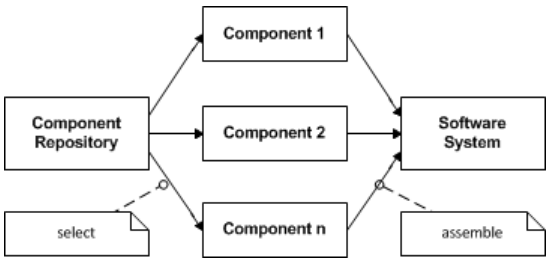


Figure 3.2: Component-based software development [1]

Other benefits that Component-based software development brings and we believe U-Sem will benefit from include: significant reduction of development cost and time-to-market, and also improvement on maintainability, reliability and overall qualities of software systems [2] [3]. Additionally, the applicability of this this approach is supported by the fact that is widely used in both the research community and in the software industry. There are many examples of technologies implementing this approach including: OMG's CORBA, Microsoft's Component Object Model(COM) and Distributed COM (DCOM), Sun's(now Oracle) JavaBeans and Enterprise JavaBeans, OSGI.

3.2.2 Component model

Designing the component model of a system provides the specification defining the way that the system can be build by composing different components applying the component-based software engineering approach. More formally, it is the architecture of a system or part of a system that is built by combining different components [6]. It defines a set of standards for component implementation, documentation and deployment. Usually, the main components that a component-based software system consists of are [7]:

Interfaces determine the external behaviour and features of the components and allow the components to be used as a black box. They provide the contract which defines the means of communication between components. As illustrated on figure 3.3, interfaces can be considered as points where custom functionality provided by another component can be plugged in.

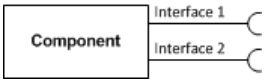


Figure 3.3: Component interfaces

Components are functional units providing functionality by implementing interfaces. As can be seen on figure 3.11, components provide features by implementing the interfaces provided by other components. One of the main question regarding building components is how to define the scope and characteristics for a component. According to [6] there are no clear and well established standards or guidelines that define this. In general, however, a component has three main features:

- a component is an independent and replaceable part of a system that fulfils a clear function
- a component works in the context of a well-defined architecture
- a component communicates with other components by its interfaces

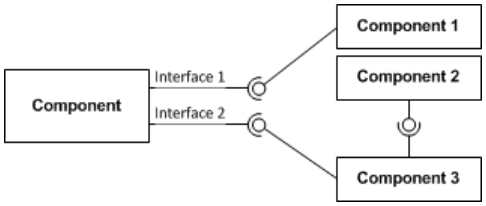


Figure 3.4: Components implementing interfaces

Coordinator is the entity which is responsible to glue together and manage all the components. It is needed because components provide a number of features, but they are not able to activate the functionality themselves. This is the responsibility of the coordinator.

3.2.3 State of the art component model implementations

In previous sections, we discussed that integrating a component model in the architecture of U-Sem is the scientifically proven approach that promises to solve the design problem and fulfil the requirements of the customers. However, we had to decide whether to design and implement our own custom component model or we can reuse an existing one. Reusing a popular and widely used solution might be beneficial because it is likely it is heavily tested(at least from the engineers using it) and thus provide higher quality.

[8] suggests classification of the component model implementations based on which part of the life cycle of a system the composition of the components is done. They identify the following groups:

- Composition happens during the design phase of the system. Components are designed and implemented in the source code of the system.
- Composition happens during the deployment phase. Components are constructed separately and are deployed together into the target execution environment in order to form the system.
- Composition happens during the runtime phase. Components are put together and executed in the running system.

For the architecture of U-Sem we are only interested in the last group since one of the main requirements is that scientists should be able to add, update and remove components while the system is running, without restarting it. This is essential since the system is used by multiple scientists and system restart will cause temporary unavailability of all services.

Apart form this, there is also another critical concern when choosing component model implementation for U-Sem. The implementation should support the Java language since it is the language in which all current services are implemented and most scientists are familiar with. Having to learn a new language and/or rewriting all source code in different language is considered as a big disadvantage for the scientists.

We performed further investigation in order to find what are the current state of the art technologies that satisfy all requirements. It showed that currently there two standards that satisfy our needs: Fractal [9] and Open Services Gateway initiative (OSGI) [10]. Both of them seemed quite popular and widely used and therefore we concluded that reusing them is more beneficial than implementing a component model from scratch. For the proposed architecture of U-Sem we chose to use OSGI since our impression is that it provides a simpler way of defining components(no component hierarchies) which will be beneficial for scientists that do not have so in depth knowledge of component-based engineering. OSGI is also widely used[11] which may suggest that it is well tested and therefore is more stable. Next subsection focuses on how OSGI works and its features that are interesting for the architecture of U-Sem.

3.2.4 OSGI

Proposed first in 1998, OSGI represents a set of specifications that defines a component model which represents a dynamic component system for Java. These specifications enable a development model where applications are dynamically composed of different independent components. Components can be loaded, updated and deleted on demand without having to restart the system. OSGI defines the main components of the standard component model which are discussed in the previous section as follows:

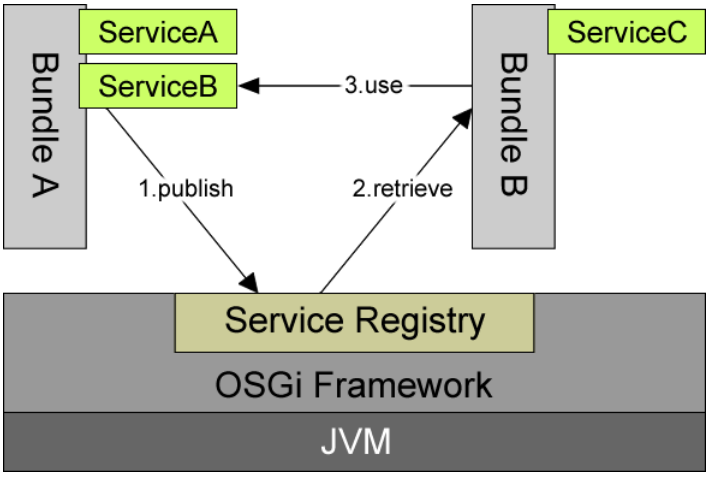


Figure 3.5: OSGi Service Registry [11]

Interfaces in OSGI define the contract for communication between different components by describing the operations that has to be implemented by the components. Basically, they represent standard Java interfaces or classes which has to be available to both the component that implements the interface and the components that use the implemented functionality.

Components in OSGI are called bundles. Bundles are basically a regular Java JAR files that contain class files and other resources such as images, icons, required libraries. One of the important benefits for U-Sem is that OSGI enables better modularization providing facilities for better information hiding then the one provided by the Java language [11]. Each bundle should provide a manifest file, which enables engineers to declare static information about the packages that are exported and therefore can be used by other bundles. Furthermore, bundles provide functionality to the rest of the system in the form of services. In the OSGi architecture, services are standard Java objects that implement the interfaces described in the previous paragraph.

Coordinator The OSGi standard also provides coordinator component which represents a runtime infrastructure for controlling the life cycle of the bundles which includes adding, removing and replacing bundles at run-time, while preserving the relations and dependencies among them. Another key functionality that the coordinator component of OSGi provides is the management of the services provided by the

bundles. This functionality is provided by the Service Registry, which keeps track of the services registered within the framework. As illustrated on Figure 3.5, when a bundle is loaded it registers all the services that it implements(step 1). As soon as a service is registered, it can be retrieved by any other components that are interested in this functionality (step 2). Once a bundle has retrieved a service, it can invoke any method described by the interface of this service (step 3). Another interesting feature of the OSGi Service Registry is its dynamic nature. As soon as a one bundle publishes a service that another bundle is interested in, the registry will bind these two bundles. This feature is very important for U-Sem since it will enable scientists to plug in any new functionality dynamically when it is needed.

Security

The security capabilities of OSGI are also very important for U-Sem since a lot of custom code is being executed which poses a significant threat to the system. The OSGi platform is considered to be highly secure by its creators [14]. There are two main reasons in favour of this statement. Firstly, OSGi is executed on top of the JVM and inherits its security capabilities. Secondly, it has been designed to support a proper level of isolation between components.

Since its introduction, the Java language and JVM have been widely used and subjected to extensive tests and validation [14]. The platform was designed to support the safe execution of fully untrusted code [23]. In order to achieve this, it has introduced the following features [24]: Type Safety of the language, Garbage Collection(no user-defined memory management), Bytecode verification ensuring that executed programs are compliant with the language specification, and Sandboxing, which can prevent the access to sensible resources like the network or file system. Additionally, the system provides secure class loaders, which enable loading of several modules that cannot interact with each other.

On top of this, OSGI defines additional permissions that provides full control of the interactions between the components. It provides functionality that enables dependencies at the package level and at the service level to be allowed or prevented. Additionally, management capability from one component to the others can also be restricted [14].

3.3 Architecture

This section describes the proposed architecture for the dynamic component model feature of U-Sem. It focuses on the four aspects of typical for a software architecture [19]: its static structure, its dynamic structure, its externally visible behaviour, and its quality properties.

One of the critical things when describing a software architecture is to manage complexity of the description so that it is clear and understandable by the stakeholders. [19] suggests that capturing the essence and all details of the whole architecture in a single model is only possible for very simple systems. Doing this for a more complex system is likely to end up as a model that is unmanageable and does not adequately represent the system to you or any of the stakeholders. [19] also claims that the best way to deal with the complexity of the architecture description is to break

it into a number of different representations of all or part of the architecture, each of which focuses on certain aspects of the system, showing how it addresses some of the stakeholder concerns. These representations are called views.

In next sections, we provide a set of interrelated views, which collectively illustrate the functional and non-functional features of the system from different perspectives and demonstrate that it meets the requirements. Note that in the reminder of this chapter we use the term *plug-in* for the components containing the custom functionality provided by scientists. In this way they are easily distinguished from the architectural components.

3.3.1 Context View

The Context view aims to define the environment in which the system operates and more specifically the technical relationships that the system has with the various elements of this environment [20]. [20] also identifies the concerns that the view has to address:

- Identify which are the external entities and what are the responsibilities of each of them.
- Identify the dependencies between the external entities which affect the system.
- Identify the the nature of the connections between the entities.
- Define the interactions that are expected to occur over the connections between the entities.
- Define what are the system's external interfaces.

As explained in the introduction section, initially, the system only communicated with providers of semantic content and the clients which execute the services defined by the scientists. The way the interactions work is not affected by the architecture discussed in this chapter. Introducing the dynamic component model of U-Sem, however, brings two new entities on the stage. Figure 3.6 illustrates the updated runtime environment of U-Sem.

Scientists Apart from defining workflows, the role of the scientists is also to add new functionality to U-Sem. When scientists want to do that they first have to build a plug-in that encapsulates the new logic. Then, they upload the plug-in to U-Sem through a web user interface and it will be installed in the storage space of the scientist. Once installed, the scientist can start using the newly added functionality. Additionally, scientists can also communicate with U-Sem in order to manage the existing plug-ins loaded into the system. Using a web user interface, they can view the list of all available components and if needed they can even remove some of them. All communication between scientists and U-Sem is achieved trough HTTP/s.

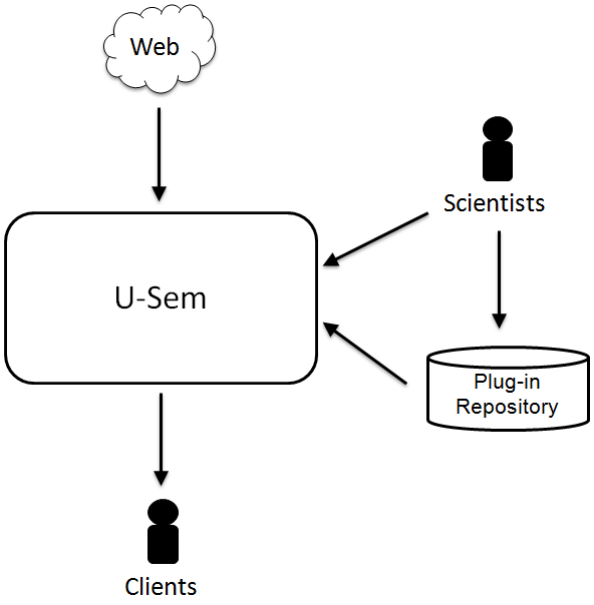


Figure 3.6: Context diagram of U-Sem

Plug-in repository Another interesting addition to the environment is the Plug-in repository. It represents a storage location where plug-ins are stored and when needed, they can be retrieved and installed into the system. Scientists build and then publish their plug-ins there so that anyone interested can install and use them. The need for this approach emerges from the fact that scientists has to be able to share functionality with one another.

By using the Plug-in repository scientists are able to exchange components before they are installed into U-Sem. Alternative approach would have been enable scientists to share already installed components between each other. In this way, whenever a scientist creates a new version or an entirely new component it is installed to U-Sem, shared and then all other scientists are able to use it. However, that approach has one major disadvantage. When a new version of a component is installed then all scientists automatically start to use the new version. The version, though, might introduce a bug or it might not be completely compatible with the previous version. As a result, all other scientists' services and components that are using it are threatened to experience failures.

Using the Plug-in repository overcomes this problem. When a scientist releases new version of component, other scientists can decide whether or not to immediately adopt the new release. If they decide not to, they can simply continue using the old one. Later, when they decide that are ready for the change, they install and use the new release. The benefit from this approach is that none of the scientists are at the mercy of the others. Changes made to one component do not need to have an immediate affect on other scientists that are using it. Each scientist can decide whether or when to move to the new releases of the components in use.

3.3.2 Process modelling

After we have identified the new actors(scientists) and external systems(plug-in repository) in the environment of U-Sem we have to define how they interact with each other. In this section we use the Business Process Management Notation(BPMN) [12] to define the business processes that describe the interactions needed for each of the use cases regarding the dynamic component model feature of U-Sem. The notation enables us to model activities, decision responsibilities, control and data flows. The decision to use BPMN to define the interactions is based on its suitability for interaction modelling and the fact that it is more popular compared to its alternatives [13]. Next subsections describe each of the defined processes and expand them into Business Process Diagrams (BPD).

Create U-Sem Plug-in process

UC1 - Create custom functionality for U-Sem is the most important use case regarding the dynamic component model feature. In this section, we modelled this use case as a business process. Figure 3.7 provides the business process model diagram that illustrates this process.

As illustrated in the diagram, there are three participants in this process(U-Sem, Scientists and the Plug-in repository) which are illustrated in separate BPMN pools. When a scientist wants to create new functionality for U-Sem, he/she first writes the source code, providing all required resources and implementing the desired U-Sem interfaces(the component interfaces discussed in previous sections). Then, everything is built and encapsulated into a single plug-in. If it is only for private use, scientists can directly upload it to U-Sem . When U-Sem receives a component it is responsible to install it into the scientist's dedicated storage space and make available all functionality provided by the component. Finally, U-Sem sends confirmation message back to the scientist. Alternatively, the scientist might also want to share the component with other scientists. In this case, the component is sent to the plug-in repository. When received, the repository is responsible to store it and make it available to the other scientists. Again, at the end a confirmation message is sent back to the scientist.

Reuse shared plug-ins

As defined in section 3.1, U-Sem also enables scientists to reuse plug-ins shared by other scientist(UC2). This use case is also modelled as a separate business process which is illustrated in Figure 3.8. Again, we have three participants in this process(U-Sem, Scientists and the Plug-in repository) which are illustrated in separate BPMN pools.

The process consists of two main phases. First, the scientist contacts the plug-in repository in order to determine what are the currently available plug-ins and then, he/she contacts U-Sem providing information about the desired plug-ins. Upon receiving the request, U-Sem is responsible to contact the plug-in repository and retrieve the requested plug-ins which are, at the end, installed into the private space of the scientist and a confirmation message is send back.

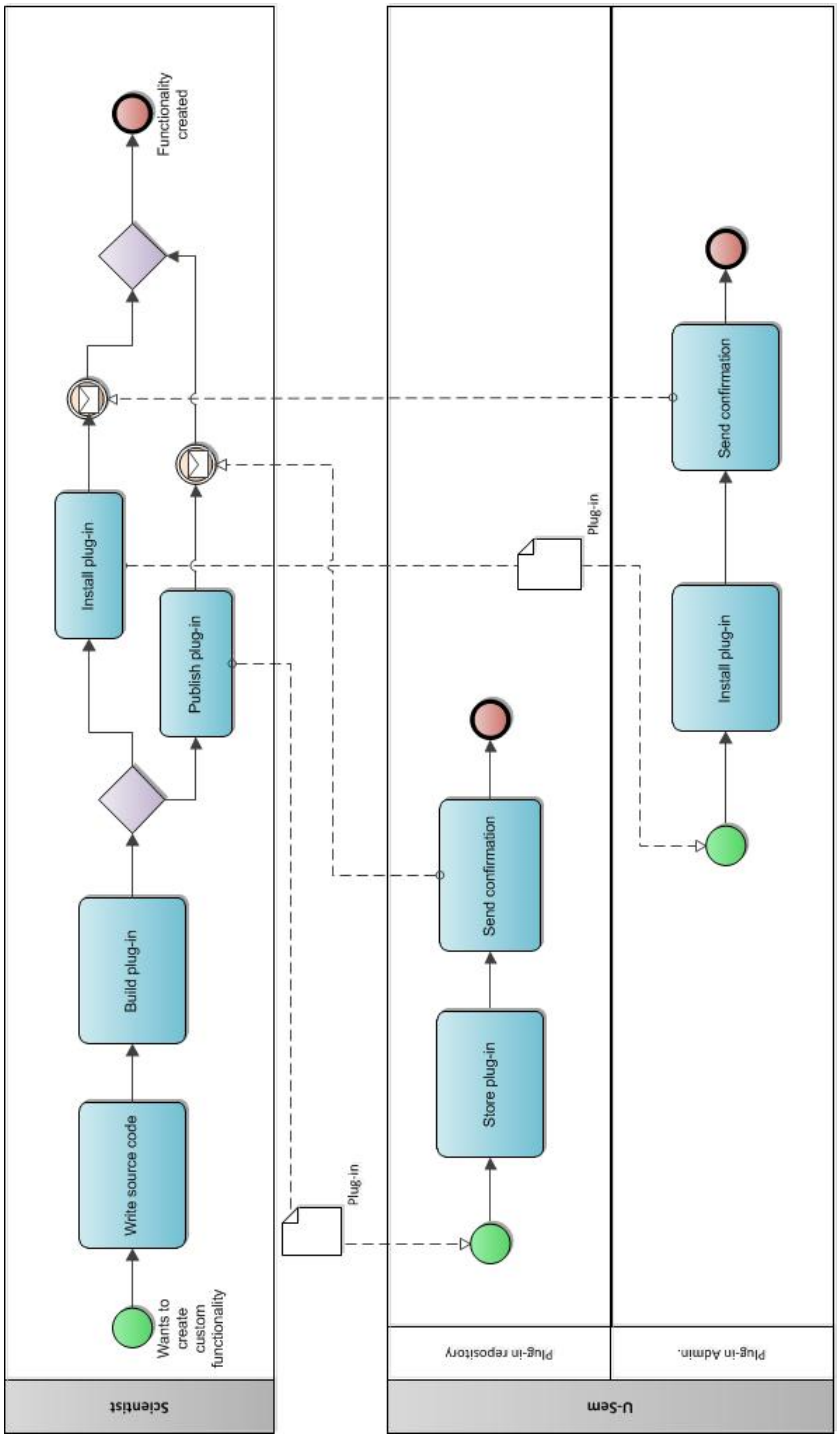


Figure 3.7: Business model describing the process for creating a new plug-in for U-Sem

Plug-in Management

Managing plug-ins(UC3) is also another important use case. Implementing it enables scientists to view all plug-ins installed into U-Sem and if needed remove any of them.

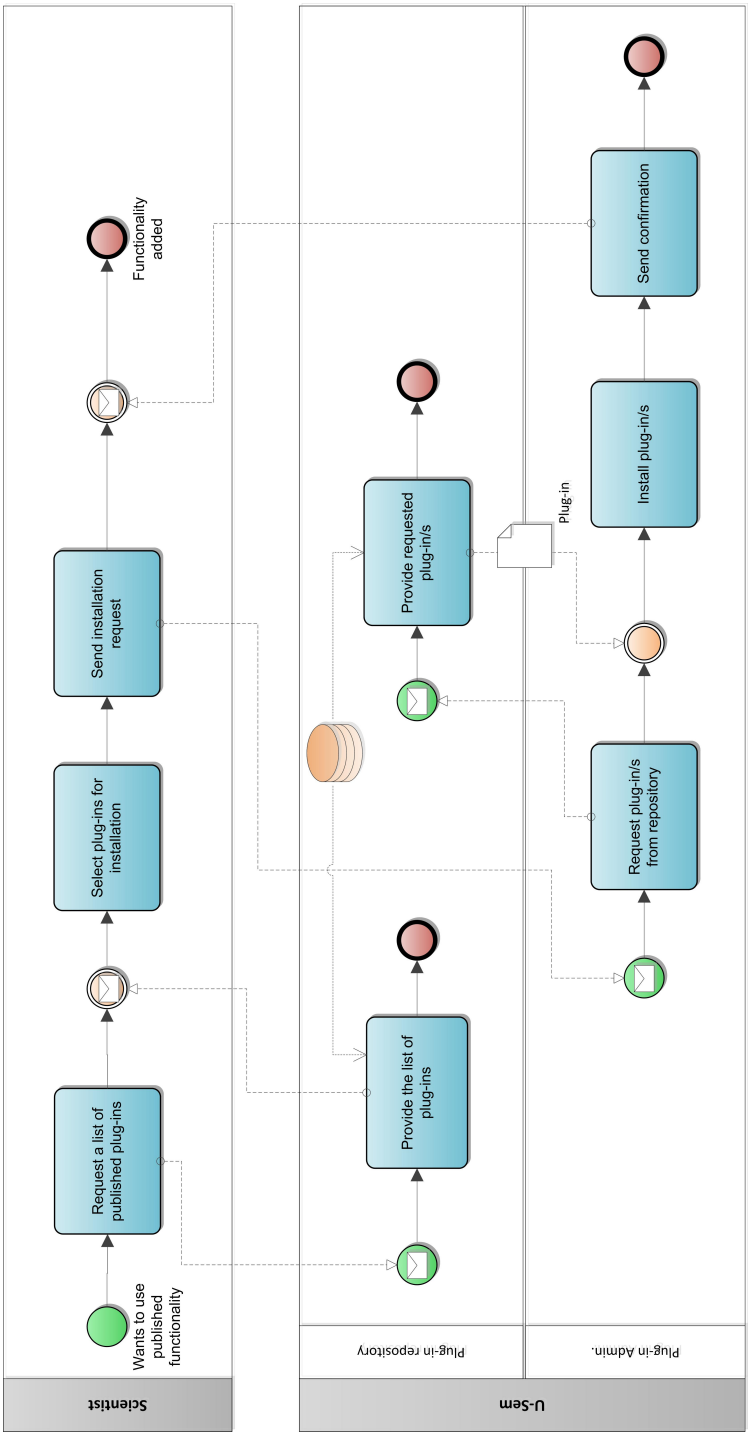


Figure 3.8: Business model describing the process for installing a shared plug-in/s to U-Sem

This use case was modelled into the *Plug-in Management process* which is illustrated on Fugure 3.9. In this case, we have interaction only between the scientist and U-Sem.

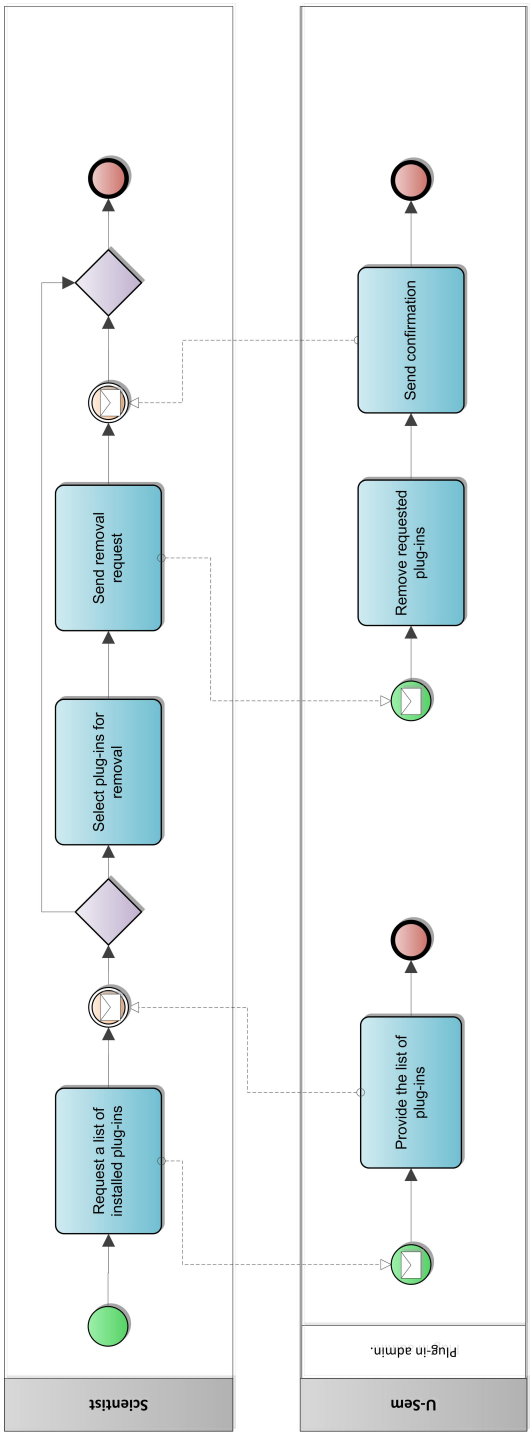


Figure 3.9: Business model describing the process for managing plug-ins in U-Sem

Scientists can monitor the currently installed plug-ins at any time by contacting U-Sem. When such request is received, U-Sem is responsible to send back detailed information about all the plug-ins. Having this list, scientist are also able to remove

plug-ins. In this case, scientists have to submit request for removal providing details for the plug-in that has to be removed. Upon receiving a request for plug-in removal, U-Sem is responsible to permanently remove it form the private space of the scientists and when finished send back a confirmation message.

3.3.3 Functional view

After identifying all actors that are part of the environment of U-Sem and the way they interact with one another, in this section, we define the internal structure of U-Sem that is responsible to accommodate all these interactions. The functional structure of the system includes the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them [19]. All these together demonstrate how the system will perform the required functions.

All components that take part in the dynamic component model functionality can be classified in three layers. This organization is illustrated in figure Figure 3.10 and consists of the following layers:

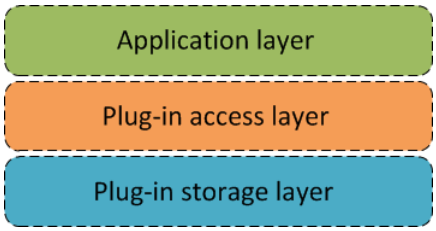


Figure 3.10: Layer organization of U-Sem

- *Plug-in storage layer* is responsible to provide storage functionality for storing the installed plug-ins. Additionally, it should provide place where plug-ins can store data during their execution.

- *Plug-in access layer* provides functionality for plug-in management and provides access to services provided by the plug-ins. The functional components that build this layer are responsible to enforce the security and privacy policies of the system.

- *Application layer* this layer consists of all functional components that are interested in using the services provided by the plug-ins. These applications are also responsible to provide functionality to the user for adding new plug-ins to the system or managing the existing ones.

High-level component organization

This section describes the internal structure of the layers and identifies the high level components that build up the feature. Figure 3.11 illustrates this organization. It shows how the high-level components are organized into the layers and the way they depend on each other. We have identified the following high level components:

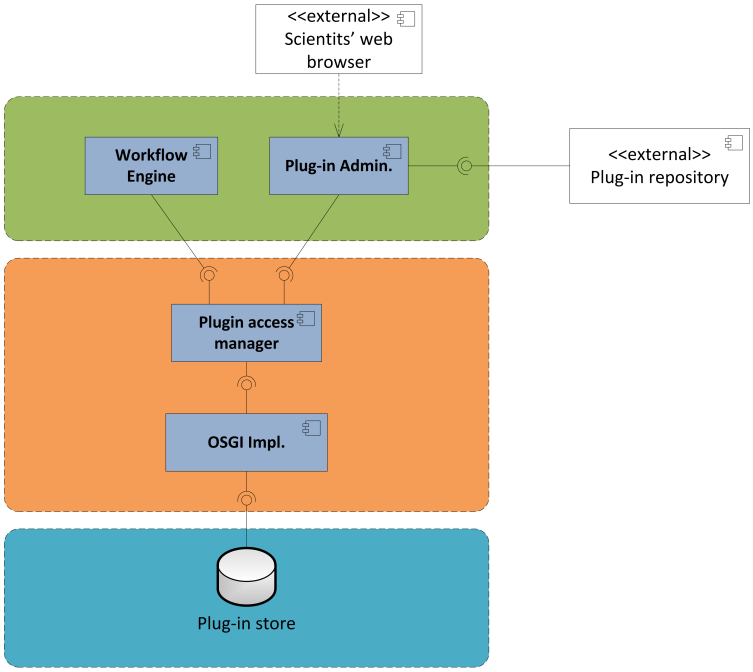


Figure 3.11: Component diagram illustrating the functional organization of U-Sem

- *Plug-in Store* is responsible to store the installed plug-ins for each user. It should provide permanent store for the components so that after system restart they are still available. This component is also responsible to provide storage space for each component in case any data storage is required. It has to ensure that at any point of time the data is secured and backed up. The current state of OSGI only allows integration with file systems for plug-in storage and therefore, this component has to provide file system interface for communication.

- *OSGI Implementation* - As we already discussed in previous sections, we will use the OSGI standard as a base for providing the dynamic component model for U-Sem. It is responsible to mange the plug-ins' life cycle and provide access to the services implemented by the different components. It provides an API which enables other components to communicate with the framework.

- *Plug-in access manager* acts as a level of abstraction over the OSGI component. It is responsible to deal with the configuration and manage the life cycle of the OSGI framework. It is also responsible to enforce the security policy and provide isolation between scientists. It provides API for the application layer components for dealing with services and management of plug-ins. Further decomposition of this component is provided in the next sections.

- *Plug-in admin* is responsible to deal with the administration of the plug-ins. It provides the system's endpoint(user interface) for interaction with the scientists. Additionally, it also provides functionality for communication with the plug-in repository. Further decomposition of this component is provided in the next section.

- *Workflow engine* uses the interface provided by the *Plug-in access manager* to access the services implemented by the components. During the workflow configuration phase it uses the interface in order to obtain the list of available services implemented by the components, while during the workflow execution phase it uses the interface to execute and retrieve the result of the services.

Plug-in admin

This section defines the functional decomposition of the *Plug-in admin* component which is illustrated on figure 3.12. It consists of the following components:

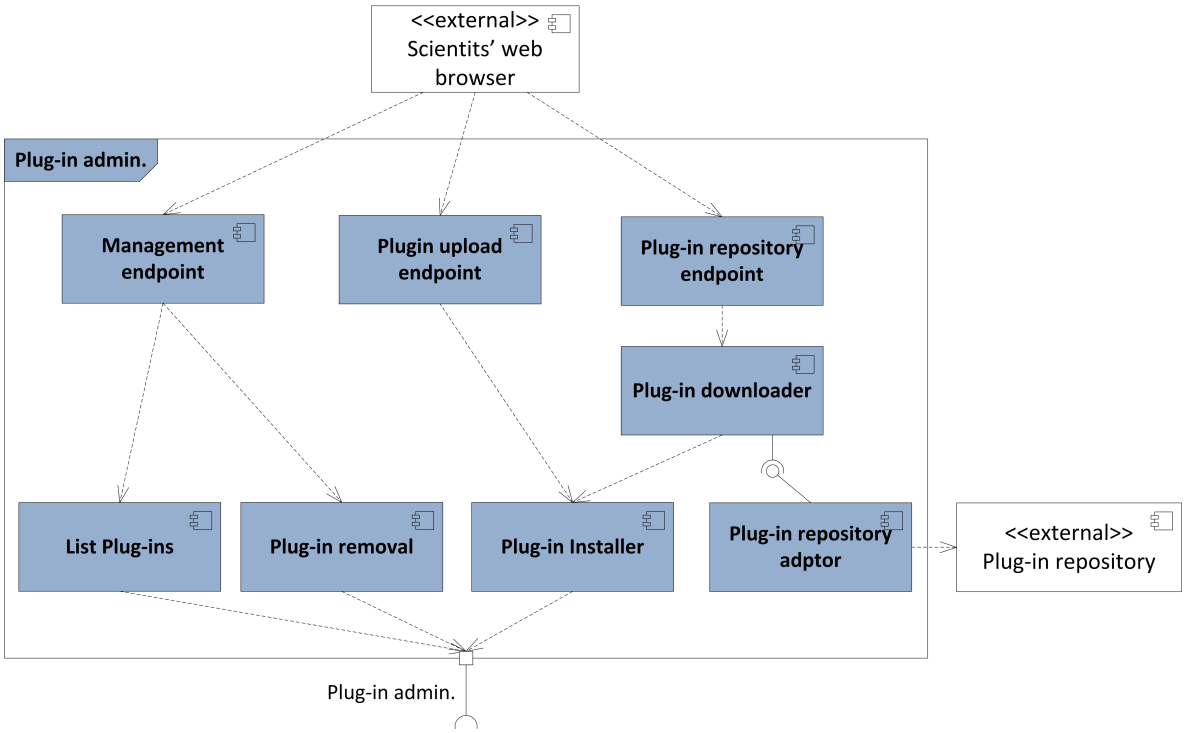


Figure 3.12: Functional decomposition of the *Plug-in admin.* module

- *List Plug-ins* - This component acts as a level of abstraction over the OSGI framework and is responsible to provide functionality for tracking the current state of the system. The state consists of a list of all plug-ins that are installed for the particular user including detailed information for each of them: id, name, vendor, etc.

- *Plug-in removal* - This component also acts as a level of abstraction over the OSGI framework but it provides the functionality for removing installed plug-ins.

- *Management endpoint* - This component provides the user interface for the plug-in management functionality. It acts as a bridge between the user and the components that provide the actual functionality. It depends on the *List Plug-ins* and

Plug-in removal components in order to be able to present the current state of the system and enable users to remove components.

- *Plug-in installer* - This component receives a plug-in in the form of a *jar* file and is responsible to install it to the storage space of a particular user using the API provided by the OSGI framework.
- *Plug-in upload endpoint* - This components provides the user interface needed for uploading plug-ins. It enables users to select a *jar* file from their local file system and upload it for installation. When the plug-in is uploaded to U-Sem it is sent to the *Plug-in installer* for future processing.
- *Plug-in repository adaptor* - This component manages the communication with the plug-in repository. It acts as a level of abstraction over it. U-Sem might evolve in future and need to use different repositories and in that case, this is the only component to change if support for a new repository system is needed.
- *Plug-in downloader* - This component is responsible to download the desired plug-ins from the repository and upon successful download notify the *Plug-in installer* to continue with the installation of the plug-in.
- *Plug-in repository endpoint* - This component provides the user interface which enables users to browse the plug-in repository and indicate which plug-ins should be downloaded and installed on U-Sem.

Plug-in access manager

This component is responsible to provide API which can be used by application layer components in order to manage the plug-ins and access the functionality provided by them. Figure 3.13 shows the functional decomposition of the Plug-in access manager module. It consists of the following components:

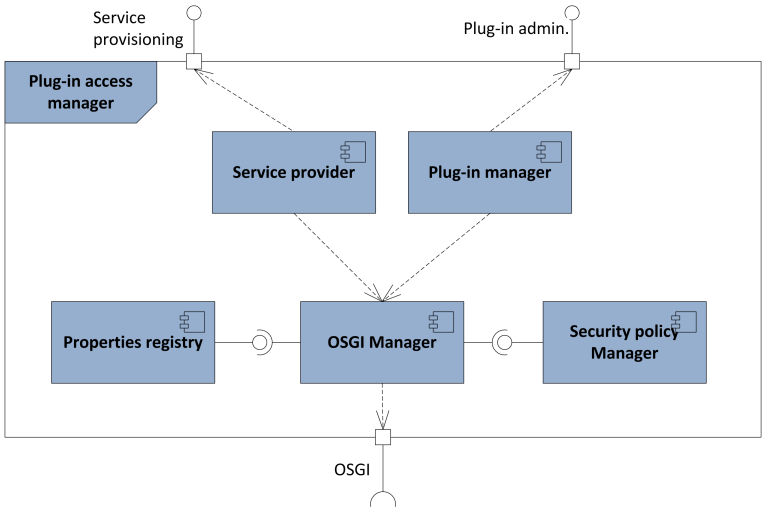


Figure 3.13: Functional decomposition of the *Plug-in access manager* component

- *OSGI Manager* - This component manages the communication with the OSGI framework. It is responsible to start/stop the framework and monitor its life cycle. All needed properties for the framework operation are provided by the *Properties registry* component. The OSGI Manager is also responsible to enforce the security policies by setting up the Security Manager options provided by the *Security policy* component. It also acts as a level of abstraction over the plug-in engine and in case any change in future is required, this is the only component that will be affected.

- *Properties registry* - Keeps track of all common and user related options that are required for the correct operation of the OSGI framework. The main properties this component is responsible to provide are the paths to the plug-in storage space for each user. It has to make sure that this spaces are not overlapping. Additionally, it also provides information about the security policy that has to be applied to a particular user.

- *Security policy manager* - This component is responsible to provide access to the security policies of U-Sem which define the operations that the custom code loaded by the framework is allowed to perform.

- *Plug-in manager* - This component is responsible to provide an API for the *Application layer* components that enables them to perform plug-in management tasks.

- *Service provider* - This component is responsible to provide an API for the *Application layer* components that enables them to access the services implemented by the loaded components in U-Sem.

3.3.4 Concurrency view

This section describes the concurrency structure of U-Sem. We show how functional elements map to concurrency units(processes, process groups and threads) in order to clearly identify the parts of the system that can work concurrently. We also show how this parallel execution is coordinated and controlled.

Figure 3.14 illustrates the concurrency organization of U-Sem. The main functionality of the system is situated in the U-Sem process group. All U-Sem processes and all external processes(Plug-in repository) including the Database process operate concurrently. The main processes wait for requests from the user(web browsers and/or other systems). Each request is processed in separate thread depending on its type(workflow or plug-in management related tasks). As a result, multiple client requests can be handled simultaneously. Workflow execution initiated by U-Sem clients and plug-in manipulation by scientists can happen at the same time. However, the two processes does not communicate with each other and thus, the synchronization on the installed plug-ins is achieved throw the storage process. Every time before executing a workflow, the process loads the installed plug-ins into the main memory. This means any changes to the plug-ins during an execution of a workflow will not be applied and thus, the execution will not be disrupted. All changes will take place the next time a workflow is executed.

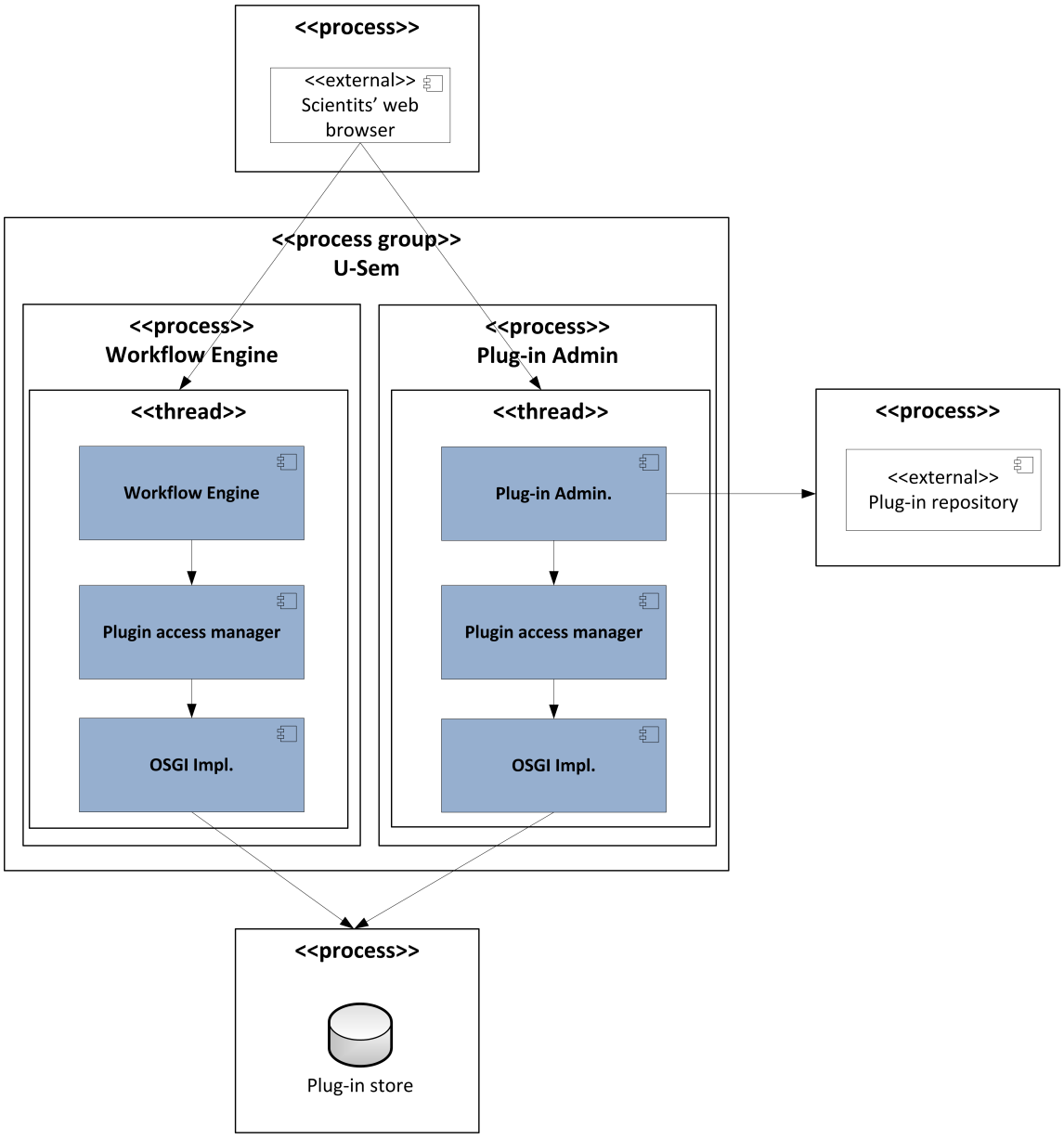


Figure 3.14: Diagram illustrating the concurrency model of U-Sem

3.3.5 Deployment View

This section describes the environment in which the system will be deployed. It defines where each of the processes defined in the previous section will operate.

Due to the nature in which the system operates we had to consider scalability but also simplicity. On one hand, in order to test their work scientists has to be able to easily install and set up U-Sem. Usually, in this scenario scalability and performance are not critical, however, simplicity is crucial to enable scientist to concentrate entirely on their work rather then setting up the system. On the other hand, in production mode the system will be used by many users and many services will be running simultaneously.

Therefore, in this situation the performance and scalability issues are important. As a result, we designed U-Sem to be flexible and be able to accommodate both scenarios.

Simple setup

This setup targets the scenarios where there is no need for high performance and scalability. The aim here is to enable scientists to setup the system fast, on a single machine and with little configuration effort.

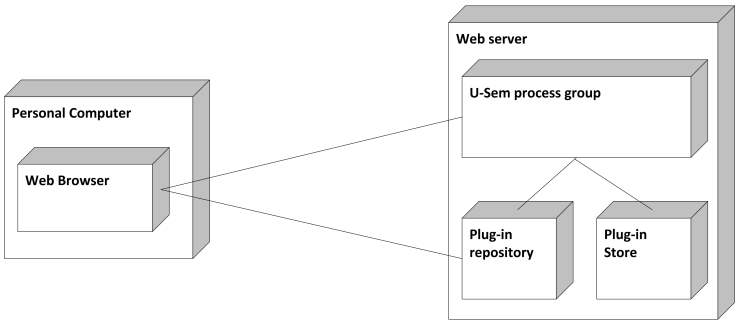


Figure 3.15: Deployment diagram illustrating the simple setup of U-Sem

This deployment organization is illustrated on figure 3.15. As one can see, all components are deployed and run on the same web server and on the same physical machine. We recommend this setup only for development and test purposes since it does not satisfy the scalability, availability and security requirements. For production use, we recommend the setup described in the next section.

3.3.6 High Availability setup

This setup aims to cover the scenarios requiring scalability and high availability. Figure 3.16 provides the deployment diagram illustrating this setup.

The architecture defines a typical three tier organization. The presentation tier is represented by the users' web browsers and other client systems, the logical tier by the U-Sem process group and the data storage tier by the plug-in store. In order to satisfy the scalability and availability requirements the logical and data storage tiers are distributed on multiple physical devices. U-Sem processes are replicated on several processing nodes to form a cluster. A load balancer situated between the cluster nodes and the client devices is responsible to distribute the load amongst the nodes. In case of a failure of a processing node, the load balancer no longer sends client request to it.

The database is also distributed. The storage is divided in several pieces based on the owner of the installed plug-ins. Then, a node is assigned to each piece of data. Each node is responsible to store and provide access to the data of the assigned pieces. For extra security each device can be accompanied by another one which serves as a backup. There is also a load balancer between the logical and data tier which is responsible to redirect requests to the data node that is responsible to sore the data for the required user. Additionally, in case of a failure of a data node the load balancer is responsible to start sending request to its backup node and thus, shorten the time of unavailability.

In order to satisfy the security requirements, all communications are managed by firewalls placed before the load balancers. For clarity reasons they(the firewalls and load balancers) are displayed in the same components in the model but they can also be deployed on different processing units as well.

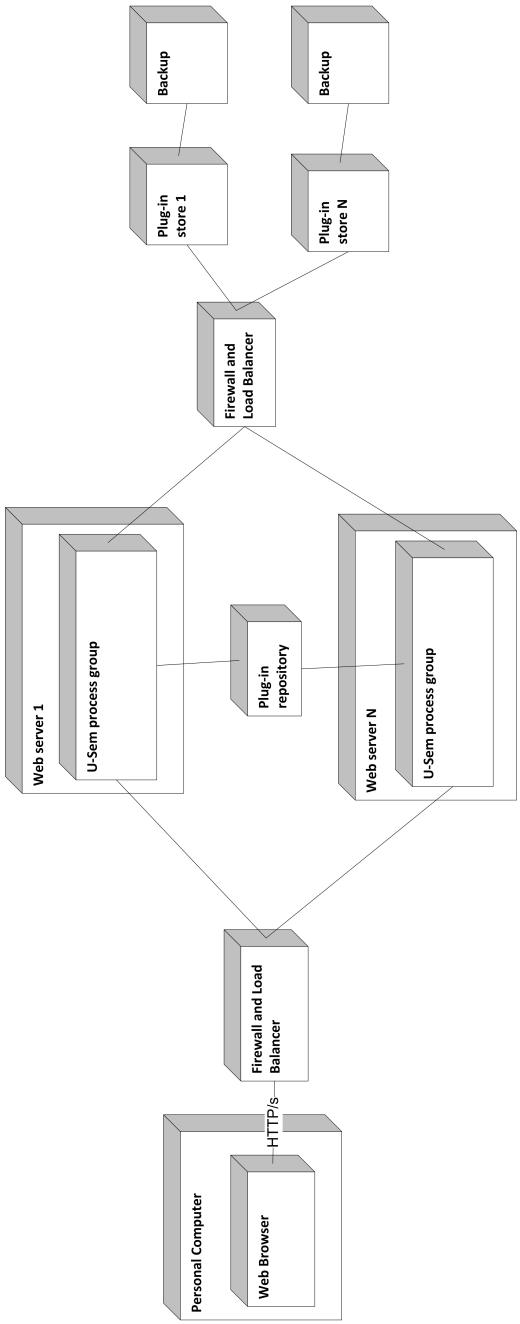


Figure 3.16: Deployment diagram illustrating the high availability setup of U-Sem

3.4 Implementation

We implemented the proposed architecture in order to evaluate its applicability and capabilities. This section describes the main steps we performed during the implementation of the system.

First, we had to choose which OSGI implementation to use. Nowadays there are several vendors that provide implementations. The most popular are: Equinox, Felix and Knopflerfish [10]. Theoretically, they all strictly implement the OSGI standard and therefore, there should be little difference. However, we choose Equinox because it seemed more matured and more widely used compared to the others. Moreover, Equinox is highly integrated in the popular Eclipse IDE. This enables scientist to use the out-of-the box functionality for creating plug-ins in Eclipse.

Secondly, we had to decide the points where U-Sem can be extended by providing custom functionality from plug-ins. Looking at the requirements, we identified that scientist has to be able to provide custom workflow functions and entire workflow definitions. As explained earlier, in OSGI this points for extension are represented as java interfaces or classes. For providing custom workflow functions scientists have to use the *RGLFunction* class. The situation with the workflows was more complicated since they are represented as resource(xml) files. OSGI does not provide direct way for providing custom resource files from plug-ins. In order to overcome this problem, we defined a new class called *WorkflowTemplate* which acts as a bridge and enables the workflow engine and other components to access workfow definition files provided by custom plug-ins.

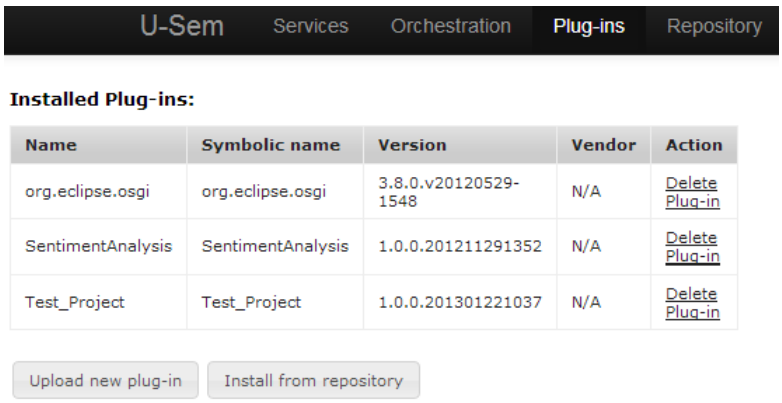


Figure 3.17: User interface for plug-in management.

Next, we provided a very simple implementation of a plug-in repository. It represents a simple web application which stores the plug-ins locally into the file system of the web server where it is deployed. The implementation provides REST interface for retrieving the list of available plug-ins and providing the contents of a selected plug-in. We also implemented very simple user interface which enables scientists to upload their plug-ins.

We continued by implementing all the components defined in the proposed architecture of U-Sem. In order to implement the endpoints(user interface) we used

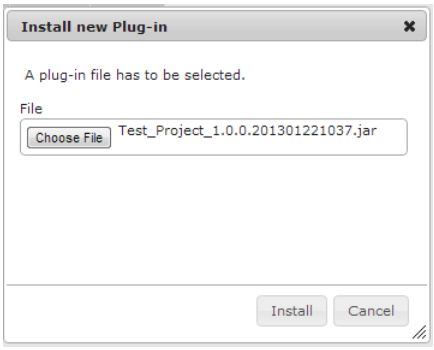


Figure 3.18: User interface for uploading and installing plug-ins in U-Sem.

the jQuery UI[1] and Bootstrap[2] technologies. Figure 3.17 represents the endpoint for viewing all installed plug-ins. The detailed information about all installed plug-ins is represented in a table. Each row has a "Delete" button which provides access to the functionality for removing plug-ins. At the bottom of the view, there are two buttons that lead to the endpoints for uploading a plug-in depicted in figure 3.18 and the endpoint for browsing and installing functionality from the repository depicted in figure 3.19.

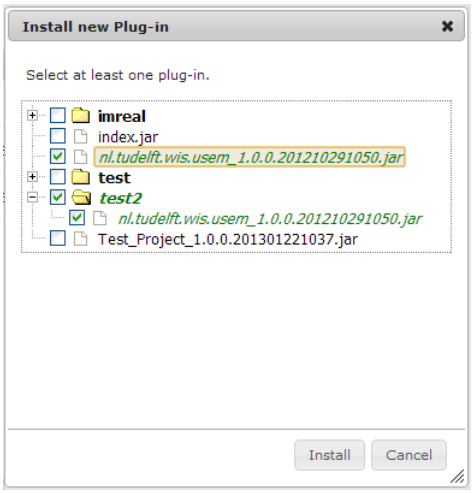


Figure 3.19: User interface for browsing and installing plug-ins from the plug-in repository.

Finally, we constructed a Maven build script that packs all source files into deployable components(war files) that can be directly deployed to a web server. The entire system is built into the following deployable files:

- *rdfgears.war* providing the workflow engine.
- *pluginmanagement.war* providing the functionality for managing plug-ins.

[1] http://jqueryui.com/
[2] http://twitter.github.com/bootstrap/

- *localPluginRepo.war* providing the implementation of the simple plug-in repository.

3.5 Evaluation

After successfully implementing the system, in this section we evaluate whether the system complies with all functional and non-functional requirements defined at the beginning of this chapter.

3.5.1 Functional requirements

In order to verify that all functional requirements are fulfilled we performed experiments and executed each of the defined scenarios.

However, in order to be able to test the scenarios we needed a sample component. We constructed a component that encapsulates all functionality needed to perform the "Sentiment analysis" service. This functionality was initially hardcoded into the workflow engine. Using the tools provided by the Eclipse IDE to create the new plug-in, we removed all functionality and resource files from the workflow engine and put them into the newly created plug-in. The only additional thing that we had to do was to register the provided functionality. In Equinox this is done in the *Application* class which is executed every time the plug-in is loaded. At the end, we exported the plug-in into a traditional java jar file.

Once we had the plug-in needed for the experiments, we tried to execute each of the functional scenarios. Using the user interface we were able to install the plug-in and use the services provided by it(UC1). We were then able to view the installed plug-in in the management user interface and we were also able to successfully remove it from the system(UC3). At the end, we tested the sharing functionality of U-Sem. We were able to successfully share and install the plug-in through the plug-in repository(UC2). All these experiments proved that all functional requirements has been accommodated by the proposed architecture and implementation of U-Sem.

3.5.2 Non-functional requirements

We believe that the proposed architecture will also satisfy the nonfunctional requirements of the system for the following reasons:

- *Availability* - On one hand, the proposed architecture enables clients to load and manage their custom functionality dynamically. There is no need for the system to be restarted in order the changes to take place and thus, the availability of the system is not compromised.

  On the other hand, in production setup, all operations are performed by clusters which means that in case of hardware or software failure of a cluster node the other nodes will continue to operate and the entire system will continue to be available. Additionally, availability in case of a storage node failure is ensured by redirecting all requests to its backup. Cluster nodes can also be placed in different physical locations to handle situation where an accident(e.g. network or electrical failure) in one data center can cause all devices to fail.

Load balancers can be considered as a single points of failure. However, we solve this problem by also providing clusters of load balancers which are accessed through DNS load balancing [22].

- *Scalability* - Another non-functional requirement for the system is to be able to gradually scale for supporting more users by just adding new hardware components. The proposed architecture satisfies this requirement because in order to scale one should just add new nodes to the backend cluster and/or add new nodes to the data storage cluster. The system can be scaled up to the point where each workflow is executed by different processing node and the plug-ins for each scientist are stored on a different storage node.

- *Isolation* - All components of each users are stored at separate place in the file system. There system keeps track that there is no overlapping between the storage spaces. Plug-ins for each user are loaded by a separate instance of the OSGI framework which acts as a barrier and allows users to operate in isolation from one another. Additionally, the users have the full control on the plug-ins that are installed in their storage space. If a new version of a shared plug-in is released the users have to install it manually from the plug-in repository and thus, exercise control over the version of the plug-ins they are using.

- *Security* - Firstly, using the security infrastructure of Java and OSGI, the system ensures that the security policy specified by the administrators of the system is enforced on all custom code that is running in the system. Secondly, the architecture ensures that the custom functionality of the users' is protected from unauthorized access. All connections between the user and the backend can be encrypted through HTTPS and thus, protected against eavesdropping and man-in-the-middle attacks. The system's communication channels are also secured by firewalls. Finally, as required each storage node is backed up in case a device fails.

- *Modularization* - This non-functional requirement is accommodated by the dynamic component model provided by OSGI. It address the limitations of the standard Java module system discussed in section 3.1 and provides information hiding and cross component communication capabilities that enforce better component modularization [11].

3.6 Limitations and Future Work

In this section we identify the limitations of the proposed architecture and we also suggest approaches that can be used to overcome these limitation in the future. We have identified two groups of limitations concerning U-Sem. The first group represents the limitations that are inherited from the usage of OSGI. The second one consists of the limitations concerning the rest of the U-Sem's architecture.

Most of the limitations concerning OSGI originate from the potential vulnerabilities of running external code which the security mechanism fails to fully address. The authors of [14] have studied in details the potential vulnerabilities of OSGI. These vulnerabilities can be grouped into the following categories:

- *Vulnerabilities on operating system level* - This kind of vulnerabilities result from the fact that it is possible that a plug-in runs malicious native code using the Java JNI. Native code is not managed by the JVM and thus, the security policy is not applied. [21] proposes a portable solution for sandboxing of Java's Native Libraries without modifying the JVM's internals which might be used for overcoming these vulnerabilities.

- *Vulnarabilities on OSGi platform level* - This kind of vulnerabilities are related to weaknesses in the OSGi run-time. [14] suggests an approach for overcoming them by adding additional security checks in the OSGi implementation.

- *Vulnarabilities on JVM level* - This vulnerabilities can be further divided into categories [15]:

  - *lack of isolation* - Even though components for each user are loaded through separate OSGI instances, on JVM level *java.lang.Class* objects and static variables are shared by all plug-ins. A malicious bundle can interfere with the execution of other bundles by altering static variables or obtaining lock on shared objects.
  - *lack of resource accounting* - In OSGI each plug-in is loaded with a separate class loader. However, JVM does not perform resource monitoring on a per class loader basis. Therefore, in case of the overuse of resources(CPU, memory), it is impossible to identify the faulty bundle and stop its execution.
  - *failure to terminate a bundle* - If the system recognizes a bundle as misbehaving and wants to stop its execution it might fail if methods of the bundle are being executed at that point. Moreover, a malicious code can run an infinite loop in the Java *finalize* method and thus prevent memory reclamation.

  [15] proposed and approach for overcoming these vulnerabilities. They have designed I-JVM, an extension of the Java Virtual Machine which provides functionality for component isolation and termination in OSGi.

At this point, U-Sem is aimed to be used by scientists from a single organization. Therefore, the components will only be reused within the organization which limits the possibility for any external person adding plug-ins into the system. Therefore, exploitation of the discussed vulnerabilities on purpose is not so likely. Therefore, we believe that this limitations does not pose a significant threat for U-Sem. However, all these vulnerabilities has to be addressed if in future the system is to be extended to enable access from unverified scientists.

Apart from the inherited limitations from OSGI, another limitation of the architecture comes from the fact that all plug-ins are reloaded before each execution of a workflow. This is done in case the installed plug-ins are changed between two workflow executions. The consequence is the caused delay for reloading the components before each workflow execution. Even though this delay might be considered minute for most scenarios it can cause reduced performance in the case where large number

of workflows with short execution time has to be executed in a short amount of time. In future, this limitation can be addressed by establishing communication mechanism between the workfow engine and the plug-in management component. In this way when a change is made to the plug-ins, the workflow engine marks them as dirty and only then it finds a suitable moment to reload and apply the changes. Additionally, [16] also proposes approach for reducing the time needed to reload the components in OSGI.

Bibliography

[1] G. Pour, Component-Based Software Development Approach: New Opportunities and Challenges, Proceedings Technology of Object-Oriented Languages, 1998. TOOLS 26., pp. 375-383.

[2] G. Pour, Enterprise JavaBeans, JavaBeans and XML Expanding the Possibilities for Web-Based Enterprise Application Development, Proceedings Technology of Object-Oriented Languages and Systems, 1999, TOOLS 31, pp.282-291.

[3] G.Pour, M. Griss, J. Favaro, Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles - Patterns for Success, Proceedings of Technology of Object-Oriented Languages and systems, 1999, pp.419 - 419.

[4] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053-1058, 1972.

[5] H Jifeng, X Li, Z Liu, Component-based software engineering, Theoretical Aspects of Computing-ICTAC 2005

[6] Cai, X. and Lyu, M.R. and Wong, K.F. and Ko, R, Component-based software engineering: technologies, development frameworks, and quality assurance schemes, Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific

[7] Z Chen, Z Liu et al, Refinement and Verification in Component-Based Model Driven Design, Report of International Institute for Software Technology, 2007

[8] Kung-Kiu Lau and Zheng Wang, Software Component Models, Software Engineering, IEEE Transactions on, 2007

[9] E. Bruneton, T. Coupaye, and J. Stefani, The Fractal Component Model, ObjectWeb Consortium, Technical Report Specification V2, 2003

[10] OSGi Alliance. http://www.osgi.org

[11] Andre L. C. Tavares, Marco Tulio Valente, A Gentle Introduction to OSGi, ACM SIGSOFT Software Engineering Notes, 2008

[12] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006.

[13] Decker, G. and Barros, A., Interaction modeling using BPMN, Business Process Management Workshops, 2008

[14] P. Parrend and S. Frenot. Security benchmarks of OSGi platforms: toward hardened OSGi. Software: Practice and Experience, 39(5):471-499, April 2009

[15] Nicolas Geoffray, Gael Thomas, Gilles Muller, Pierre Parrend, Stephane Frenot, and Bertil Folliot: I-JVM: A Java Virtual Machine for Component Isolation in OSGi. In: DSN 2009

[16] Vladimir Nikolov, Rudiger Kapitza, Recoverable Class Loaders for a Fast Restart of Java Applications, Mobile Networks and Applications, 2009

[17] S Srivastava, M Hicks, Modular information hiding and type-safe linking for C, Software Engineering, IEEE Transactions on 2008

[18] J Eder, G Kappel, M Schrefl, Coupling and cohesion in object-oriented systems, 1994

[19] Rozanski, Nick and Woods, Software systems architecture: working with stakeholders using viewpoints and perspectives, Addison-Wesley Professional, 2011

[20] E Woods, N Rozanski, The System Context Architectural Viewpoint, 2009

[21] Sun, Mengtao and Tan, Gang, JVM-Portable Sandboxing of Java's Native Libraries, Computer Security–ESORICS 2012

[22] Membrey, Peter and Hows, David and Plugge, Eelco, DNS Load Balancing, Practical Load Balancing, 2012

[23] Dean D, Felten EW, Wallach DS. Java Security: From HotJava to Netscape and beyond. SP'96: Proceedings of the 1996 IEEE Symposium on Security and Privacy. IEEE Computer Society: Washington, DC, U.S.A., 1996; 190.

[24] Gong L, Ellison G, Dadgeforde M. Inside Java 2 Platform Security-Architecture, API Design, and Implementation (2nd edn). Addison-Wesley: Reading, MA, 2003.

Chapter 4

Conclusions and Future Work

This chapter gives an overview of the project's contributions.

4.1 Contributions

4.2 Conclusions

4.3 Discussion/Reflection

4.4 Future work

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

foo: ...

bar: ...

Appendix B

Requirements and Guidelines

This chapter details some requirements and guidelines for MSc theses submitted to the Software Engineering Research Group.

B.1 Requirements

B.1.1 Layout

- Your thesis should contain the formal title pages included in this document (the page with the TU Delft logo and the one that contains the abstract, student id and thesis committee). Usually there is also a cover page containing the thesis title and the author (this document has one) but this can be omitted if desired.
- Base font should be an 11 point serif font (such as Times, New Century Schoolbook or Computer Modern). Do not use sans-serif fonts such as Arial or Helvetica. *Sans-serif type is intrinsically less legible than seriffed type*
- The final thesis and drafts submitted for reviewing should be printed double-sided on A4 paper.

B.1.2 Content

- The thesis should contain the following chapters:
  - Introduction.
    Describes project context, goals and your research question(s). In addition it contains an overview of how (the remainder of) your thesis is structured.
  - One or (usually) more "main" chapters.
    Present your work, the experiments conducted, tool(s) developed, case study performed, etc.
  - Overview of Related Work
    Discusses scientific literature related to your work and describes how those approaches differ from what you did.
  - Discussion/Evaluation/Reflection
    What went well, what went less well, what can be improved?

  - Conclusions, Contributions, and (Recommendations for) Future Work
  - Bibliography

B.1.3 Bibliography

- Make sure you've included all required data such as journal, conference, publisher, editor and page-numbers. When you're using BIBTEX, this means that it won't complain when running `bibtex your-main-tex-file`.
- Make sure you use proper bibliographic references. This especially means that you should avoid references that **only** point at a website and not at a printed publication.

  For example, it's OK to add a URL with the entry for an article describing a tool to point at its homepage, but it's not OK to just use the URL and not mention the article.

B.2 Guidelines

- The main chapters of a typical thesis contain approximately 50 pages.
- A typical thesis contains approximately 50 bibliographic references.
- Make sure your thesis structure is balanced (check this in the table of contents).

  Typically the main chapters should be of equal length. If they aren't, you might want to revise your structure by merging or splitting some chapters/sections.

  In addition, the (sub)section hierarchies with the chapters should typically be balanced and of similar depth. If one or more are much deeper nested than others in the same chapter this generally signals structuring problems.
- Whenever you submit a draft of your thesis to your supervisor for reviewing, make sure that you have checked the spelling and grammar. Moreover, *read it yourself at least once from start to end, before submitting to your supervisor*.

  Your supervisor is not a spelling/grammar checker!
- Whenever you submit a second draft, include a short text which describes the changes w.r.t. the previous version.

Appendix C

Prioritization questionnaire

In this appendix we provide the questionnaire used for the prioritization of the requirements.

TODO