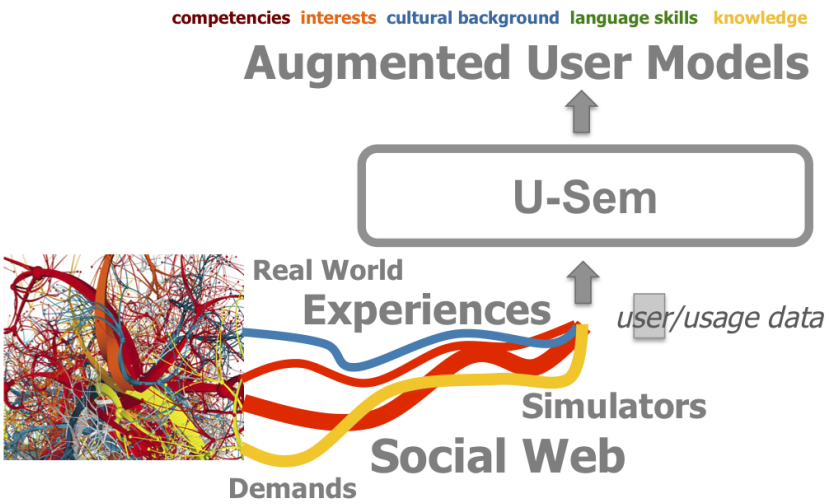
U-Sem, a framework for augmented user and context modeling

Master's Thesis



Borislav Todorov

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

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 modeling

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 6, 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.

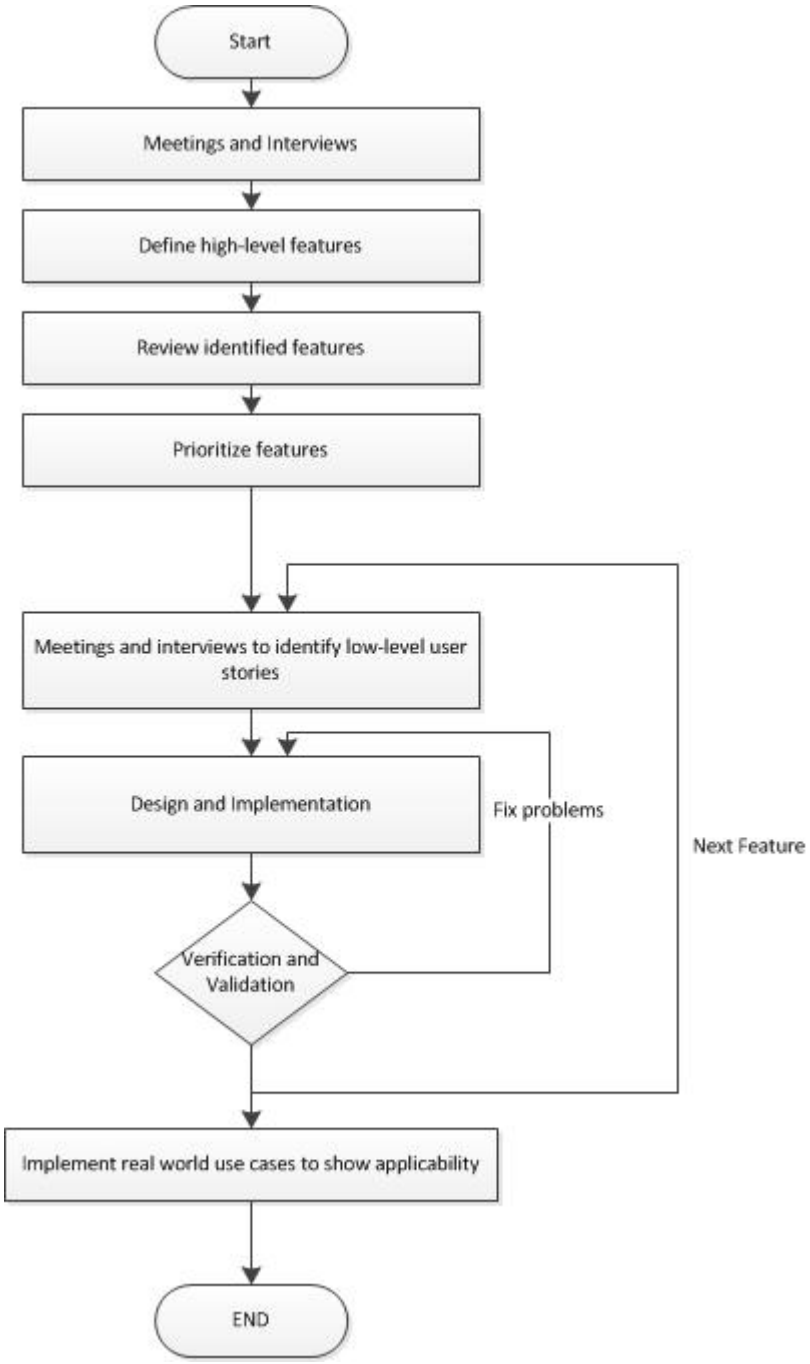


Figure 1.1: Flow chart defining the sequence of actions taken in order to complete the project.

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.

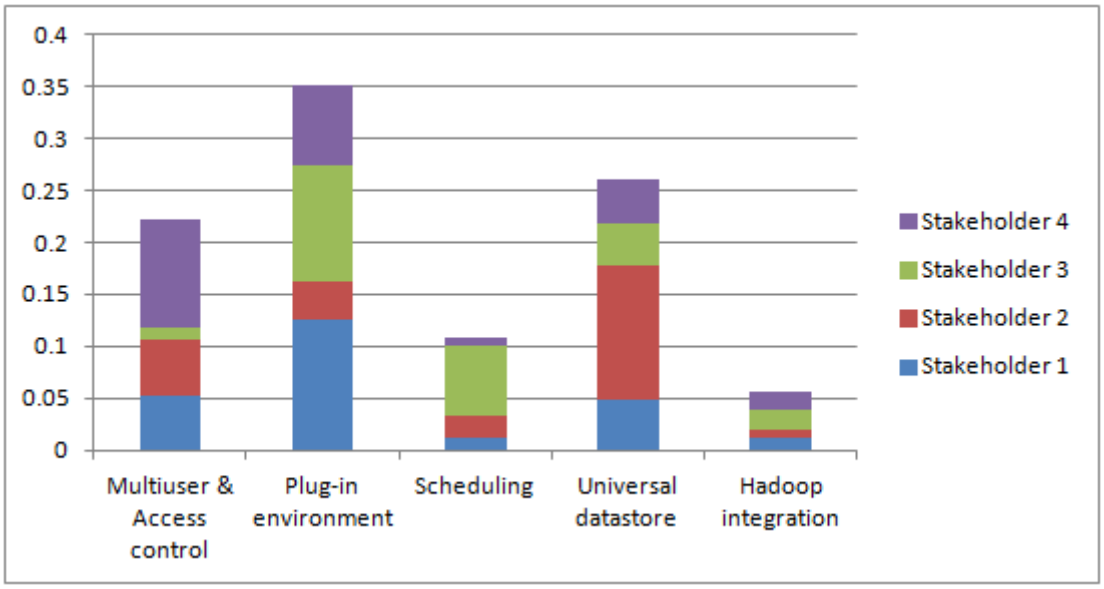


Figure 2.1: The value distribution of the 5 requirements in the U-Sem project.

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.

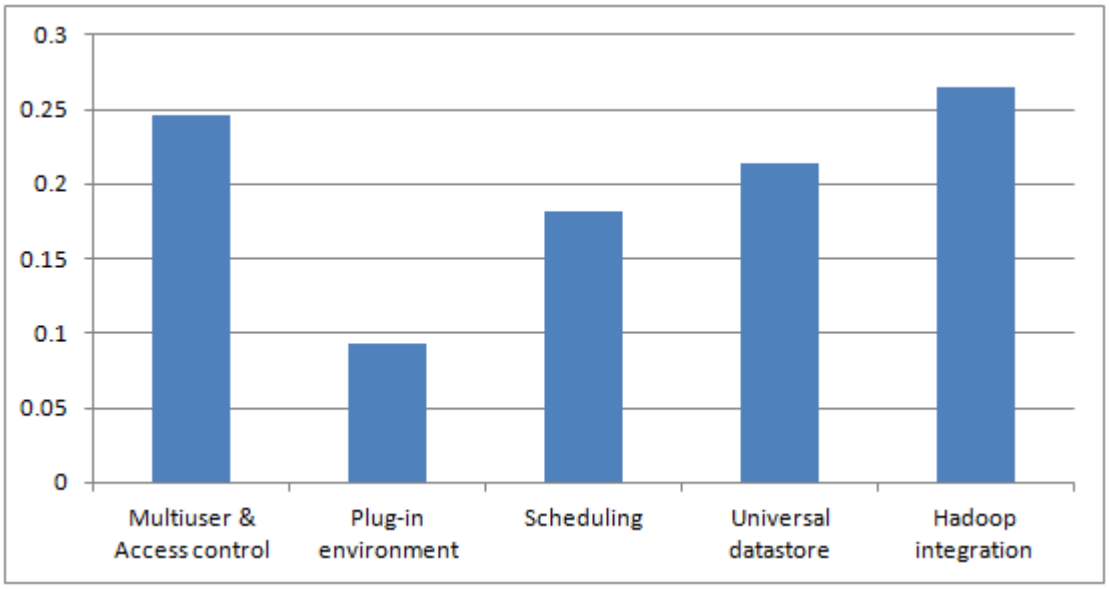


Figure 2.2: The cost distribution of the 5 requirements in the U-Sem project.

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

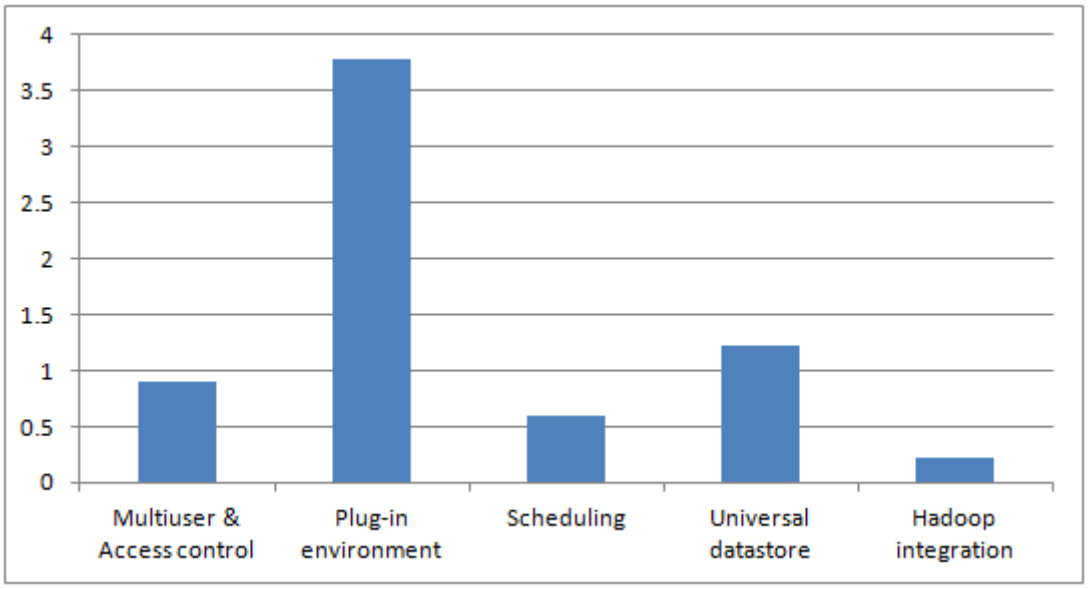


Figure 2.3: This diagram shows the comparison of the value/cost ratio of the requirements.

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 covers the architecture of the dynamic component model of U-Sem which enables scientists to dynamically load and use custom functionality.

3.1 Problem definition

All of initial interviews with potential users of the system reviled that the nature of their work is extremely dynamic. The main responsibility of the scientists is to come up with new algorithms and approaches for user modelling. As a result they are continuously producing new software code that implements this algorithms. After each production cycle, this program code has to be put into U-Sem so that it is available for testing, demonstration and evaluation purposes.

We performed additional interviews with stakeholders in order to reveal how this process is currently done. Scientists reported that they have on their disposal only the capabilities of the workflow engine. However, it provides no functionality that enables to plug 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 become part of the workflow engine. 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 functionality requires a lot of time and knowledge. This is because in order to add the new functionality 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 process: checking out the source code from the software repository, writing 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 creates a serious overhead and discomfort.

- In order to add new functionality to the system one has to stop the web server where the system is deployed, replace the program entities of the system and start the server again. The problem is that during the time the server is down all previously created services are unavailable. This is a major problem for everyone that is using the system during that time.

- Another major disadvantage is that the training period for new scientist working with the system is significant because of all the additional knowledge required. This may easily cause project delays and missed deadlines.
- Multiple scientists adding functionality simultaneously result in missing functionality. **Figure 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 scientists will be lost when the second deploys his version.
- It is hard to verify what additional functionality is added to the system and therefore, additional documentation is required. This problem becomes more serious when there are more people are working on the project simultaneously and it is hard to track the changes in functionality.

But this is not everything. 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 a key property for improving extensibility, comprehensibility, and reusability in software projects [4]. In this case, engineers can only rely on the modular 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. 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 **Cai**.

When designing the U-Sem architecture we considered all these pitfalls of the current approach. However, before going to the architecture description, for clarity and easier validation, in the next subsection, we provide a more formal representation of all user requirements.

3.1.1 Functional Scenarios

In this section we are formally identify the functional requirements which define the main interactions between the scientists and the system. This compact representation of the requirements also makes validation an evaluation clearer.

- **Adding custom functionality** - This is the main scenario regarding the dynamic component model. Scientists has to be able to extend U-Sem by adding custom functionality on demand. They has to be able to be able to compose the custom functionality independently from the system, add the produced functionality to U-Sem during the execution time of the system and/or if desired share it with other scientists.
- **Use functionality shared by other scientists** - Scientists has to be able to reuse custom functionality provided by other scientist.

- **Manage loaded functionality** - Users has to able to manage all functionality already added to the system. This includes to be able to view a list of all added functionality. And secondly they have to be able to remove any functionality from the provided list.

3.1.2 Non-functional requirements

This section defines the main quality scenarios of the system that model how the system should react to a change in its environment.

- Availability - adding custom functionality should not affect other users at all.
- Insulation - scientists not being affected by any future changes to the reused components.
- Privacy - As a user, I don't want other users to see what is my custom functionality and use them.
- Security

3.2 Approach

After a detailed investigation of the requirements we reached the conclusion that the core of the problem lies firstly in the poor separations of concerns(modularization) and secondly 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 help to overcome these problems. The review showed 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 a system out of different components [5]. It is called Component-based software engineering.

Next subsection discusses the basic idea and advantages that this approach brings. Then we 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 to assemble them together with a well-defined software architecture [1]. This software development approach improves on the traditional approach since applications no longer has 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 1**, where components can be checked out from a component repository, and assembled into the desired software system.

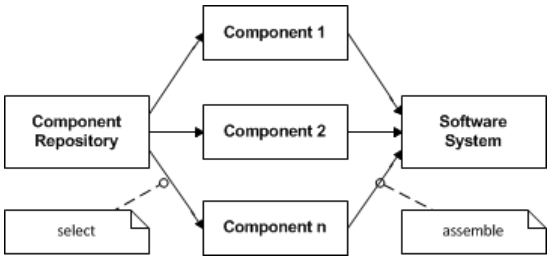


Figure 3.1: Component-based software development **cite 335**

The main benefits that Component-based software development brings are: 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

A component model is the architecture of a system or part of a system that is built using 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 **fig** interfaces can be considered as points where custom functionality provided by another component can be plugged in.

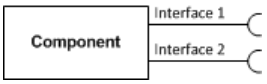


Figure 3.2: Component interfaces

Components are functional units providing functionality by implementing interfaces. As you can see on **figure** components provide features by implementing the provided interface. 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

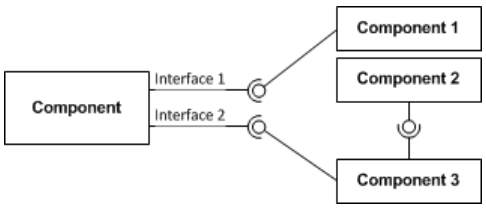


Figure 3.3: Components implementing interfaces

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

Previous sections clearly show that integrating component model in the architecture of U-Sem will solve the design problem of will fulfil the requirements of the customers. Nowadays, there are many implementation of component models and it might be wiser to reuse one of them instead of implementing everything from scratch. The reasons for doing that are because this implementations are developed for long time, they are popular and widely used. This suggests that they are heavily tested 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. It is essential since the system is used by multiple scientists and any 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 also having to learn a new language is considered as a big disadvantage for the scientists.

We performed an 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]. For 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 extensively 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 is a set of specifications that implement the component model and defines 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 implements the main components of the standard component model which are discussed in the previous section as follows:

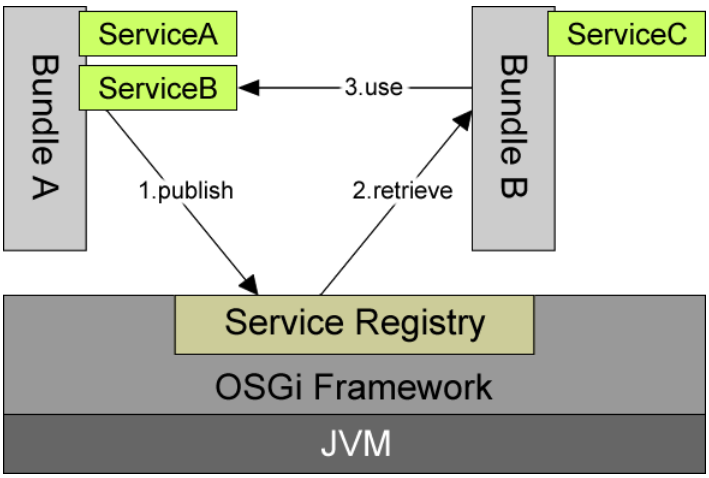


Figure 3.4: OSGi Service Registry [11]

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 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. OSGI also provides 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 required Interfaces explained 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 is provided by the Service Registry, which keeps track of the services registered within the framework. As illustrated in **Figure 1** when a bundle is loaded it registers all the services that it implements(step 1). As soon as it 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.

3.3 Architecture

This section describes the architecture of U-Sem regarding the dynamic component model. In order to provide more efficient description we have provided 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.

3.3.1 Context View

This section discusses the runtime environment of U-Sem. As explained in the introduction section initially the system only communicated with providers of semantic content and the clients which execute the services. However, introducing the dynamic component model of U-Sem we bring to entities on the stage. **Figure** illustrates the updated runtime environment of U-Sem.

Scientists As one can see on the diagram the system also communicates with the scientists in order to enable them to plug-in new functionality. When scientists want to add new functionality they first have to build a component that encapsulates the new logic. Then they upload the component to U-Sem and it will be installed in the components space of the scientist. Once installed, the scientist can start using the

newly added functionality. Scientists can also communicate with U-Sem in order to manage the existing components loaded into the system. They can view the list of all available components and if needed they can also update or even remove them.

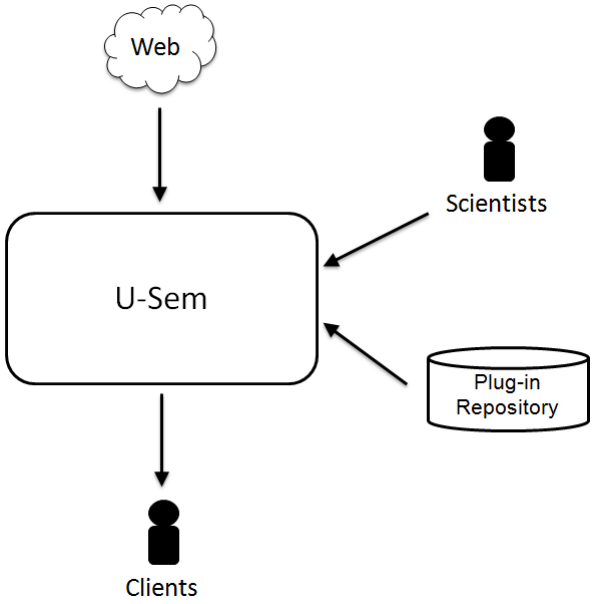


Figure 3.5: Runtime environment in U-Sem

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 their components with one another.

By using the Plug-in repository scientists are able to share 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 entirely new component it is installed to U-Sem, shared and then all other scientists are able to use it. However, this 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. Otherwise, when they decide that are ready for the change, they install and use the new release. The benefit from this 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 weather or when to move to the new releases of the components in use. This approach is already adopted in the industry **TODO P2, Eclipse etc.**

3.3.2 Process modelling

Once we have already identified the new actors(scientists) and external systems(plug-in repository) in the runtime environment of U-Sem we have to define how they interact between each other. In this section we will use the Business Process Management Notation(BPMN) [12] to model the business processes that defines the interactions needed in each of the use cases that regard the dynamic component model feature of U-Sem. It also enables to model activities, decision responsibilities, control and data flows. The decision to use BPMN to model the interactions is based on its relative 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 Component process

Creating new functionality for U-Sem is the most important use case regarding the dynamic component model feature. We modelled this use case as the *Loading new components* business process. **Figure** 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 Plug-in repository) which are illustrated in separate BPMN pools. When a scientist wants to create new functionality for U-Sem, he/she firsts writes the source code, providing all required resources and implementing the desired U-Sem interfaces(the component interfaces discussed in previous sections). Then everything has to be build and encapsulated into a component. Once the component is ready, scientists can directly upload it to U-Sem if the component is only for private use. When U-Sem receives a component it is responsible to installed it into the scientist's component storage please 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 confirmation message is send to the scientist.

Reuse shared components

As already explained, U-Sem also enables scientists to reuse components shared by other scientist. This use case is modelled into the *Reuse shared components* business process which is illustrated in **figure**. Again we have three participants in this process(U-Sem, Scientists and 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 components. Secondly, he/she contacts U-Sem providing information about the desired component/s.

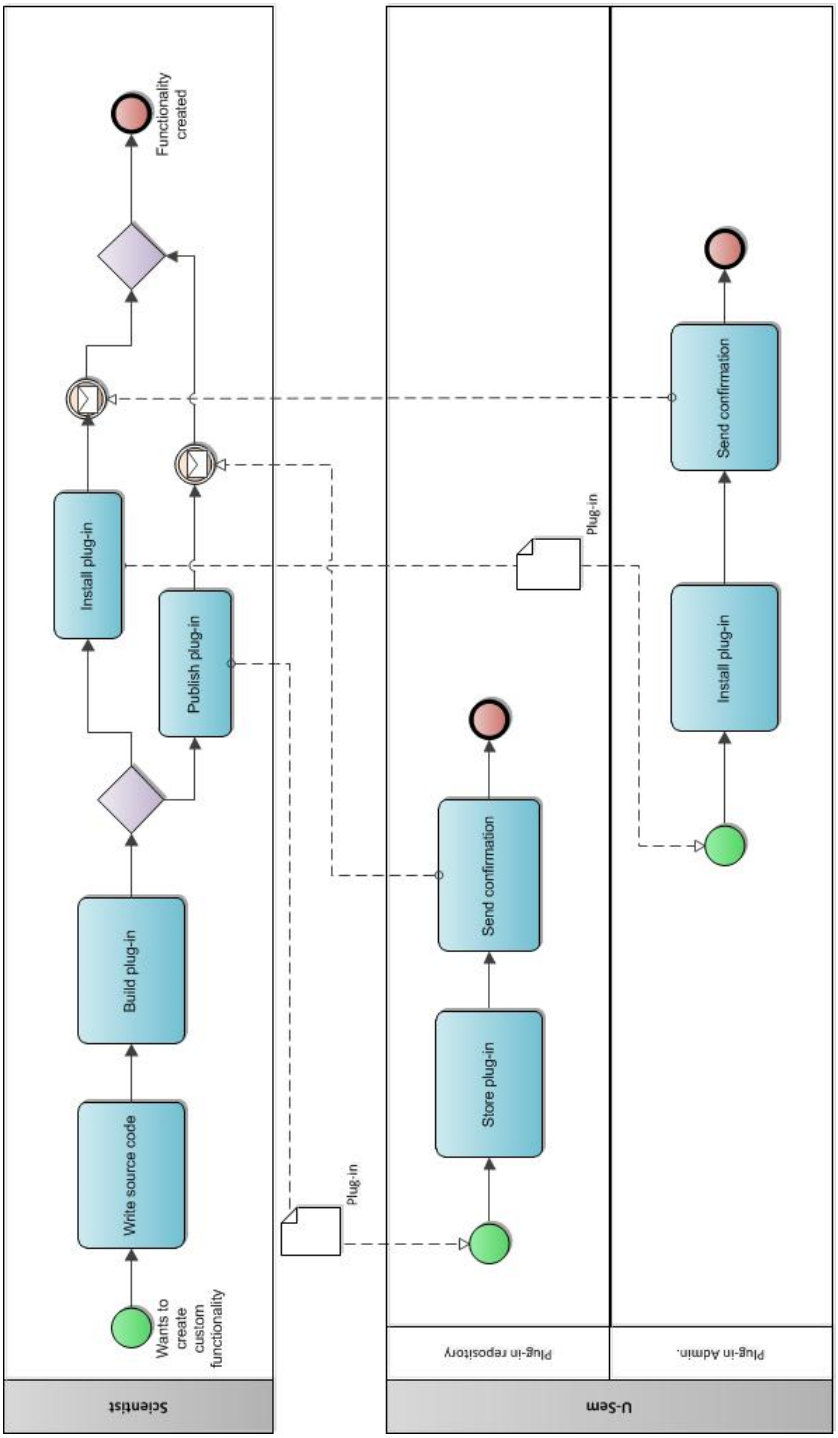


Figure 3.6: Business model describing the process for adding new plug-in to U-Sem

U-Sem then contacts the plug-in repository in order to get the components. Finally, the provided components are stored into the private space of the scientist and a confirmation message is send back.

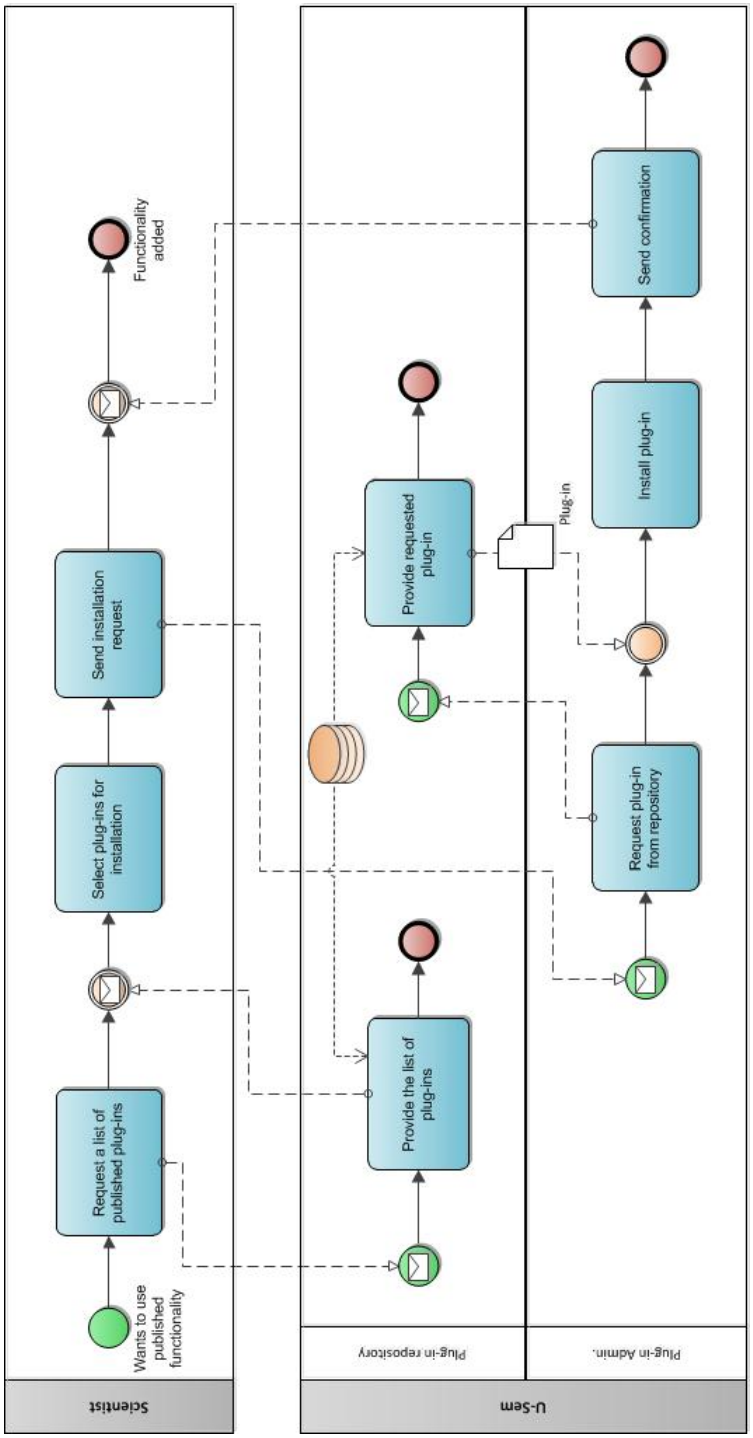


Figure 3.7: Business model describing the process for adding existing plug-in from repository to U-Sem

Component Management

Managing components is also another important use case. Implementing it will enable scientists to view all components installed into U-Sem and if needed remove them.

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

Scientists can monitor the currently installed components at any time by contacting U-Sem. When such request is received, U-Sem is responsible to send back detailed information about all the components. Having this lists, scientist are also able to remove components. In this case scientists have to submit request for removal providing detailed for the component that has to be removed. Upon receiving such request U-Sem is responsible to permanently remove the component form the private space of the scientists and when finished send back a confirmation message.

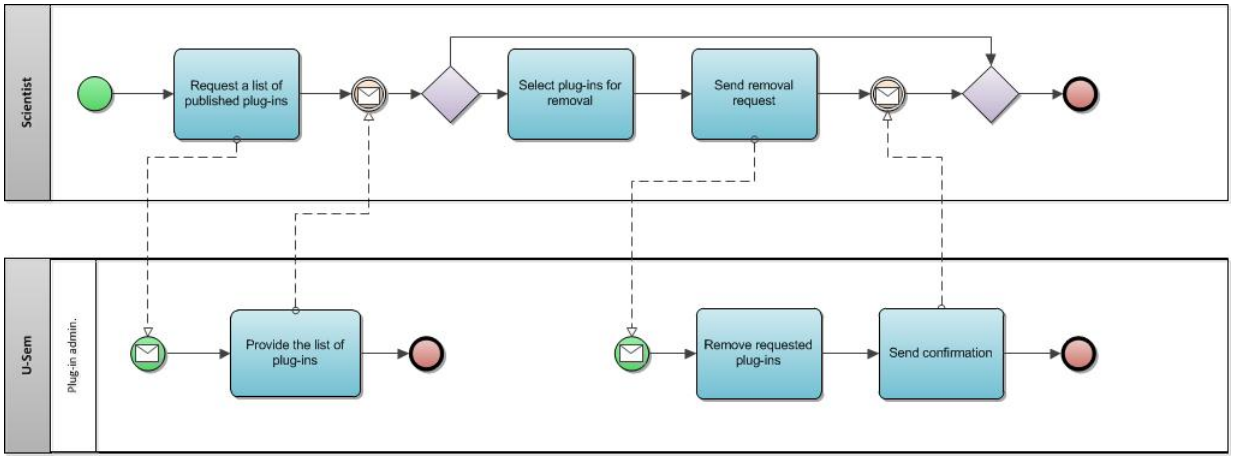


Figure 3.8: Business model describing the process for managing plug-ins in U-Sem

3.3.3 Functional view

After identifying all actors that that are part of the environment of U-Sem and the way they interact with one another, it is now time to discuss the internal structure of U-Sem that accommodates all these interactions. This section defines the architectural elements that provide the functionality of the system. It describes the functional structure of the system including the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them. Taken together, this demonstrates how the system will perform the required functions.

All components that are take part in the dynamic component model functionality can be classified in layers. This three layered organization is illustrated in figure **Figure** and defines the following layers:

- *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 components. Components in this layer are responsible to enforce the security, privacy, etc. policies of the system.

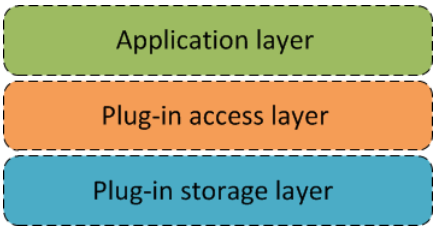


Figure 3.9: Layer organization of U-Sem

- *Application layer* this layer consists of all 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 once.

High-level organization

This section describes the internal structure of the layers and specifies the high level components that build up the feature. **Figure** illustrates this organization and shows how the high-level components are organized into the layers. We have the following components starting from bottom up:**TODO better explanation is needed**

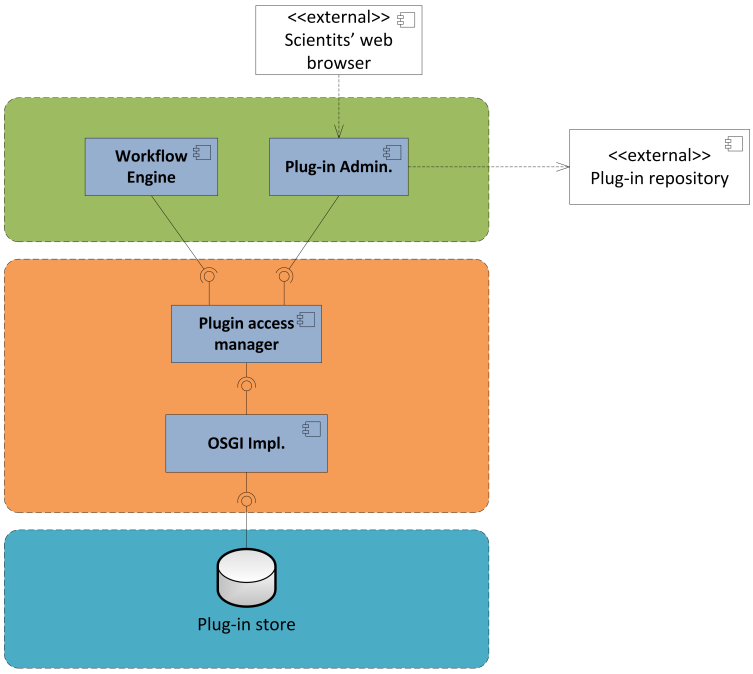


Figure 3.10: Layer organization of U-Sem

- *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. Current implementation of OSGI can only operate with file store and therefore this component is represented by the file system of the operating system. For availability reasons this storage should be **replicated**.

- *OSGI Implementation* - As we already discussed in previous sections, we chose OSGI as a standard for achieving the dynamic component model for U-Sem. It is responsible to keep track of plug-ins life cycle(create, store, ..) and provide access to the services implemented by the different components. It provides API which enables communication 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 ans start of the framework. It is also responsible to enforce the security policy and provide insulation between scientists. It provides API for the higher level components for dealing with services and management of plug-ins. Further decomposition of this component is provided in the next section.

- *Plug-in admin.* is responsible to deal with the administration of the plug-ins. It provides the system's endpoint(UI) for interaction with the scientist scientists. Further decomposition of this component is provided in the next section.

- *Workflow engine* uses the interface provided by the *Plugin access manager* to get list of the available services and when required execute them.

Plug-in admin. module

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

- *List Plug-ins* - This component is responsible to provide functionality that provides the list of all plug-ins that are installed for the particular user. This component has to provide detailed information for each plug-in: its id, name, vendor, etc.

- *Plug-in removal* - This component should enable users to remove(uninstall) 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.

- *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.

- *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.

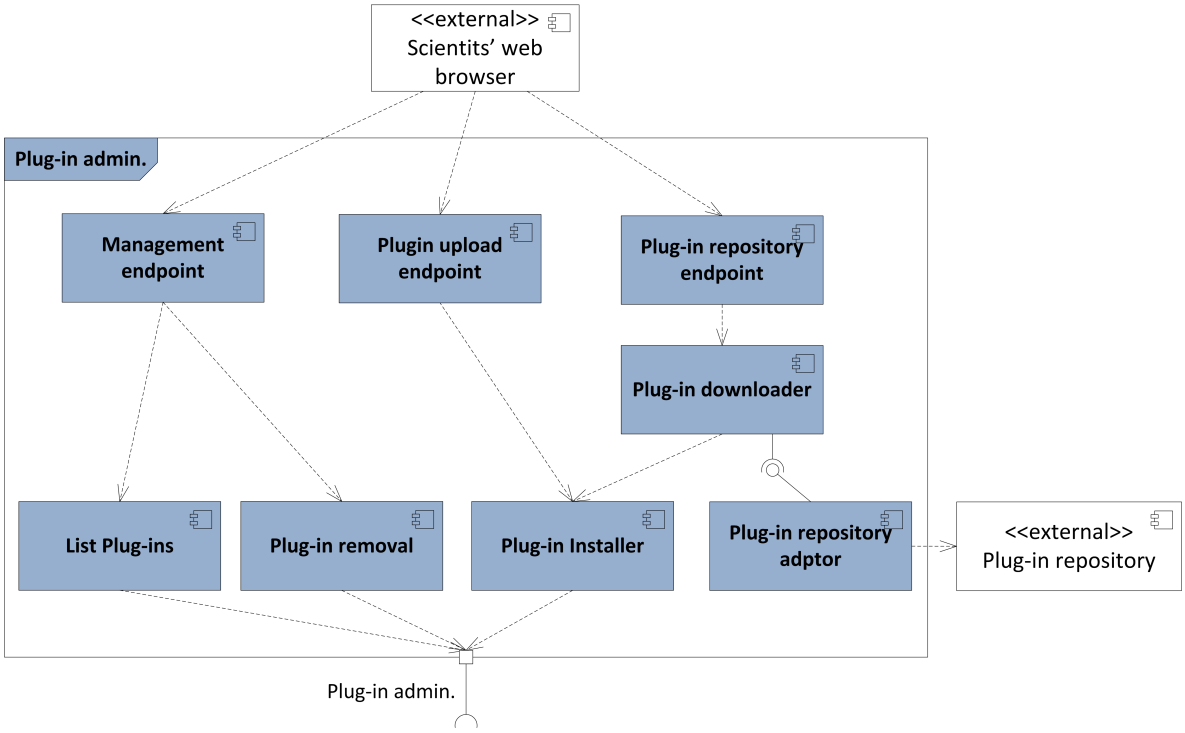


Figure 3.11: Layer organization of U-Sem

- *Plug-in repository adaptor* - This component manages the communication with the plug-in repository. It acts as a level of abstraction. U-Sem has to be able to use different repositories and this component 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 endpoint for the administrators to manage the plug-ins. They can install and delete plug-ins. This component is responsible to store and manage the access to all plug-ins. **Figure** shows the functional decomposition of the Plug-in access manager module. It consists of the following components:

- *OSGI Manager* - This component manages the communication with the OSGI engine. It is responsible start/stop the engine and monitor its lifecycle. It is also responsible to enforce the security policies by setting up the Security Manager options of the engine. It also acts as a level of abstraction over the component

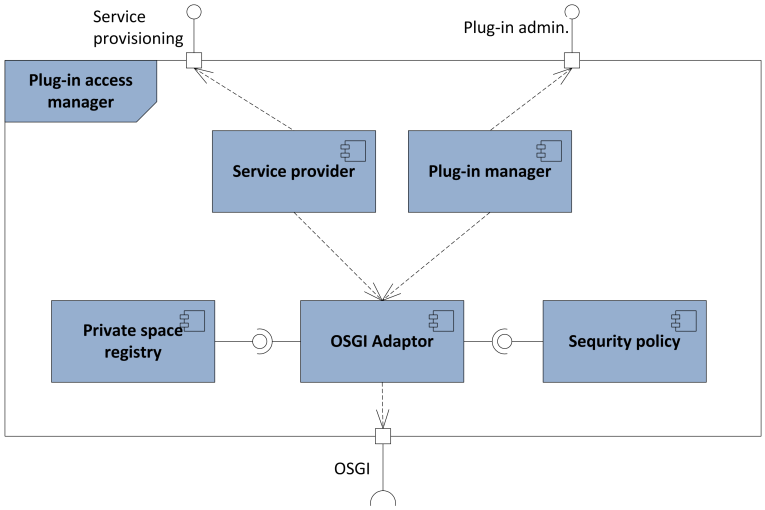


Figure 3.12: Layer organization of U-Sem

engine and in case any change in future is required this is the only component that will be affected.

- *Private space registry* - Keeps track for the storage space and settings for each user. It has to make sure that the storage places are not overlapping.
- *Security policy* - Provides the security policy for each user. It defines the Java operations(file access **...**) that users are allowed to perform.
- *Plug-in manager* - This component is responsible to provide an API that is used by high level components to perform plug-in management tasks.
- *Service provider* - This component is responsible to provide an API that is used by high level components to get the list of available services and load them.

Security

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 execute. We also show how this parallel execution is coordinated and controlled.

Figure **fig** 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. 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

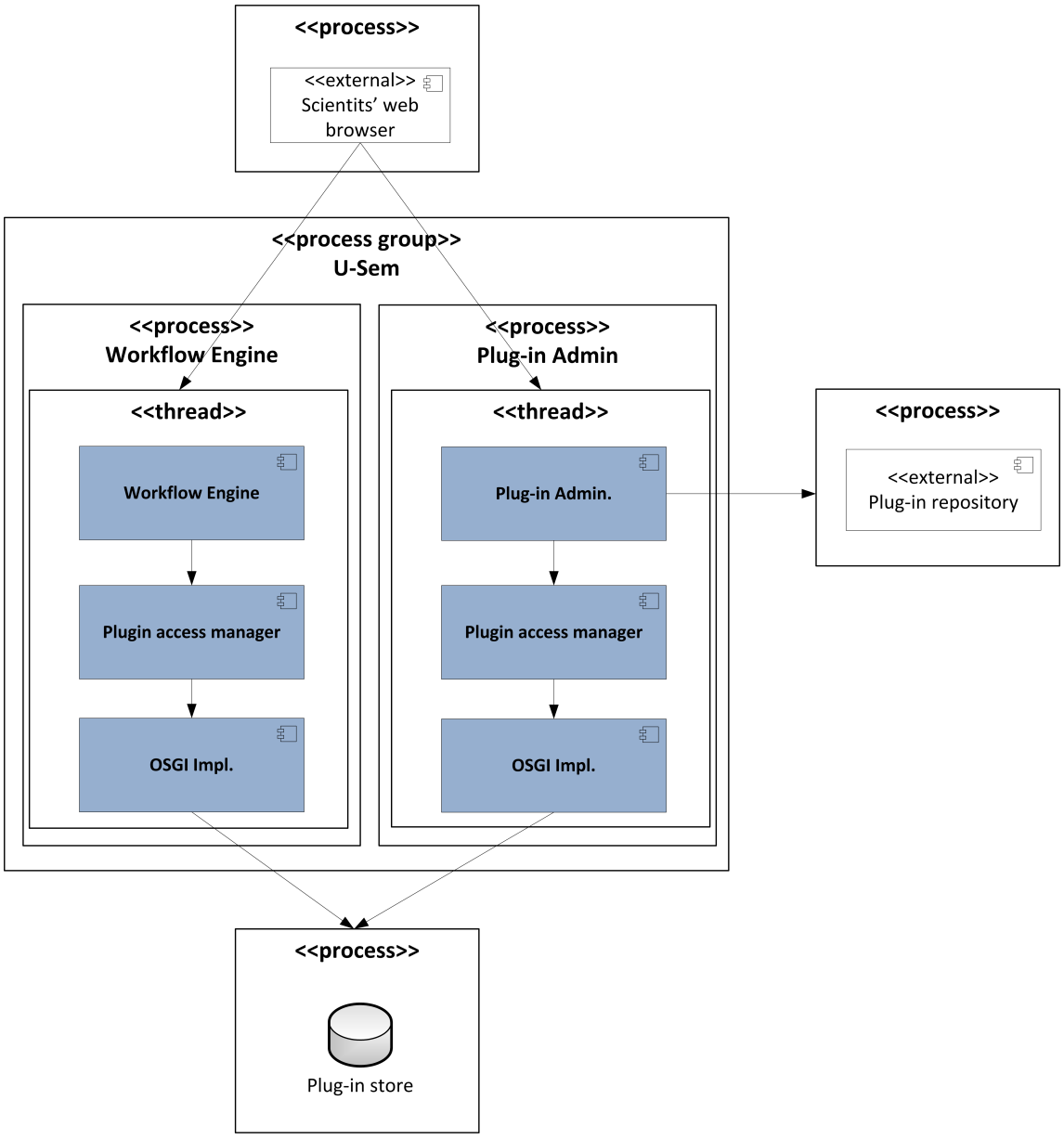


Figure 3.13: Layer organization of U-Sem

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 isntalled plug-ins into the 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 distrupted. All changes will take place the next time a workflow is executed.

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 run.

Due to the nature in which the system operates we had to consider flexibility and scalability. In order to test their work scientists has to be able to easily install and set up U-Sem. In this scenario scalability and performance is not critical however simplicity is crucial. 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 most 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 is to enable scientists to setup the system fast on a single machine and little configuration.

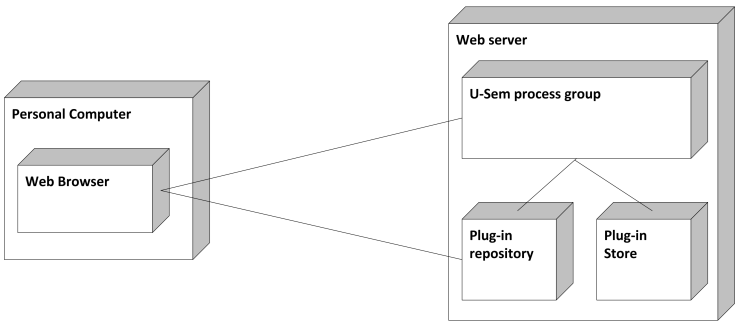


Figure 3.14: Layer organization of U-Sem

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

3.3.6 High Availability setup

This setup aims to cover the scenarious requiring high performance and high availability.

The architecture defines a typical three tier organization. The presentation tier is represented by the user's web browsers and other client systems, the logical tier by the U-Sem backend process 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. The backend process is 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 **Not in picture**. These is also a balancer between the logical and data tier which is responsible to redirect requests to the data node that is responsible for 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 cause little times 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.

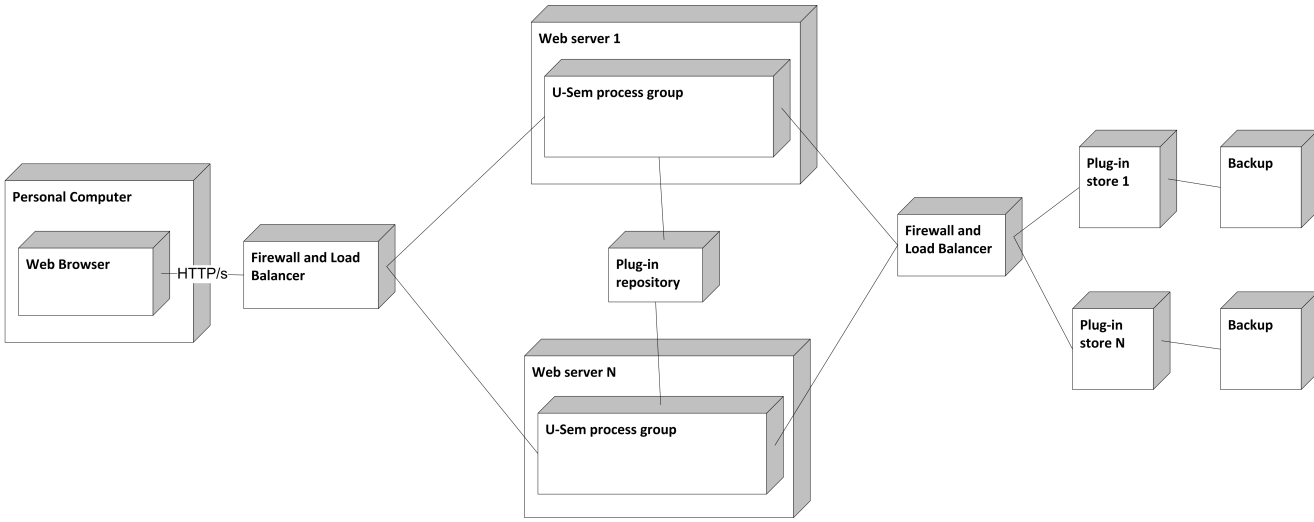


Figure 3.15: Layer organization of U-Sem

3.4 Implementation

In order to prove the applicabilities and capabilities of the system we actually implemented it. This section covers the most interesting steps 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. 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 then the others. Moreover, Equinox is highly integrated in the 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 extendend by providing custom functionality from plug-ins. At this point we identified that scientist has to be able to provide custom rdf gears functions and workflows. As explained earlier, in OSGI this points are represented as java interfaces. For custom functions we 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 identified new class called *WorkflowTemplate* which acts as a bridge and enables the workflow engine and other components to access workfow files provided by custom plug-ins.

Next, we provided a very simple implementation of a plug-in repository. It represents a simple web application which stores the plug-in locally into the file system of the web server where it is deployed. The implementation provides resentful api for retrieving the list of available plug-ins and provide 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 described in the architecture of U-Sem. In order to implement the end points(user interface) we used the JQuery and Bootstrap. Figure **fig** represents the end point for viewing all installed plug-ins. Detailed information about all plug-ins is represented in the 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 illustrated in figure **fig** and the endpoint for browsing and installing functionality from the repository illustrated in figure **fig**.

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

- *rdfgears.war* providing the workflow engine.
- *pluginmanagement.war* providing the functionality for managing plug-ins.
- *localPluginRepo.war* providing the implementation of the simple plug-in repository.

3.5 Validation

After successfully implementing the system we wanted to validate that the system complies with all functional and non-functional requirements discussed at the beginning of this chapter.

3.5.1 Functional requirements

In order to validate the functional requirements we performed sevaral experiments and performed each of the defined scenarious.

Initially we had to build a component that encapsulates all functionality needed to perform the "Sentiment analysis" service. This functionality is currently hardcoded into the workflow engine outside. We used the already existing tools in 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. At the end we exported the plug-in into a traditional java jar file.

After we had built the plug-in we tried to execute each of the functional scenarios. Using the user interface we were able to plug in and use the components provided by the plug-in. We were able to view the installed plug-in in the management user interface and we were also able to successfully remove it from the system. At the end we were also able to share and install the plug in through the plug-in repository. All these proved that all functional requirements has been accomudated by the 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:

- *Performance* - In each phase the user requests are distributed between several nodes which in the case of many users using the system makes it effective and reduces the response time.
- *Availability* - 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. Additionally, load balancers can be considered as a single points of failure. In order to overcome this problem we provide also clusters of load balancers and enable DNS load balancing.**more about dns**
- *Scalability* - One of the important requirements is that the system has to be able to gradually scale for supporting more users by just adding new hardware components. This structure satisfies this requirement because in order to scale one should just add new nodes to the backend cluster, add new nodes to the database clusters. The system can be scaled up to the point where each workflow is executed by different node and the plug-ins for each scientist are stored on a different storage node.
- *Security* - The connection between the user and the backend can be encrypted(HTTPS) so that data transition is protected. The system's communication channels are also secured by firewalls. Also each storage node is backed up in case a device fails.

3.6 Limitations and Future Work

Most of the limitations and shortcomings of our approach are inherited from OSGI.
not loading every time plug-ins. use cashe or communication between processes.

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

[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

[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

[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

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