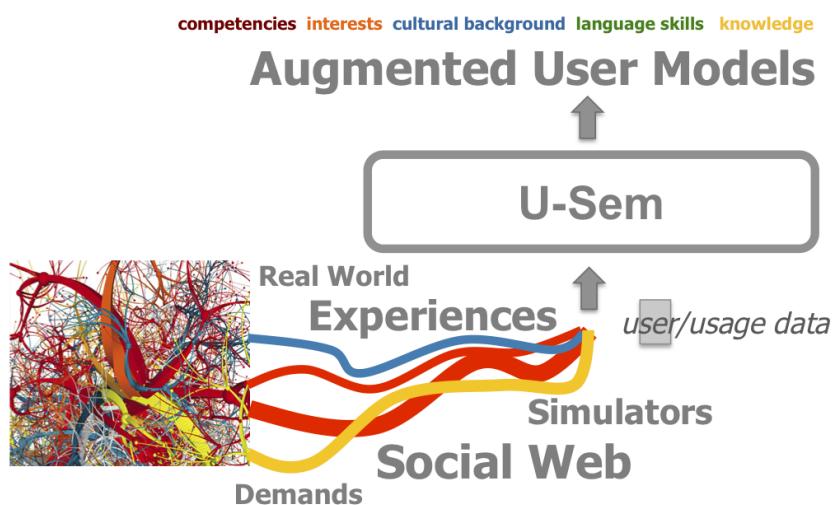


U-Sem, a platform for augmented user and context modelling

Master's Thesis



Borislav Todorov

U-Sem, a platform 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 Vratsa, Bulgaria



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

© 2013 Borislav Todorov. Coverpicture: some picture.

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

Author: Borislav Todorov
Student id: 4181840
Email: B.Todorov@student.tudelft.nl

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. Geert-Jan Houben, Faculty EEMCS, TU Delft
University supervisor: Dr. ir. Jan Hidders, Faculty EEMCS, TU Delft
Committee Member: Dr. Andy Zaidman, Faculty EEMCS, TU Delft

Preface

A place to put some remarks of a personal nature.

Borislav Todorov
Delft, the Netherlands
June 24, 2013

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
1.1 Research questions	2
1.2 Outline	2
2 Background	3
2.1 User Modelling Systems	3
2.2 The U-Sem framework	4
2.3 Workflow Management Systems	5
3 Problem definition	9
3.1 Requirements gathering	9
3.2 Feature prioritization	11
3.3 Plug-in Environment	12
3.4 Data Management	14
3.5 Conclusion	16
4 Plug-in Environment	17
4.1 Requirements	17
4.2 Approach	18
4.3 Architecture	26
4.4 Implementation	39
4.5 Evaluation	41
4.6 Limitations and Future Work	43
5 Data Management	45
5.1 Requirements	45
5.2 Challenges	46
5.3 Approach	46

5.4	Architecture	56
5.5	Implementation	62
5.6	Evaluation	64
5.7	Limitations and Future work	67
6	Conclusions and Future Work	69
6.1	Contributions	69
6.2	Conclusions	69
6.3	Discussion/Reflection	69
6.4	Future work	70
	Bibliography	71
A	Glossary	77
B	Requirements and Guidelines	79
B.1	Requirements	79
B.2	Guidelines	80
C	Prioritization questionnaire	81

List of Figures

3.1	The value distribution of the 5 requirements in the U-Sem project	12
3.2	State diagram illustrating the scenario where two scientists extend U-Sem simultaneously and the changes made by Scientist A are lost.	14
4.1	Component-based software development [46]	20
4.2	Component interfaces	20
4.3	Components implementing interfaces	21
4.4	OSGi Service Registry [59]	22
4.5	Context diagram of U-Sem	27
4.6	Business model describing the process for creating a new plug-in for U-Sem	29
4.7	Business model describing the process for installing a shared plug-in/s to U-Sem	30
4.8	Business model describing the process for managing plug-ins in U-Sem .	31
4.9	Layer organization of U-Sem	32
4.10	Component diagram illustrating the functional organization of U-Sem .	33
4.11	Functional decomposition of the <i>Plug-in admin.</i> module	34
4.12	Functional decomposition of the <i>Plug-in access manager</i> component . .	36
4.13	Diagram illustrating the concurrency model of U-Sem	37
4.14	Deployment diagram illustrating the simple setup of U-Sem	39
4.15	User interface for plug-in management.	40
4.16	User interface for uploading and installing plug-ins in U-Sem.	41
4.17	User interface for browsing and installing plug-ins from the plug-in repository.	41
5.1	Class diagram illustrating the RGL Values class hierarchy	49
5.2	Layer organization of the feature	56
5.3	Component diagram illustrating the high level functional organization of the feature	57
5.4	Functional decomposition of the Entities admin. module	58
5.5	Functional decomposition of the Entities definition manager module . .	59
5.6	Functional decomposition of the Entities manipulation manager module .	60
5.7	Context diagram of U-Sem	61
5.8	User interface for entity definitions management	63

5.9 User interface for constructing entity definitions	63
5.10 User interface that provides a quick preview of the entity definitions	64
5.11 Workflow defining the service for building language profiles before the introduction of the feature	65
5.12 The entity definitions needed for building the service for building language profiles	65
5.13 Workflow defining the service for building language profiles after the introduction of the feature	66

Chapter 1

Introduction

Nowadays, there are hundreds of millions of Internet users all over the world that use the internet in their work and leisure time. Every day they are sharing more and more content on the Web especially through the social media sites [26]. This sharing is facilitated by many different systems: microblogging systems like Twitter¹, picture sharing like Flickr², professional profile sharing like LinkedIn³, etc. As a result, there are many user traces on the Web [2].

A lot of businesses and fields are starting to understand how valuable this information can be and how they can benefit by extracting certain kinds of knowledge out of it. There are many areas that can benefit from this including recommender systems, advertising, e-learning, etc. As a result of the many possible applications there is a high demand for approaches and mechanisms for analysing this information and extracting the needed knowledge out of it.

The field of user modelling and analysis is one of the fields that covers this area. It is considered to have a lot of potential [8] and therefore, there is a high demand for engineers to implement systems that provide new or adapt existing algorithms in order to meet the requirements of the customers. There are many existing approaches for building such systems [30] and one of the newly emerging ones is U-Sem [1]. It is based on the idea that the user modelling functionality can be provided to the customers in the form of services which are built by orchestrating different functional components. This approach, however, focuses mainly on the organization and semantics of the building blocks of the services and pay little attention on how the systems are actually constructed. Currently, each engineer has adapted a personal approach for executing the tasks required for the construction of the systems. Many of these tasks, however, are not part of the core of the engineers' work. Such overhead tasks include setting up web servers and databases, implementing functionality that supports the interaction with the databases, communicating with fellow scientist in order to arrange reusing some of their work, etc.

We believe that the current approach that engineers use for building user modelling services based on the U-Sem idea costs them a lot of time and efforts and requires them to spread their attention on a lot of areas many of which are not part of their

¹<https://twitter.com/>

²<http://www.flickr.com/>

³<http://www.linkedin.com/>

speciality which may also influence negatively the quality of the final result. Therefore, in this thesis work we extend the U-Sem idea by proposing a user modelling platform that aims to facilitate the work of the engineers and improve their productivity by reducing the amount of the overhead work they have to perform and focus on devising the modelling services which is also their main expertise.

1.1 Research questions

The main research question is the following:

How to facilitate the work of engineers building user modelling services following the U-Sem approach?

The following sub-questions articulate the problem:

1. **How to enable engineers to extend and modify the services provided by the system without affecting the other users using the system?**
2. **How to enable engineers to collaborate by reusing each others functionality?**
3. **How to enable engineers to work with persistent data without having to deal with issues connected to where and how it is stored?**
4. **How to enable engineers to collaborate on data level?**
5. **How can real-world use cases benefit from this system?**

1.2 Outline

The structure of this document is as follows:

First, Chapter 2 introduces the background information that is needed in order to be able to understand and articulate the problem.

In Chapter 3 we define the problem that has to be solved and further decompose it in two separate sub problems. We also discuss the relevance of each of the sub-problems.

In Chapters 4 and 5 we address each of the sub-problems. This requires the identification of the formal requirements that has to be satisfied in order to solve the problems. We devise an approach for solving each of the problems that is based on the state of the art developments in the field. Then, we discuss the actual architecture of the solution and the way it is implemented. Finally, we evaluate the benefits that each of the solutions brings and propose some directions for future work.

Chapter 6 concludes this thesis. It provides a summary with conclusions and discusses future work.

Chapter 2

Background

This chapter discusses the background concepts that are needed in order to articulate the problem and also the related existing works. Section 2.1 provides a short introduction to the field of User Modelling Systems. In Section 2.2 we briefly introduce the idea behind the U-Sem framework. Finally, Section 2.3 presents the idea of workflow engines and elaborates on RDF Gears in particular which is currently used to support the construction of the user modelling services.

2.1 User Modelling Systems

User modelling is usually considered to originate from the work of Allen, Cohen and Perrault ([3], [12]) and Elaine Rich ([49], [50]) [30]. Their work had inspired many engineers and scientists to build systems that collect information about their users and adapt specifically for each them [60]. Initially, the user modelling functionality is integrated within the systems and there is little separation between the core and the user modelling functionality [30]. However, in the late 80s an effort is made to decouple the user modelling functionality from the rest of the user-adaptive systems and define it as separate components that can be reused [30].

In his work [30] classifies the user modelling components in two categories:

- *User Modelling Shell Systems* - they are part of the system that makes use of the user modelling functionality. During the development phase of the system, they have to be integrated and configured specifically for the application they are used for.
- *User Modelling Servers* - they are not a part of an system but rather independent from it. Usually, they are "centralized", reside in the local area network or a wide area network and serve more than one applications at the same time. The main advantages of this approach lie in the fact that the information resides in a single place (no duplication and easier access control) and the functionality can be used by multiple applications which can also collaborate (information acquired by one application can be used by others).

A large number of user modelling servers have been developed over the time. Some of them are simple academic prototypes others are more sophisticated and com-

mercially used. Examples of the foremost generic user modelling servers include DOPPELANGER [43], PersonisAD [4], Cumulate [7], UMS [31].

However, with the increasing popularity of Social Web applications and the increasing amount of data that is published on the Web everyday there is a demand for a new type of user modelling servers that are specifically designed to take advantage of the available data on the Web [8]. In [1] the authors address this issue and propose a special kind of framework for building user modelling servers based on various types of services. Next section discusses the idea behind it.

2.2 The U-Sem framework

U-Seam represents a framework for semantic enrichment and mining of people's profiles from usage data on the Social Web. It provides methodology for designing services for user modelling and semantically augmenting user profiles. Its idea lies on the assumption that user modelling in today's Social Web sphere relies mainly on three types of input data [1] that can be modelled in RDF format [39]:

- *Observations* - usage data or events such as clicks, tagging activities, bookmarking actions or posts in social media.
- *User characteristics* - explicit data provided by the users about their characteristics such as name, address, homepage, occupation, date of birth, etc.
- *Domain knowledge* - this data is needed in order to produce user profiles that support certain application domains.

Based on these types of input data the authors propose that the user modelling services can be build out of various functional components. This components are classified in the following categories:

- *Semantic enrichment, Linkage and Alignment* - these components are responsible to process the input data. They provide functionality for aggregating and linking user data from Social Web systems like Facebook and Twitter, and integration and alignment of RDF data from Linked Data providers such as DBpedia¹.
- *Analysis and User modelling* - components that given the enriched data from the components in the previous group perform the analysis and user modelling generating user profiles that describe interests, knowledge and other characteristics of the users.

Services are build by orchestrating these components into workflows that provide the required functionality. The process of creating and executing workflows is usually automated using workflow management systems which are discussed in the next section.

¹<http://dbpedia.org/>

2.3 Workflow Management Systems

Workflow Management Systems are systems that allow a set of high level operators that all do a single well-defined specific task to be composed in a larger whole - the workflow. Depending on the desired application some systems are specifically concerned with timing, concurrency, synchronization, data management aspects of a complex process. They may also imply special domain specific languages for defining the workflows. Literature provides a lot of examples of workflow management systems that are currently available: *Taverna* [23], *Kepler* [37], *RDFGears* [17], etc.

RDF Gears seems the best option for engineers applying the U-Sem framework because, firstly, it is specifically designed to deal with RDF data which is how all the user modelling inputs are modelled according to the specification of the U-Sem framework. Secondly, it already being used by the engineers and it has proved to be suitable for the situation. Therefore, we will discuss how *RDF Gears* works and what are its features in the next section.

2.3.1 RDF Gears

RDF Gears is a workflow management system designed specifically for building workflows for integration and transformation of RDF data [17]. It aims to provide a data integration framework that allows expressing and executing complex algorithms. *RDF Gears* consists of three main components: *RDF Gears Language*, *RDF Gears Engine* and *RDF Gears graphical user interface* [17]:

RDF Gears Language (RGL) is a workflow language that is used to define the workflows as a combination of functional components. Each functional component has zero or more inputs and one output and defines certain operation that is performed on its inputs. The result of the operation is provided as the output of the component. The inputs are defined as input ports and can be connected with a particular output port of another component. All ports are associated with data type. The type system of the language is discussed in the next subsection. The connections between the components represent the flow of data between them. Components and the connections between them form an acyclic directed graph which is the actual workflow.

RDF Gears graphical user interface represents a web application that facilitates the construction of workflows in order to make the construction of complex RGL expression easier. It provides drag and drop operation for editing workflows and visualizing the connection between components as a visual directed graph [54]. The GUI builds the defined workflows as XML files which are the workflow source code that represent the RGL expressions and are executed by the *RDF Gears Engine*.

RDF Gears Engine represents an application that is responsible for the execution of the predefined workflows by evaluation of the RGL expressions. Discussed in the next sub-section, it also provides optimizations for efficient execution of workflows. *RDF Gears* expects as an input XML files the define the workflows and provides graphical and RESTful interfaces that enable users to execute the workflows.

RDF Gears comes with a set of built-in components that can be used for creating workflows. However, this set is not exhaustive and usually engineers have to implement their custom components in order to be able to build the required user modelling services. The RDF Gears engine is implemented in Java and the process of adding custom components to it requires the following steps: First, engineers have to check out the source code of RDF Gears from the software repository. Then, they have to implement their components in Java and add the result Java classes and resources to the source code of the engine. Finally, they have to build the system and deploy it on a Web Server so that is available to the users.

Type System

The RGL language defines values and type system. The values of RGL combine the value system of the Named Nested Relational Calculus (NNRC) with the RDF data model [17]. The RGL values can be defined as follows:

- Every RDF value (URI, literal or graph) is a value in RGL.
- If r is a named row over values, then $[r]$ is a value which is called a record.
- A finite multi set of values is a value that is called a bag. Bags are restricted to contain only values of the same type.

RGL uses static type system that allows engineers to specify the expected input and output values for each component on design time. It also allows the evaluation of the RGL expression to be performed before the actual execution in order to ensure the correctness of the RGL expression. RGL introduces types for the URI, literals and RDF graph values. We refer to them as basic types. RGL types are inductively defined as follows:

- The basic types are types
- If T is a named row over types, the $\text{Record}(T)$ is a type that defines record values
- If T is a type, then $\text{Bag}(T)$ is a type that defines bag values

Optimizations

In order to execute the predefined workflows efficiently RDFGears performs special optimizations like pipelined bag iteration, lazy evaluation of expressions and buffered, streaming serialization. Designing the solution proposed in this work we have to ensure that the current way for compiling and executing workflows is not compromised and also workflows are still optimized in order to execute efficiently.

The way RDF Gears engine applies some of the optimizations is based on the assumption that workflows are build from components that have no side effects. This means that they are only responsible to produce its output value and do not change its state or the state of any other component or system. As a result, if the output value of a component is not needed the engine can decide not to execute it at all and there will be no change in behaviour of the workflow. The engine takes advantage of this idea in two aspects:

- *Workflow compilation* - workflows are compiled into a data structure which represents directed graph and contains the output node and all components that the output directly or indirectly depends on. Therefore, if there is no path in the graph from a component to the output node then the component is removed. The removed components are never executed.
- *Workflow evaluation* - the engine provides functionality that performs lazy evaluation of the compiled workflows. The engine takes the output node and recursively executes all components which outputs are needed for the execution. As a result, components which output is not requested are not executed at all(eg. when the If/Then/Else component is used). Additionally, when a component's output is required by several components then the first time it is executed it caches the result and directly provides it for the next request. Therefore, RDF Gears guarantees that each component is executed at most once.

Chapter 3

Problem definition

As discussed in the background section the U-Sem paradigm defines how the user modelling services has to be organized by orchestrating specific types of functional components. The specification, however, does not provide any formal mechanism or guidelines about how these services and components are constructed and managed in practice. Currently, the only work done in that direction is the adoption of a workflow management system (RDF Gears) which facilitates the orchestration of components into user modelling services.

Because of the lack of standardized approach for building and managing all these components every engineer is responsible to adopt his own approach. Additionally, discovering the potential of Social media for user modelling [8], has lead to an increased demand for U-Sem engineers to design and build many user modelling services. As a result engineers are constantly required to implement or adapt variety of functional components that are needed for the new services. We believe that the lack of formal approach and tools that support the process results in a lot of overhead work for engineers and can also potentially lead to increased amount of implementation errors. Therefore, in this work we investigate whether the current situation can be improved by introducing a user modelling server that is designed to support the process of creating and exposing the user modelling services to the users so that the overhead work of the engineers is limited and they can focus on the core of their work - the user modelling and analysis algorithms.

In order to be able to provide such system we, first, have to identify how engineers work and what are the tasks that require them to do a lot of overhead work and can be fully or partially automated. Next sections describe the approach that we used in order to identify the high level features that engineers can benefit most from.

3.1 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.

3.1.1 Interviews

Stakeholders are the people that have some kind interest in the project. They are the ones that will be affected by the project and thus they are the source for identifying

the characteristics of the system we have to build. In our case the main stakeholders are the engineers that are already using or will use the U-Sem approach for constructing user modelling services.

Having identified the stakeholders of the system, literature suggests a wide variety of possible approaches for eliciting the requirements [21] : interviews, questionnaires, user observation, workshops, brain storming, role playing, etc. At the end, we decided to use the semi-structured interviews technique since it is widely used and it has proved to be effective through the years [15].

3.1.2 Identified features

We analysed carefully all the raw information that was gathered 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 list of features that the system should provide:

- **Simplified access to social media** Engineers base a lot of their analysis on the information provided by users in the social media. However, each social media provides access through a special API. Currently, each engineer has to implement a component for retrieving the required data from each type of social media. Providing feature that makes available to all scientists flexible components that can be configured to retrieve specific data from the different types of social media is considered a benefit from the engineers because it will save them considerable time and efforts. They also will not need to know the specific details concerning each of the APIs.
- **Plug-in environment** Engineers constantly build new services and improve and adapt existing one. Therefore, engineers will benefit if the system is able to facilitate the process of extending itself by plugging in custom logic such as RDFGears functions and other functional components. Engineers should be able to manage(add/update/remove) this custom logic at runtime (without restarting the system). This process should not affect the work of other users.
- **Universal data storage** Many of the engineers build services that need to store various types of data(e.g. intermediate and final results). Currently, they are forced to manually set-up databases and program the components needed for interacting with each type of data. This is usually not a trivial task, it requires time and knowledge and therefore, it resembles a big overhead to the engineers. Thus, they can be significantly benefited if the system provides a mechanism that enables storage and retrieval of arbitrary types of data.
- **Integration with Hadoop** The amount of information that has to be processed in the system can sometimes be huge. Therefore, sometimes, it has to be processed by external Hadoop based systems. Currently, engineers have to manually manage the exchange the information to and from the Hadoop systems. Therefore, providing a mechanism that facilitates and standardises the process is also considered as a benefit for some engineers.

3.2 Feature prioritization

Because of the limited scope of this thesis work we are focusing on the features that will bring the most advantage for the engineers. In order to do that, we have to prioritize the requirements based on the impact they will provide. This process, however, is not trivial since each of the stakeholders has slightly different view on the benefits provided by each of the features that we already identified. Therefore, we performed a research in order to find an approach that can be applied in this situation.

Requirements prioritization is a relatively old research topic and there are numerous approaches that are currently available [40]. The most popular include Quality Function Deployment (QFD), the Analytical Hierarchy Process(AHP), the cost-value approach proposed by Karlsson, Wiegert's 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 analytic hierarchy process approach [51] . This decision was based on the fact that it is especially suitable for prioritizing a small number of requirements [27] and it is a proven and widely used [28]. In literature, as a disadvantage of this approach is considered the fact that it takes no account of interdependencies between requirements [51]. However, this issue is not a problem for our solution because the project high-level features are loosely coupled and they have little dependency between each other.

3.2.1 Features prioritization using AHP

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). **Appendix 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, as advised in [27] we had to make sure that the provided results are consistent. In order to do that we had to calculate the consistency indices of the pairwise comparisons. According to [27] values lower than 0.10 are considered acceptable and even values around .12 are commonly achieved in the industry and can also be considered acceptable. The calculation showed that two of the participants have indices higher than .23 which indicates serious inconsistencies (table 3.1). Therefore, we asked them to revise their answers and the results afterwards we measured values that are acceptable (table 3.2).

Once we had achieved satisfying results we calculated the distributions. We outlined the candidate requirements in a diagram (figure 3.1) 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

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

Table 3.1: 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 3.2: Consistency ratios for each of the stakeholders after refinement.

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.

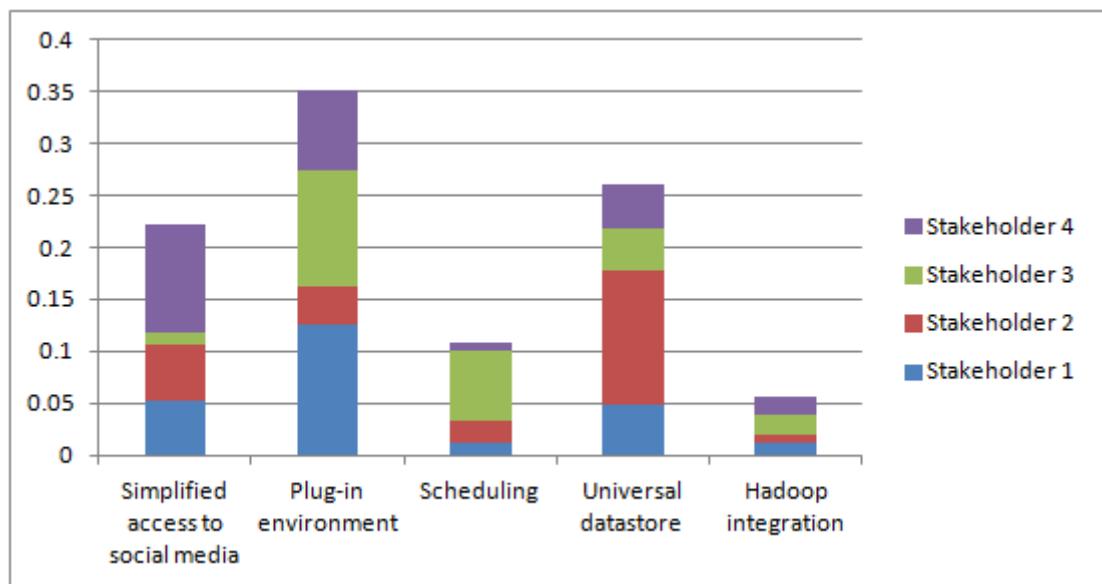


Figure 3.1: The value distribution of the 5 requirements in the U-Sem project.

Finally, based on the provided results from the prioritization and the limited scope of this thesis we decided to further address the two features that will provide the most benefits for the users: "Plug-in Environment" and "Data Management". Next section sections cover each of the problems in details.

3.3 Plug-in Environment

During the initial interviews with the engineers that are going to potentially use the system, we revealed 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 for the purpose of experimentation, it is available for testing, demonstration and evaluation purposes.

We also performed additional interviews with the engineers in order to reveal how this process is currently done, what are the problems they currently face and cost them a time and efforts, and what are their expectations for the future system. Based on the results from the interviews we concluded that the main problem is that engineers have to "hardcode" the code implementing new components into the source code of the workflow engine (RDF Gears). In this way, the software code implementing the algorithms becomes part of the workflow engine. This approach has a lot of disadvantages, it is error prone and brings a lot of discomfort to the engineers working with the system because:

- 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 3.2 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.
- 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 disadvantage from software engineering perspective and it concerns the lack of a mechanism for establishing modularization between the implemented components. In software engineering, modularization is con-

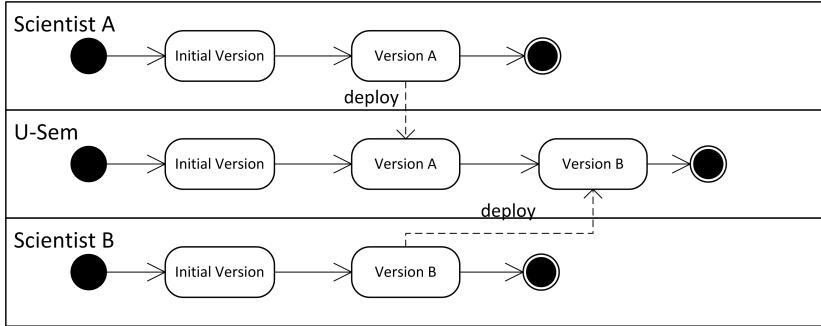


Figure 3.2: State diagram illustrating the scenario where two scientists extend U-Sem simultaneously and the changes made by Scientist A are lost.

sidered as a key property for improving extensibility, comprehensibility, and reusability in software projects [44]. The most important aspect of a successful modular system is its information hiding capabilities [56]. 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 [16]. 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 [9].

TODO Transition to the next part of the story

3.4 Data Management

Our research showed that many of the services implemented by the engineers require data storage and retrieval in some form. The structure and semantics of the data varies greatly but it can be roughly classified in the following groups:

- *Raw data* - Engineers reported that most of the analysis and modelling services they devised are based on the social media. Basically, they have to retrieve certain entries(e.g tweets from Twitter) which are the basis for the analysis. In certain cases engineers have to store this "raw data" locally. Usually, this is as a result of the fact that the retrieving the entities requires time and also some social media APIs limit the number of entities you can get in certain amount of time [11]. This makes the execution of workflows slow(the system has to wait to get the needed entries every time). Therefore, engineers are forced to store these entries locally and only make sure they are up-to-date prior to the execution of a workflow.
- *User provided data* - Some services require information that is not available from the social media and has to be provided by the users of the system. For

example, in certain usecases users has to fill in questioners and the results from later analysis is adjusted based on the answers provided by the users. It is infeasible to ask the users for that information every time that is why it has to be stored within the system.

- *Intermediate results* - Some services consist of two phases. The first phase continuously calculates some intermediate representation of the raw data. In the second phase, on user request the final result is calculated based on the intermediate data. One of the examples is the Twinder service [58]. Therefore, between the two phases the intermediate results have to be stored and later retrieved back.
- *Ready user profiles* - Some services require that users has to be able to monitor how the results evolve over time. For example, in e-learning systems users want to be able to see how the knowledge of a certain person has changed after following a certain course in order to measure how helpful the course was for that person. Therefore, every time a service is executed the results have to be stored so that they can be later further analysed.
- *System data* - Finally, many of the features of the system need to store some kind of information. For example, the multi-user sport has to store all kind information about the users of the system: user names, passwords, privileges, etc. The scheduling feature has to store information about the time each workflow has to be executed.

Currently, the system does not provide any functionality to support users in defining workflows that require data operations. Each scientist is forced to create custom components that serve the particular requirements. However, this process costs a lot of time and efforts, and suffers from many downsides and problems:

- *Knowledge required* - designing and implementing components for dealing with persistent data is not a trivial job. It requires specific type of knowledge(**database**, **administration**). Many of the engineers building services have mathematical or statistical backgrounds and are likely not to have in depth knowledge in the database field. In order to be able to build their services they have to acquire this knowledge which can cause significant overhead and waste of time. Additionally, the fact that they are not professionals in the field may lead to problems and shortcomings.
- *Server administration* - Most of the storing solutions require setting up a dedicated database server. These servers have to be hosted somewhere, maintained, backed up, etc. All these require a lot of effort and if every user has to do it, it will result in large amount of duplicated work and overhead. If engineers decide to use a shared database server then appears the question of who is responsible to manage it and ensure its security and privacy.
- *Dynamic data structure* - It is expected that the structure of the stored data might change over time. When a database with fixed schema is used (like most SQL solutions) then every time the structure changes the engineers has to manually

connect to the database and apply the changes manually. This can be really annoying, time consuming and error prone. Therefore, automating this process can save time to engineers and reduce the number of mistakes caused by carelessness.

- *Collaboration* - collaborations between engineers on data level is reported to be quite important and can save them a lot of time. Currently, there are no facilities available to support that requirement. Engineers have to organize this collaborations personally. Additionally, because the collaboration are not integrated in the system they are likely to be hard to monitor and control.
- *Low level queries* - Currently, most services use SQL databases. As a result, engineers have to build various SQL queries. The SQL query language is considered as low level language **needs ref** and causes some overhead to the engineers especially the ones that are not familiar with it. Therefore, introducing higher level query language can be a big plus.
- *RGL translation* - Generally available database solutions are not capable to deal with data in the RGL format introduced in RDF Gears. Therefore, every single component that deals with persistent data has to translate the RGL values to values compatible with the databases solution and vice versa. Clearly, all this code is redundant and removing this responsibility from the engineers will save them time and efforts so that they can focus their attention to the core of their work.
- *RDF Gears and components with side effects* - Components that store data are components that have side affects. However, RDF Gears is not designed to work with such components and some unexpected behaviour might be expected. Therefore, engineers building components with side effects and are not aware of the way RDF Gears operate internally risk to introduce problems that are hard to detect.

In this thesis we aim to propose a solution that is capable to overcome these problems and save engineers a lot of time, efforts and prevent mistakes while dealing with persistent data.

3.5 Conclusion

TODO

Chapter 4

Plug-in Environment

This chapter proposes a solution to the problem of extending U-Sem by adding custom functionality. Section 4.1 identifies all functional and non-functional requirements that a successful solution must satisfy based on the problems discussed in Section 3.3. Section 4.2 discusses state of the art approaches and technologies that can contribute to solving the problem. Section 4.3 describes the architecture of U-Sem that we propose in order to solve the problem. Section 4.4 discusses the implementation that we provide in order to be able to verify the capabilities of the proposed architecture. Section 4.5 discusses and verifies whether the proposed solution satisfies all of the requirements. Finally, in section 4.6 we discuss the limitations of the proposed design and suggest aspects in which the design can be improved in the future.

4.1 Requirements

Having all the problems identified in section 3.3 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 in the evaluation section where we discuss how and to what extend the architecture satisfies each of them.

4.1.1 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 have to be able to extend U-Sem by adding custom user modelling components 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 installed functionality** - Users has to be 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.

4.1.2 Non-functional requirements

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

- *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 backed up. 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 section 3.3, 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.

4.2 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, monitor) 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: *Component-based software engineering* [24]. Following this idea we propose an approach for solving the problem by decomposing it into the following sub-problems:

- *Modularization and component management* - this problem addresses how the system is modularized into components that can be added removed from the system on demand without affecting its operation.
- *Collaboration between engineers* - enable engineers to collaborate by reusing each others work.
- *Dependencies management* - help engineers to manage the dependencies between components ensuring that the system is in a consistent state.
- *Component construction support* - assist and if possible automate the process of creating components.

Next subsections address each of these sub-problems presenting the way they are approached and solved.

4.2.1 Modularization and component management

In this section we discuss the basic idea and the advantages that the *Component-based software engineering* idea brings. We also discuss 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.

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 [46]. 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 4.1, 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 scientist is responsible to build only a piece of the system which provides certain service.

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 [47] [48]. Additionally, the applicability of this approach is supported by the fact that it 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

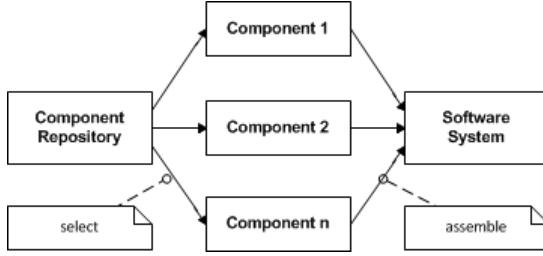


Figure 4.1: Component-based software development [46]

Distributed COM (DCOM), Sun's(now Oracle) JavaBeans and Enterprise JavaBeans, OSGI.

TODO Add refs

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 [9]. 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 [10]:

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 4.2, interfaces can be considered as points where custom functionality provided by another component can be plugged in.

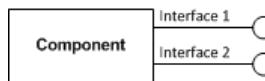


Figure 4.2: Component interfaces

Components are functional units providing functionality by implementing interfaces. As can be seen on figure 4.10, 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 [9] 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

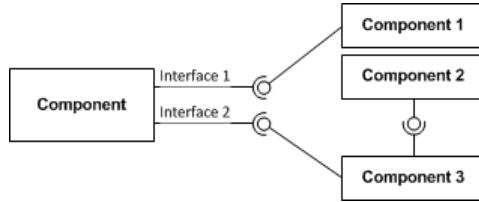


Figure 4.3: Components implementing interfaces

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.

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.

[32] 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 from 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 [6] and Open Services Gateway initiative (OSGI)¹. 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 [59] which may suggest that it is well tested and therefore is more stable. The next subsection focuses on how OSGI works and its features that are interesting for the architecture of U-Sem.

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:

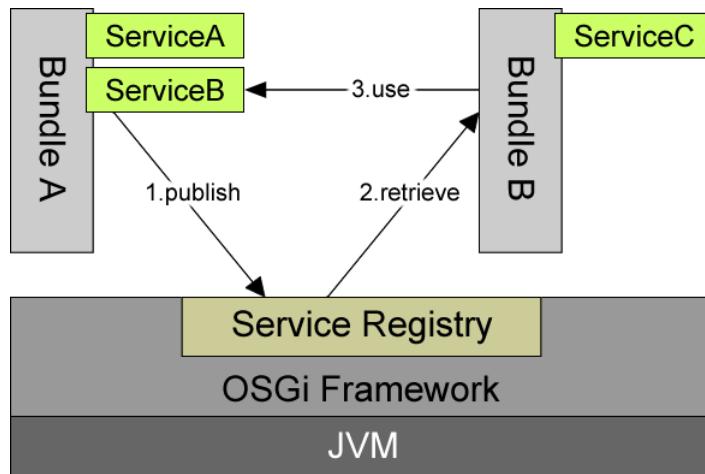


Figure 4.4: OSGi Service Registry [59]

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.

¹OSGi Alliance. <http://www.osgi.org>

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 than the one provided by the Java language [59]. 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 4.4, 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 [45]. 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 [45]. The platform was designed to support the safe execution of fully untrusted code [13]. In order to achieve this, it has introduced the following features [19]: 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 [45].

TODO user interface for monitoring and installing the plug-ins

4.2.2 Collaboration between users

Engineers have to be able to collaborate between each other by reusing each others functionality. In order to support this feature we propose an approach based on the *Component Repository* idea [53]. We introduce a Plug-in repository which 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. Literature suggests that this approach is also used in other plug-in based systems like Eclipse [38].

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.

4.2.3 Dependencies management

It is very likely that plug-ins have dependencies between each other. This may be as a result of reusing functionality from other engineer's plug-ins or depending on a plug-in that contains common functionality like extracting data from social media and storing data into a database. Therefore, in order to use one plug-in engineers have to make sure that all its dependent plug-ins are also installed. The problem is that the dependency information is only available on compile time. Once the plug-in is compiled it hard to tell what are its dependencies because this information is packed inside the *Jar* file representing the plug-in. Therefore, engineers are responsible to remember what are the dependencies of the plug-ins they are installing or to refer to the source code. This problem is even more severe when engineers reuse shared plug-ins from the plug-in repository since they do not have access to the source code and therefore, have to contact and communicate the dependencies with the owners of the plug-ins which is an obvious overhead and waste of time. We believe that this approach for dealing

with the plug-in dependencies is not effective and will pose a significant overhead for engineers.

Literature suggests that there is an already existing attempt to overcome this problem which is the Eclipse P2 framework [33]. It represents a component provisioning system that can be configured to work with any kind of components(not only plug-ins) and proposes the idea of packing components into installable units that define the dependencies between them. Clients of the system can choose to install between the installable units which ensures that all the required components will be installed. The system is designed to be flexible in order to be applicable for a variety of different scenarios. However, as a result it requires a lot of additional work from the engineers in order to set all the installable units up which is a big disadvantage for the U-Sem scenario. Therefore, we propose a simpler approach specifically designed for solving the problem in U-Sem. It is based on the fact that U-Sem is designed to work only with plug-ins and each them already contains information about its dependencies and therefore, the system can automatically read these dependencies and verify if they are all satisfied and the system is consistent. The proposed solution consists of two parts:

- The Plug-in repository is extended so that when a new plug-in is being installed it checks the current configuration and any missing dependent plug-ins are offered for installation as well.
- When engineers open the plug-in monitoring user interface the system notifies for any missing dependencies and proposes their installation from the repository.

The main benefit from this approach is that it completely automates the dependency management process and does not require any additional work from the engineers when they are building plug-ins. There is, however, one use-case which requires some manual work. It originates from the fact that in certain cases OSGI also allows plug-ins to depend on Java packages without specifying their source [5]. This is not a significant problem for U-Sem since this functionality is only used for plug-ins to refer to the packages which are not encapsulated in plug-ins like the workflow engine. These packages are already part of the plug-in environment and therefore, no action is required. If there is still a future use-case where this functionality is needed engineers are responsible to manage these dependencies manually.

4.2.4 Plug-in construction support

In order to create a plug-in engineers have to create and configure a plug-in project. This process involves setting up all required dependencies and building the required folder structure within the project. From engineers point of view, all these steps are considered as an overhead that can be time consuming especially for new members of the team or for engineers without computer science background. Therefore, in order to automate this process the system provides a special Eclipse based SDK (software development kit) which extends the out-of-the-box capabilities of Eclipse providing user interface that assists engineers to easily create and set-up plug-ins for U-Sem. This is achieved by introducing a special *Eclipse Project Template* [55] that can automatically create U-Sem plug-ins, set up their folder structure and add the required dependencies. Additionally, often engineers have to create simple one-component workflows in order

to test a specific component. Thus, the solution is also automates this process by allowing engineers to create plug-in projects with pre-wired one-component workflows. Engineers only have to provide the names of the component and the workflow in the user interface and put the source code into the Java class dedicated for the created component.

TODO Transition to the next part of the story

4.3 Architecture

Based on the approach devised in the previous section we designed the software architecture for the *Plugin Environment* feature of U-Sem. In this section we provide the description of the proposed architecture focusing on the static and dynamic structures of the solution and its externally visible behaviour.

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. [52] 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. [52] 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.

4.3.1 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 [62]. [62] 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 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: engineers providing user modelling components for U-Sem and a Plug-in repository supporting the collaboration between engineers. Figure 4.5 illustrates the updated runtime environment of U-Sem.

Scientists In order to be able to construct user modelling workflows, engineers first have to build and deploy the required components that are needed for the workflows. In order to do that, engineers first have to build a plug-in that encapsulates the new functionality defining the components. 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 through HTTP/s.

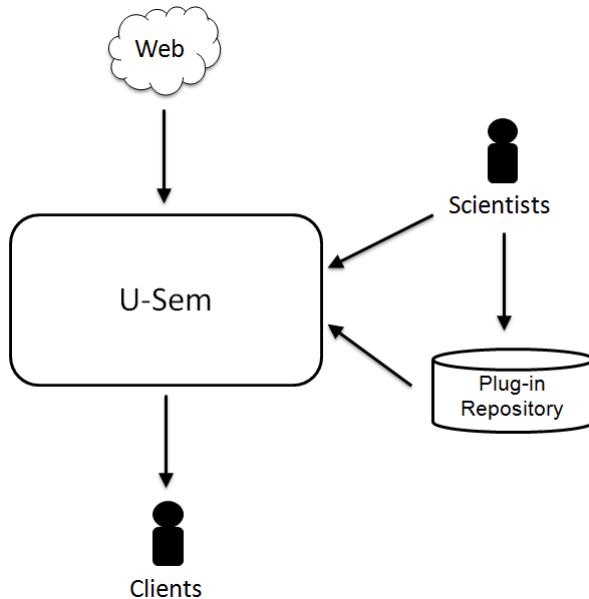


Figure 4.5: Context diagram of U-Sem

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.

4.3.2 Interactions

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) [61] 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 [14]. 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 4.6 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 3.3, 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 4.7. 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.

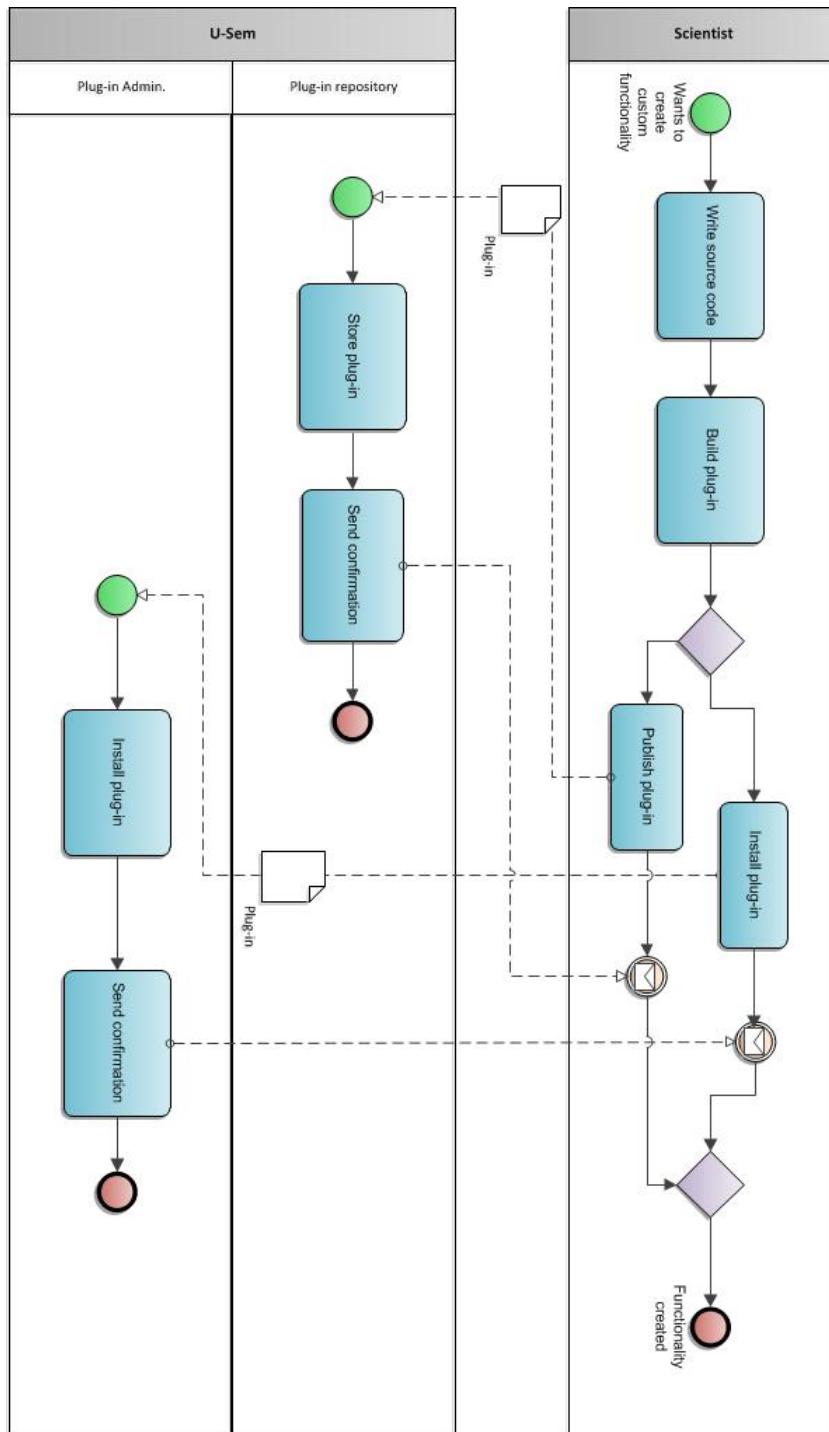


Figure 4.6: Business model describing the process for creating a new plug-in for U-Sem

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.

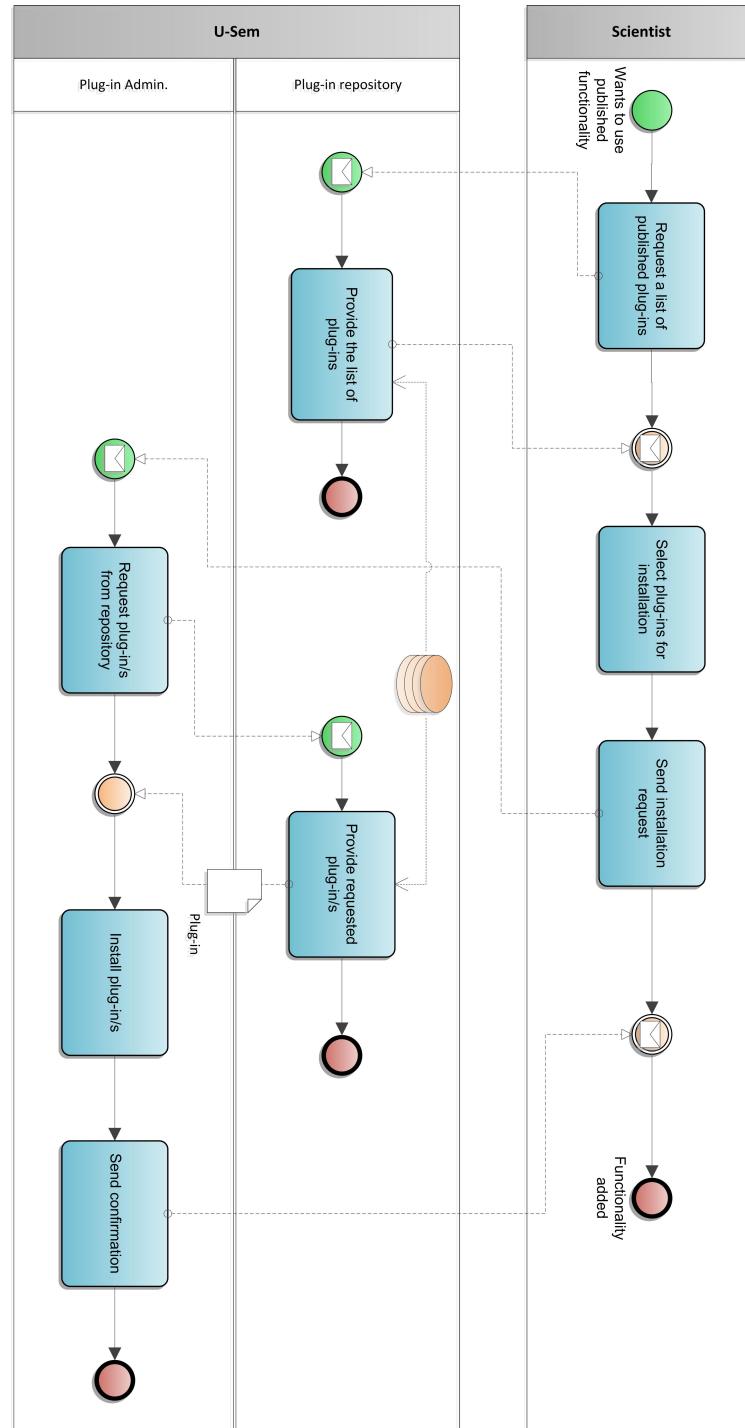


Figure 4.7: Business model describing the process for installing a shared plug-in/s to U-Sem

This use case was modelled into the *Plug-in Management process* which is illustrated on Figure 4.8. In this case, we have interaction only between the scientist and U-Sem.

Scientists can monitor the currently installed plug-ins at any time by contacting

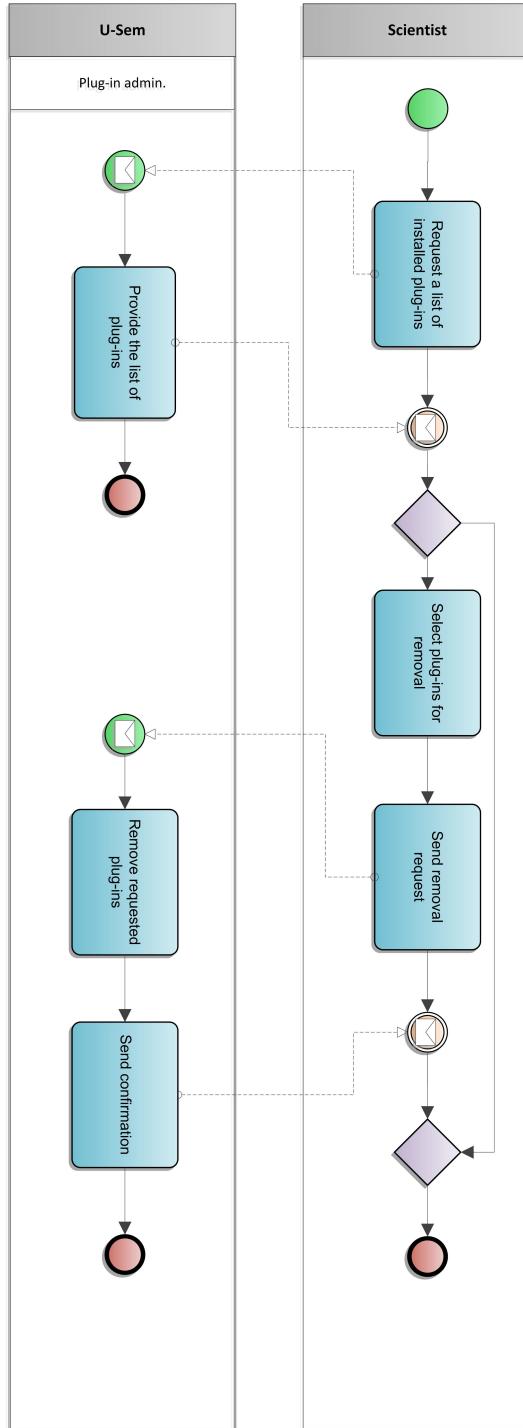


Figure 4.8: Business model describing the process for managing plug-ins in U-Sem

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 from the private space of the scientists and when finished send back a confirmation message.

4.3.3 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 [52]. 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 4.9 and consists of the following layers:

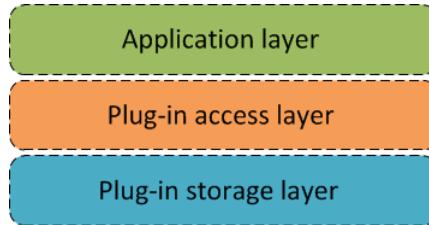


Figure 4.9: Layer organization of U-Sem

- *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 4.10 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:

- *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

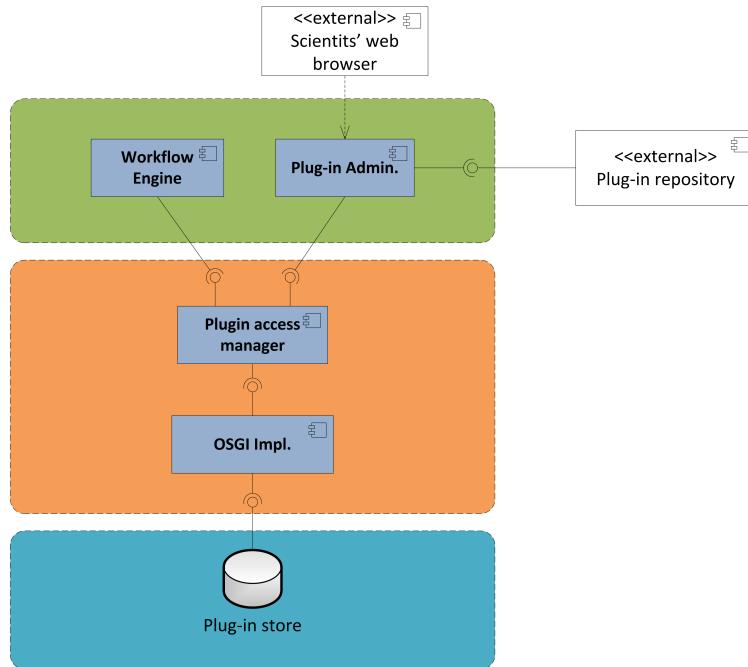


Figure 4.10: Component diagram illustrating the functional organization of U-Sem

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 4.11. It consists of the following components:

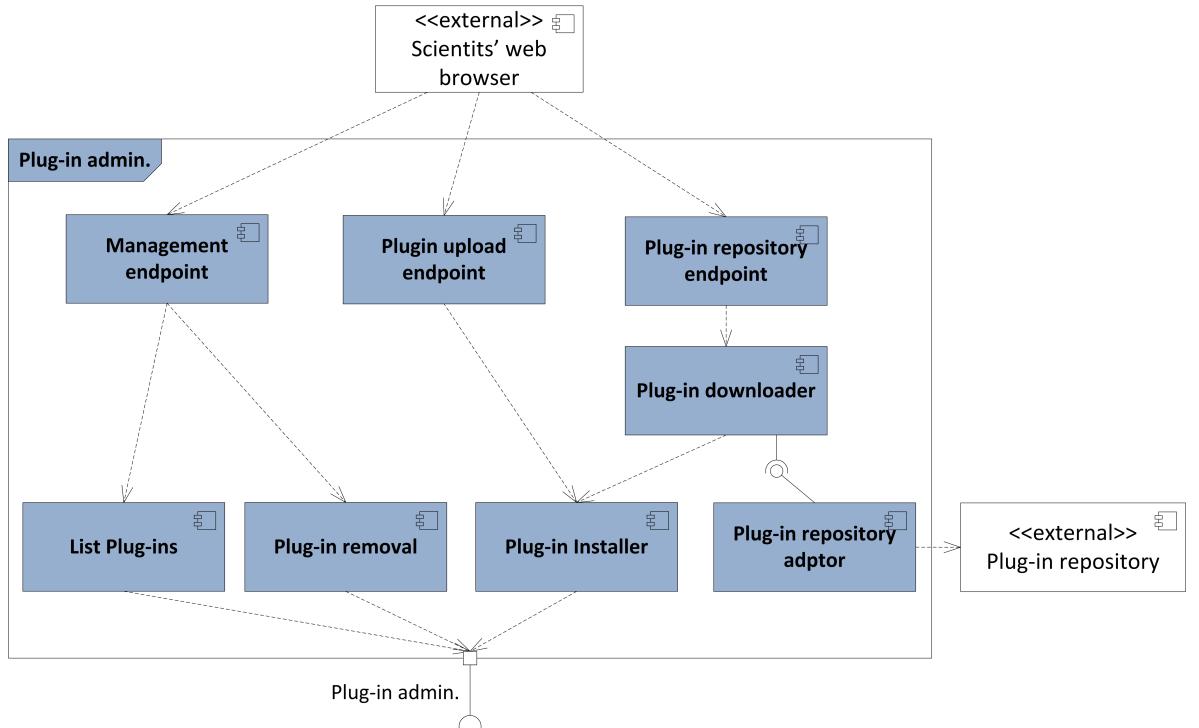


Figure 4.11: Functional decomposition of the *Plug-in admin.* module

- *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 component 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 4.12 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 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.

Starting an OSGi instance can be a time costly procedure especially when there are a lot of plug-in to be loaded. Therefore, in order to increase the performance of the system with the help of the *OSGi Instance pool* component when an OSGi instance is no longer needed it is not destroyed but cached so that it can be later reused and not created again.

- *OSGi Instance pool* - based on the connection pool idea [63], this component provides a cache of OSGi instances so that they can be reused when future interactions with the plug-ins are required.

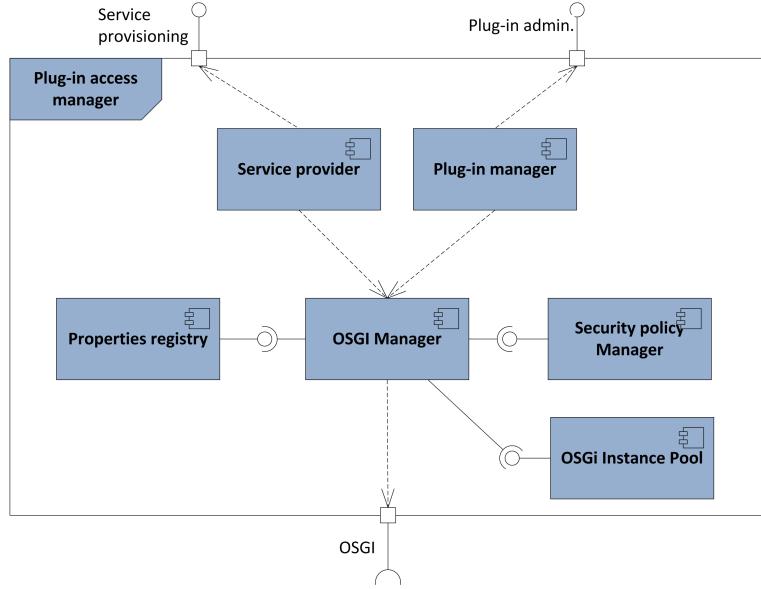


Figure 4.12: Functional decomposition of the *Plug-in access manager* component

- *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 these 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.

4.3.4 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 4.13 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

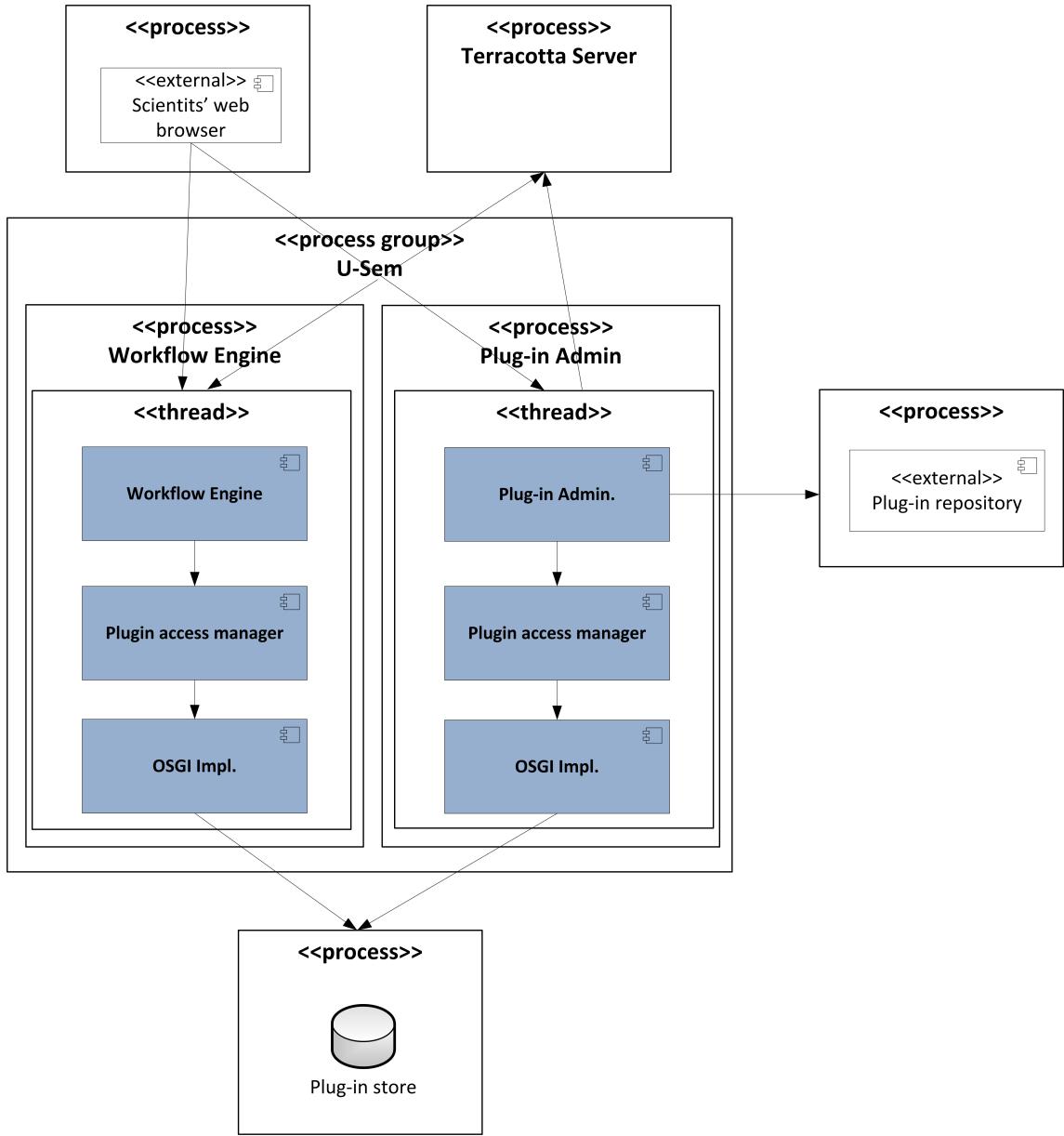


Figure 4.13: Diagram illustrating the concurrency model of U-Sem

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. The main idea behind this design is to separate the workflow and the Plug-in administration functionality into separate processes which enhances loose coupling and, hence, improves maintainability and simplifies upgrades to new versions of the workflow engine. Additionally, if in future there is a need for higher performance the workflow process can be replicated without affecting the plug-in ad-

ministration functionality. This organization, however, also introduces some problems that have to be solved:

Problems

The first problem lies in the fact that plug-in installation (plug-in admin.) and loading of plug-ins (workflow engine) are two actions that are completely independent from each other. Therefore, if the workflow engine tries to load the plug-ins when the plug-in admin. is in the middle of installation of plug-ins then the plug-ins may fail to load or load incorrectly.

Secondly, as discussed in the previous section the system keeps a pool with the OSGi instances so that they can be reused instead of loaded from scratch. However, whenever a plug-in is installed or removed these cached instances become obsolete and have to be discarded. In order to do that, the plug-in admin. has to notify the workflow engine for the change in the plug-ins which is a problem when the two components are in separate processes.

Solution

In order to solve the first problem we propose a solution that is based on synchronization between the processes. The solution is based on the *Reader-Writer Lock* idea. It extends mutual exclusion locks by enabling concurrent read access to a shared resource but requires exclusive access for write operations [34].

Our solution maps to the *Reader-Writer Lock* idea as follows:

- The shared resource is the plug-in storage of an engineer.
- The workflow engine acts as reader of the shared resource.
- The plug-in administration acts as writer of the shared resource.

As a result, when a change to the plug-ins is needed it is executed exclusively and any components that want to load the plug-ins have to wait until the change is applied. Therefore, the workflow engine is protected against loading the plug-ins while they are inconsistent. This approach also brings a performance benefit since loading plug-ins is not exclusive and can be done simultaneously by many components.

The standard *Reader-Writer Lock* in Java, however, works only within the virtual machine². In our case we have to synchronize entire processes. Our research showed that there are already existing tools that provide this functionality [20]. The tool that is used in the proposed solution is Terracotta³. It introduces a process that is responsible for the lock and the other processes communicate with it in order to obtain the lock and use the resources.

In order to solve the second problem we extend the architecture by providing mechanism for inter-process communication in order to enable the plug-in admin. to notify all processes for any updates so that they can clean their caches. The general idea is that all components that maintain caches register to receive updates when plug-ins

²<http://docs.oracle.com/>

³<http://terracotta.org/>

change. Whenever this happens all registered components are notified using a broadcast protocol [25]. Our research showed that the Terracotta system that we used to solve the first problem also provides implementation of a broadcast protocol. Therefore, when a workflow engine loads the plug-ins it contacts the Terracotta process and registers itself for updates regarding the plug-ins. When an engineer removes or installs a plug-in in the plug-in admin. contacts the Terracotta process which then notifies all the registered clients.

4.3.5 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. In order to test their work scientists has to be able to easily install and set up U-Sem so that they can concentrate entirely on their work rather than setting up the system. Therefore, aim here is to enable scientists to setup the system fast, on a single machine and with little configuration effort.

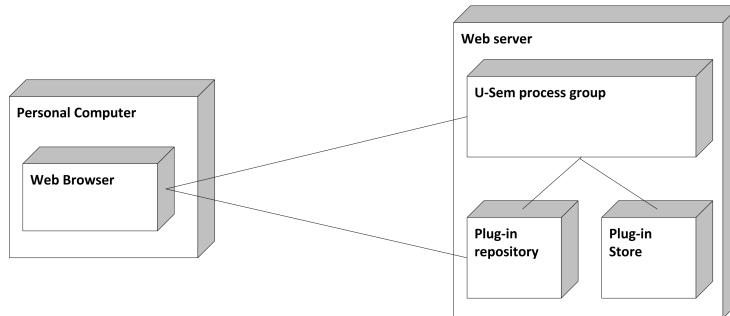


Figure 4.14: Deployment diagram illustrating the simple setup of U-Sem

This deployment organization is illustrated on figure 4.14. The benefit from this organization is that all components are deployed and run on the same web server, on the same physical machine, and as a result, the costs for maintenance and initial set-up are reduced.

4.4 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: Equinox⁴, Felix⁵ and Knopflerfish⁶ [59]. 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,

⁴Equinox. <http://www.eclipse.org/equinox>

⁵Apache felix. <http://felix.apache.org>

⁶Knopflerfish. <http://www.knopflerfish.org>

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 workflow definition files provided by custom plug-ins.

Name	Symbolic name	Version	Vendor	Action
org.eclipse.osgi	org.eclipse.osgi	3.8.0.v20120529-1548	N/A	Delete Plug-in
SentimentAnalysis	SentimentAnalysis	1.0.0.201211291352	N/A	Delete Plug-in
Test_Project	Test_Project	1.0.0.201301221037	N/A	Delete Plug-in

Figure 4.15: 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 the jQuery UI⁷ and Bootstrap⁸ technologies. Figure 4.15 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 4.16 and the endpoint for browsing and installing functionality from the repository depicted in figure 4.17.

⁷<http://jqueryui.com/>

⁸<http://twitter.github.com/bootstrap/>

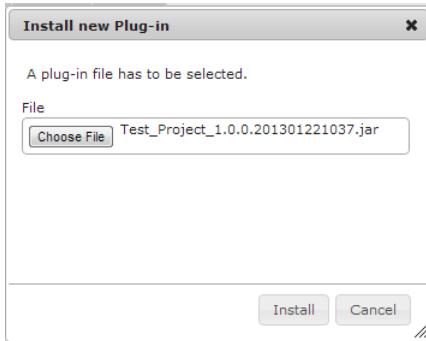


Figure 4.16: User interface for uploading and installing plug-ins in U-Sem.

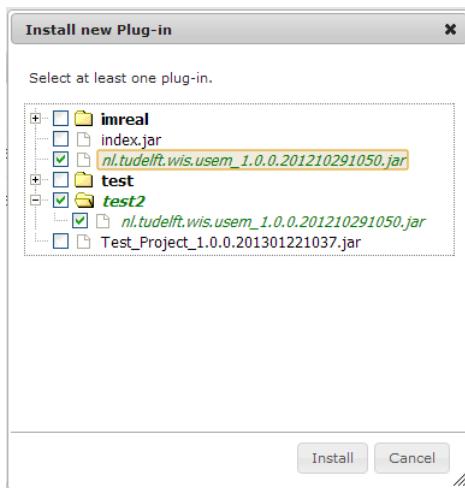


Figure 4.17: User interface for browsing and installing plug-ins from the plug-in repository.

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

4.5 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.

4.5.1 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.

4.5.2 Non-functional requirements

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

- *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 3.3 and provides information hiding and cross component communication capabilities that enforce better component modularization [59].

4.6 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 [45] 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. [57] 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.
- *Vulnerabilities on OSGi platform level* - This kind of vulnerabilities are related to weaknesses in the OSGi run-time. [45] suggests an approach for overcoming them by adding additional security checks in the OSGi implementation.
- *Vulnerabilities on JVM level* - This vulnerabilities can be further divided into categories [18]:
 - *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.

[18] proposed an 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 limitation does not pose a significant threat for U-Sem. However, all these vulnerabilities have to be addressed if in future the system is to be extended to enable access from unverified scientists.

Chapter 5

Data Management

This chapter proposes a solution to the *Data Management* problem in U-Sem. Section 5.1 identifies all functional and non-functional requirements that a successful solution must satisfy based on the problems discussed in Section 3.4. Section 5.3 discusses that state of the art approaches and technologies that can contribute to solving the problem. Section 5.4 describes the software architecture that we propose in order to solve the problem. Section 5.5 discusses the implementation that we provide in order to be able to verify the capabilities of the proposed architecture. Section 5.6 discusses and verifies whether the proposed solution satisfies all of the requirements. Finally, in section 5.7 we discuss the limitations of the proposed design and suggest aspects in which the design can be improved in the future.

5.1 Requirements

Having all concerns discussed in section 3.4 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.

5.1.1 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 - Insert data** - The system has to provide RDF Gears component and also API that can be used to store data.
- **UC2 - Update data** - The system has to provide RDF Gears component and also API that can be used to update existing data or part of it.
- **UC3 - Delete data** - The system has to provide RDF Gears component and also API that can be used to delete existing data or part of it.

- **UC4 - Query data** - The system has to provide RDF Gears component and also API that can be used to query existing data.
- **UC5 - Collaboration** - The system has to enable engineers to collaborate on data level by accessing and manipulating each others data. The system has to enable engineers to manage the privacy of their data. They have to be able to define whether their data(or parts of it) can be accessed and manipulated by the other engineers.

5.1.2 Non-functional requirements

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

- *Privacy* - the solution has to ensure the privacy of the stored data. The only way of interaction between scientists has to be achieved through the collaboration mechanism.
- *Compliance* - the solution has to ensure that the newly introduced RDFGears components comply with the RDFGears specification.
- *Configuration management* - in order to simplify the configuration procedure of the system it should enable engineers to provide all the required configuration information concerning the new types of components within the plug-ins that define them. In this way, apart from the plug-in installation no additional effort is needed in order to use the components.

5.2 Challenges

TODO ?

- Multitenant storage with collaboration
- ORM of dynamic entities
- ORM fine grained access control

5.3 Approach

In this chapter we device a framework that is responsible to provide the Data management functionality for U-Sem. By carefully analysing the requirements defined in section 5.1 we concluded that the problem can be broken down into the following sub-problems:

- *Data storage and retrieval* - enable RGL oriented workflow engines(in our case RDFGears) to hold their long-term data safely in a database and access it when needed.
- *Multi-tenancy* - enable multiple users to work simultaneously with the solution and enable them to collaborate together in a controlled and secure fashion.

- *RDFGears extension* - ensure and if required extend the workflow engine so that it is able to deal with this new type of functionality correctly and efficiently.
- *Integration with the Plug-in Environment feature* in order to simplify plug-in installation and configuration.

Next subsections explain how each of the sub-problems is approached and solved.

5.3.1 Data storage and retrieval

The main objective of this chapter is to facilitate the data storage process in U-Sem so that engineers can store and retrieve their data without having to bother how and where it is actually stored. Therefore, in order to achieve this we introduce an simplified abstraction layer which hides the actual storage implementation. Engineers only interact with the abstraction layer and they are not concerned about any implementation details. In the next sub-section we discuss the proposed abstraction which is followed by presentation of the approach we propose in order to implement it.

Abstraction

In order, to simplify the engineers work we introduce an abstraction layer which requires engineers to only configure a set of predefined components for interaction with the data. These components are implemented as RDF Gears components and can be configured only by using the user interface. We introduce the following components which enable users to perform most popular operations on data:

- *Store* - this component is responsible to store its input data into the database.
- *Update* - this component is responsible to replace existing data or parts of it with the provided input data.
- *Delete* - this component is responsible to remove data identified by the engineer.
- *Query* - this component expects a query as input and is responsible to execute it and provide the results as an output for the component.

There are two issues that concern the usage of these components and have to be further discussed. The first one concerns how the stored data is organized and the second one concerns the query mechanism.

Structure and Semantics of the data Section 3.4 makes clear that users' needs regarding data storage vary greatly and may also change in future. It is hard to predict and define the structure and semantics of all data that engineers might like to store. Therefore, a more generic solution is needed. Therefore, the solution enables engineers to define the semantics and structure of the data they want to store on demand. This structure has to be able to be adapted over time if/when users' needs change. In this work we refer to each data entry as *entity*, each group of entities that have the same semantics and structure as *entity types* and the information that defines the structure and semantics of each entity type as *entity definition*. All data in RDF Gears is presented in the RGL format and therefore, the intuitive approach is to provide users functionality that enables them to define these entity types in terms of RGL values.

Querying data As specified above querying functionality is needed in order to configure some of the components. The abstraction provides querying language that enables engineers to define queries in terms of predefined entity definitions. There are two types of queries:

- The first type of queries is required by the *Update* and *Delete* components. They provide a mechanism for identification entities or sub-entities that will be updated or removed. It allows engineers to identify them by filtering the entities based on their values.
- The second type of queries enable engineers to retrieve data by filtering, grouping and aggregating the already stored entities.

The specific configuration requirements for the components and the actual query language used, however, greatly depend on the implementation of the feature. Therefore, they are discussed below in the section discussing the implementation approach.

Existing approaches

After rigorous research we did not find any existing solutions that can solve our requirements. However, we discovered that Object/Relational Mapping (ORM) frameworks provide similar functionality to what we are looking for. Basically, they provide a methodology and mechanism for object-oriented systems to hold their long-term data(expressed as programming objects) in a relational database and later have it back to objects when it is needed [42]. ORM frameworks provide a level of abstraction over the process of storing data and in this way users can interact with the data in the form of objects and they are not required to know details about the underling representation of the data.

Applying the same approach for our RGL entities seem to completely satisfy our requirements. However, in order to exploit that idea we have two options. The first one is to implement and ORM solution from scratch and the other one is to extend/adapt existing ORM solution so that it is able to deal with the RGL entities. In general, reusing a popular and widely used solution might be beneficial because it is likely it is heavily tested(at least from all the engineers using it) and thus, provide higher quality.

Our research reviled that the Java Persistence Architecture API (JPA) is the "de facto" leader in the field ORM solutions for Java. It represents a Java specification(JSR 317) for accessing, persisting, and managing data between Java objects / classes and a relational database. Making its way as a major standard has resulted in a lot of the big players in the field providing an implementation of it. The most popular include: Hibernate(JBoss), TopLink (Oracle), EclipseLink(IBM) and openJPA(Apache) **include footnotes.**

Because of its popularity, widely usage and formal standardization using JPA seems like the best option for our solution. However, in order to be able to use it we have to solve several major problems which are discussed in the next sections.

Model RGL entities in terms of JPA

RDFGears is already capable to map the RGL values to Java objects as illustrated on figure 5.1. Therefore, the challenge is to model these classes in JPA in a sensible and

efficient way. The naive approach would be to reuse this structure directly. In this way each of the classes is represented as a separate JPA entity. The main advantage of this approach is that the structure is fixed, it does not depend on the structure of the entities that we want to store. However, this approach also brings a lot of disadvantages:

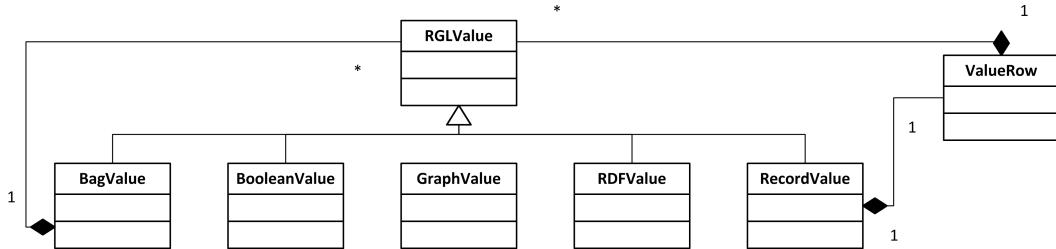


Figure 5.1: Class diagram illustrating the RGL Values class hierarchy

- all data resides in just several database tables. This makes the solution inefficient and hard to scale.
- all entities are stored in the same database tables and it is hard to distinguish between each other. For example if one wants to query only one type of entities
- structure of the data/queries is not enforced by the database and has to be explicitly validated
- changing the entities structure may easily leave the db in inconsistent state
- hard to enforce access control

The alternative approach that we propose is to map each entity type into separate JPA entity which is named after the name of the entity which has to be unique. Table 5.1 proposes a methodology to map RGL values to JPA entities.

RGL	JPA
RecordValue and its corresponding ValueRow	Class with multiple properties *
BagValue	Bag pointing to a class representing the values in the BagValue
BooleanValue	Property of type boolean
RDFValue	Property of type string
GraphValue	Property of type string

Table 5.1: Table describing how RGL values are mapped to JPA

* JPA requires that the root element is a class and therefore, our solution requires and ensures that the root element of all entity types is a RecordValue. When this value is mapped to a JPA class the name of the class is the name of the entity type. Any nested RecordValues are mapped to JPA classes named after the name of the property of the ValueRow that points to them directly or indirectly in the case where the RecordValue is wrapped in a BagValue.

This approach does not suffer from the problems of the first approach because data is distributed over many tables. However, it exposes several issues. The first thing is that each time the structure of an entity changes the JPA mapping information and the schema of the underling database has to be updated as well. Our solution to this problem is discussed in the next section. The second thing is that JPA requires that each entity has distinct id but RGL does not provide such term. We solve that problem by introducing an auto generated *id* property for each JPA class which is made transparent to the client system(RDF Gears). Finally, JPA classes must have distinct names. We solve this problem by making sure that each user defined entity type has an unique name, and all record properties have distinct names. In order to make the names of sub-entities(records) distinct we append to the front of their name the full path from the root JPA class separated by "_".

Virtual entities

JPA expects that each entity will have its own class representing it but our entities are virtual and there are no specific Java classes that represent each of them. One solution to this problem would be to generate these classes on runtime. However, this approach is complex and error prone. Therefore, investigating the capabilities of different engines implementing the JPA specification relieved a promising feature in Hibernate called "Dynamic models" [29]. It basically allows engineers to define the mapping logic into a special mapping XML file and on runtime present the entities in the form Java collections(Maps, Lists, Sets, etc.). Table 5.2 shows how U-Sem entities can be expressed in terms of Java collections. This approach requires that every time when an entity is defined we have to build the XML mapping file and on runtime convert the RGL entities into Java collections and use JPA to store them into the database.

RGL	Java
RecordValue/ValueRow	Map with separate entries for each ValueRow
BagValue	List
BooleanValue	boolean value in a collection
RDFValue	String value in a collection
GraphValue	String value in a collection

Table 5.2: Table describing how RGL values are mapped to Java collections

Dynamic entities

Another problem that has to be addressed concerns the dynamic nature of the entities. In our solution engineers have to be able to update the entity types whenever this is needed. Unfortunately, ORM frameworks are not good in dealing with such things. The problem is that whenever the entity type definitions changes not only the mapping XML file has to change but also the database schema. Hibernate provides a tool for table generation(hibernate.hbm2ddl.auto configuration option) but we were not able to find any documentation specifying how it works and what are its limitations. Additionally, we conducted experiments that reviled that this tool is not suitable for our

situation. One of the problems is that when an entity is renamed then new set of tables is generated and the old ones are left behind. Therefore, in order to solve that problem reliably our solution provides functionality that is responsible to first update the mapping files and secondly update the schema of the database so that it is consistent with the new entity type definition. The algorithm for mapping JPA entities to tables in the relational database is the following:

- A JPA class is mapped to able with columns for each atomic property.
- A JPA property refers to another class then the target class is mapped recursively following the first step but its result table is also extended with an unique foreign key column pointing to the table representing the owner of the property.
- In the case of a JPA bag property referring to another class then the class is mapped recursively following the first step but its result table is also extended with a foreign key column pointing to the table representing the owner of the property.

Querying

The solution also has to provide approach for querying the stored entities. The query language has to be able to, first, allow engineers to write queries that identify entities or sub-entities to be updated or deleted. And, secondly, to enable them to filter, group and aggregate entities.

In order to solve these requirements, we propose the use of JPQL [35]. It is a SQL like, high-level query language which omits details about the internal representation of the stored data [35]. It is the standard query language for JPA entities and is supported by most JPA providers including Hibernate. Its capabilities allow engineers to execute all the needed operations discussed above. Because of the direct mapping from RGL to JPA entities all possible JPQL query operations on the predefined entities are supported. The only exception is the additional *id* field. In order to make it transparent to the users it is named "\$id\$"(hibernate uses this notation to identify service fields) and using this field in queries is forbidden. Additional advantage of using JPQL comes from the fact that it is similar to SQL. In the current situation many of the engineers reported using SQL to implement data retrieval components and therefore, the transition to JPQL is likely to be easier.

RDF Gears Components

Finally, we have to define the RDF Gears components that are responsible to enable the user to configure and execute each type of the supported data operations. They are also responsible to automatically convert the data from RGL to the JPA compatible format and vice versa. The solution provides the following RDF Gears components:

- *Insert* - this component is responsible to insert an entity into the database. The user has to provide the name of the entity that has to be stored and the data itself.
- *Query* - this component is responsible to enable engineers to query data. They have to configure the desired query and provide the required parameters for it.

- *Delete* - this component is responsible to delete data from the database. Normally in JPA delete functionality relies on ids, but since we do not have this in the semantics we provide an alternative approach. In this case, users have to provide a query that selects all entities/sub-entities that have to be deleted. Users are responsible to make sure that the query will produce only the needed results.
- *Update* - this component is responsible to enable engineers to update entities or part of them. Like the delete operation, updates in JPA are also based on ids. So we approach the problem similarly the user writes a query that selects all entities (sub entities) that will be updated. They have to also state the field of the selected entities that has to be updated and provide the values. The system distinguishes between two types of updates. The first one is replacing the value of a field(simple type, record or bag). The second one is aimed for appending an element to a bag.

5.3.2 Multi-tenancy

The second main challenge that we have to solve is to enable multiple engineers to work with the solution simultaneously. In literature this is referred to as multi-tenancy. It has significantly increased its popularity in recent years as a result of the advancements in cloud computing and database as a service in particular [22].

Designing the solution we have to take into account the following things: how to organize the data into the database, how to enable collaboration and how to manage access control.

Database Organization

There are three main approaches for designing multi-tenant solution [22]:

- *Independent Databases and Independent Database Instances (IDII)* - in this approach users share only the hardware(server). For each user an independent database instance is running. This approach provides good data isolation and security but maintenance cost is significantly increased as a result of the multiple running instances. Additionally, collaboration between engineers is problematic because of the difficulty to execute queries on data that is spread over several databases.
- *Independent Tables and Shared Database Instances (ITSI)* - in this approach users share the hardware but also the database instance. Each user has private tables(user id is usually appended at the beginning of the table name [22]). This approach reduces the maintenance cost but still the number of tables increases linearly with the number of users. Collaboration is possible but querying data relating to all users he has to be done by uniting the data from the tables of each user. Another disadvantage of this approach is that because the table names are different for each user we also need a separate JPA mapping for each user which increases complexity additionally.

- *Shared Tables and Shared Database Instances (STSI)* - in this approach, all users share the db instances as well as the tables. In this approach the number of tables does not increase when the number of users increase. As a result the maintenance cost is reduced, queering is simplified(no unions required) and JPA mapping is straight forward. However, this approach has some disadvantages. First, engineers are responsible to maintain the required level of isolation between different users(usually a column indicating the owner of the data is required and additional where clause when querying for particular user). Second, tables contain the data for all users which may lead to decrease in performance.

Discussing the advantages and disadvantages of all the approaches above, we decided to use the STSI because it is easy to map to JPA, enables collaboration, simplifies querying and is easier to maintain.

Collaboration

The ability for engineers building workflows to collaborate and reuse each others data is essential [36]. Choosing STSI organization makes it easy for engineers to work with the data of the others. However, there are several issues that has to be discussed and addressed:

- *privacy* - sometimes users might not want others to use their data. This might be because of a privacy issues or entities are not designed to be shared. Additionally, engineers might want to limit the access to the data. For example they might only allow others to read the data but not to modify it. Therefore, when the engineer defines an entity he should be able to specify its sharing options which are enforced by the solution. This issue is further discussed in the *Access Control* section below.
- *semantics and structure of shared data* - the solution can further assist engineers that want to reuse shared data by providing information about its structure and semantics. Implementing such functionality will save them a lot of time since otherwise this information has to be communicated by other means which may take a lot of time.
- *entities mutation* - introducing the multi tenancy feature introduces the problem with the consistency. By design entities can change and evolve over time. Therefore the system should prevent any inconsistent results produced if the structure changes at the time a workflow is executed. Our solution deals with that problem using transactions. All data operations are executed in transactions which temporary lock the tables for modification. As a result, the changing request will wait for the tables to be released before applying the updates to the database schema.

Access control

As mentioned above engineers have to be able to control the access to their data. The storage abstraction discussed in this chapter enables users to insert, update, delete and

query data. Therefore, we propose an extension to the abstraction that will enable engineers to also control the execution of these operations on their data by other engineers. The basic idea is that when engineers define their entities they can also specify how their data can be accessed by others.

The hierarchical structure of the entities expressed in RGL is very similar to XML and it can be, indeed, expressed as XML [17]. Therefore, adopting an approach used for establishing access control for XML document systems seems to be feasible. Inspired by the work of [?] we propose a fine grained access control framework for RGL entities. It is based on access control rules that are defined by the engineers that build a particular entity. These rules are responsible to grant access to other engineers for entities or parts of entities. They can be presented as triples $G \times O \times R$ where:

- $g \in G$ is the grandee it can be a user or user group that this rule targets.
- $o \in O$ is the object that the rule applies for. It can be entity or parts of entity which as discussed in previous sections are identified by expressions constituting the full path from the root of the entity separated by "_".
- $r \in R$ is the operation that is granted: insert, update, delete or query.

Once we have the rules we have to identify the granularity (which components of the entities they can be applied). It can vary from entire entities to the level of basic RGL values. Choosing the entity level of granularity seems to be too sparse since it does not support use-cases where engineers want to share only parts of the entities and protect the rest. On the other hand, providing access control for every single element in an entity is likely to make the user interface too complicated and thus, harder to configure. Therefore, we propose to use an intermediate level of granularity which provides a reasonable flexibility and also can be easily configured and implemented.

The approach allows engineers to apply the access rules for each entity and also for all records in the entity hierarchy. If no access rules are specified then the elements are considered private and accessible only by their owner. The access policy for the other element is the access policy defined for the first parent record in the hierarchy or if such does not exist - the policy for the entire entity. In order to save engineers time when the same rules have to be applied for the entire entity hierarchy or parts of it the approach also defines an inheritance mechanism. By default the access rules defined for the entity are inherited for all the entire structure of the entity. Engineers, however, may choose to override these settings for a particular records. When the parent settings are overridden then the new rules are automatically inherited in the sub-hierarchy where the same approach is applied recursively.

Implementation approach The solution has to be able to enforce the sharing options of the entities and make sure that data is not accessed illegally. Therefore, the solution provides access control functionality that is responsible to enforce this policies. Basically, JPA is not good at doing this, it does not provide any access control functionality out of the box. Therefore, there are two possible places to put the access control logic.

- *Above JPA* - our solution wraps around the JPA engine and thus, we can control all the requests that it receives. We can provide functionality that validates if

the user has the needed permissions to execute that functionality. However, from engineering perspective doing this will be hard since JPA(JPQL) allows engineers to construct relatively complex queries and building a functionality that parses these queries is not a trivial job to do and significantly increases the risk of security wholes.

- *Database level* - the second option is to make the database enforce the security policies. We think that this approach is much easier because databases already provide well established access control mechanisms [41]. The only think that we have to do is to apply the security policy using the database language every time an entity is created or modified. As a result, the database will stop the JPA framework accessing data that it is not authorised to access.

Most SQL databases like MySQL¹ enable engineers to define access control rules on table level. The proposed approach in this chapter defines that each entity is mapped to a root database table and each record within the entity hierarchy is also mapped to a distinct table. Therefore, the system must only translate the access control rules of U-Sem to the rules supported by the database.

5.3.3 Components with side effects in RDF Gears

In the **Background** section we discuss how RDF Gears evaluates and optimizes workflows which is based on the assumption that components do not have side effects. However, with the introduction of the new data manipulation components this assumption is no longer correct. Therefore, the engine will no longer perform its tasks correctly. The trivial approach would be to simply remove all these optimizations but this is not a good idea because efficiency will suffer continuously. Therefore, we propose an alternative approach which aims to make the engine perform correctly but also keep the efficiency.

The problem is that the engine executes only the components that contribute to the final result which in the basic case is the output node and its (indirect) dependencies. Our solution extends this idea and states that the components that contribute to the final result are also the components that have side effects and their (indirect) dependencies. Our solution consists of the following steps:

- extend the function description notation so that the engine knows if a component has side effects or not
- on execution time when the data structure containing the components that are going to be executed is built we also include all components that have side effects and recursively all the components that they depend on.
- we perform all the optimizations to the output node(as it is currently done) as well as to all components with side effects.
- when the workflow is executed we execute it for the output node and for all components with side effects. The caching mechanism prevents components to

¹<http://www.mysql.com/>

execute several times. The output of the workflow is still the output of the output node.

These steps ensure that all components with side effects are executed and the execution is efficient because all the optimizations are applied.

5.3.4 Integration with the plug-in environment

We also designed an additional feature that aims to save time to engineers. The idea of this feature is to integrate the data management feature with the plug-in environment feature. The benefit from this feature covers the situation when engineers create a certain kind of user modelling service which requires data interactions and bundle it into a plug-in. Since this feature interacts with data it will most probably require entity definitions in order to work. Therefore, every time this plug-in is deployed on a server the engineers has to manually define the needed entities through the user interface. Obviously, this operation may cost significant time and is error prone. Additionally, the problem becomes severe when an engineer wants to use a shared plug-in produced by someone else. In this case the two engineers need a way to somehow exchange the entity definitions.

In order to solve that problem we have devised a mechanism that can deal with this issue automatically and thus save time to the engineers. Each entity definition is stored in an XML file in the file system. Therefore, every time an engineer creates an entity definition he can export it as an XML file. Then, he can include this file in the plug-in and register it within the plug-in context. When this plug-in is installed the system automatically detects the presence of these entities and installs them automatically so that the functionality of the plug-in can be used right away without any need for configuration.

5.4 Architecture

We have already discussed the main problems that our solution is facing and the approach we will use to solve each of them. In this chapter we discuss the architecture of the solution.

5.4.1 Functional View

All components that take part in the data management functionality can be classified in three layers. This organization is illustrated in figure Figure 5.2 and consists of the following layers:

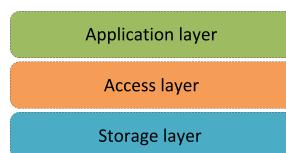


Figure 5.2: Layer organization of the feature

- *Application layer* - this layer consists of all functional components that are interested in the data management feature. These applications are responsible to provide functionality to the user for defining and manipulating data.
- *Business layer* provides functionality for defining the structure and semantics of the data and manipulating the actual data. The functional components that build this layer are responsible to enforce the security and privacy policies of the system.
- *Storage layer* is responsible to provide storage functionality for storing the entity definitions, the hibernate mappings and the actual data.

5.4.2 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 5.3 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:

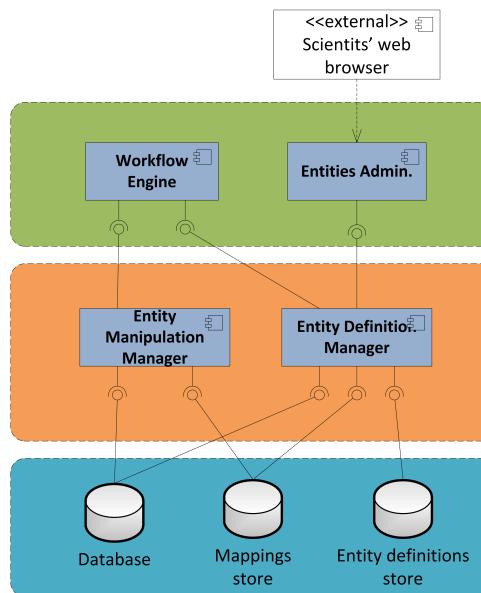


Figure 5.3: Component diagram illustrating the high level functional organization of the feature

- *Entity definitions store* - Provides functionality for storing the data describing the entity definitions.
- *Mappings store* - Provides functionality for storing the data defining how entities are mapped to the database.
- *Database* - SQL database that stores the actual data.

- *Entity Definition Manager* - this component is responsible to provide the logic for defining the structure and the semantics of the data entities. It is also responsible to provide the mappings that state how the data is mapped to the database. Additionally, this component is responsible to prepare the database(create SQL tables and set access permissions) for working with the defined entities. The functionality is exposed by a API that enables the high level components to manipulate the entity definitions. Further decomposition of this component is provided in the next section.
- *Entity Manipulation Manager* - this component is responsible to provide the functionality needed for manipulating the data based on the previously defined structure(entity definitions). It provides implementation for creating, updating, deleting and querying data. It is also responsible to enforce the access control over the data. High level components access the functionality provided by this component through an API. Further decomposition of this component is provided in the next section.
- *Entities Admin.* - is responsible to deal with the administration of the data entities. It provides the system's endpoint(user interface) for interaction with the scientists. Further decomposition of this component is provided in the next section.
- *Workflow Engine* - uses the interface provided by the Entity Definition Manager and Entity Definition Manager to enable engineers define data entities and create services that manipulate persistent data. During the workflow configuration phase it uses the entity manipulation interface in order to obtain the list of already defined entities(their structure and their semantics) assisting the user to easily define workflows that require data manipulation. During the workflow execution phase it uses the Entity Manipulation interface to execute the defined data operations.

5.4.3 Entities administration

This section defines the functional decomposition of the Entities administration component which is illustrated on figure 5.4. It consists of the following components:

- *Read entity definitions endpoint* - This component provides the user interface needed for browsing the already defined entities.
- *Delete entity definition endpoint* - This component provides the user interface needed for removing a selected entity.
- *Store entity definition endpoint* - This component provides the user interface needed for creating new or updating existing entity definition.
- *JSON de/serializer* - Since data in the user interface is presented in the form of JSON. This component is responsible to convert the entity definitions from/to JSON format.

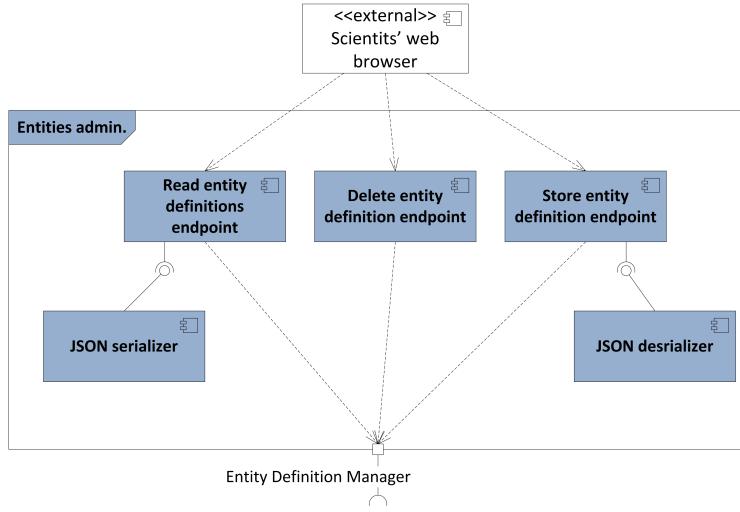


Figure 5.4: Functional decomposition of the Entities admin. module

5.4.4 Entity Definition Manager

This component is responsible to provide API which can be used by application layer components in order to manage the entity definitions. Figure 5.5 shows the functional decomposition of the *Entity definition manager* module. It consists of the following components:

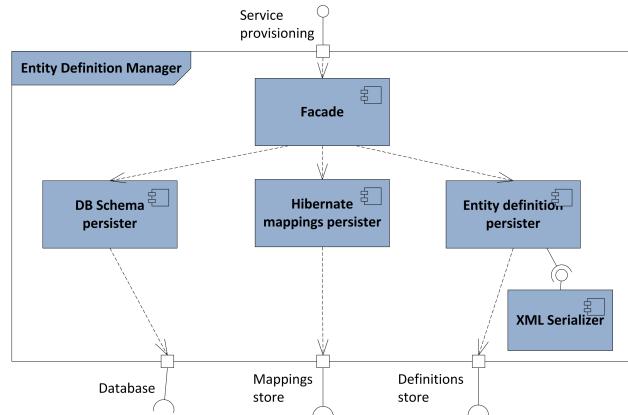


Figure 5.5: Functional decomposition of the Entities definition manager module

- *Facade* - this component provides the API for manipulating the entity definitions.
- *DB Schema persister* - this component is responsible to make sure the database schema is always corresponding to the entity definitions. Any time the entities are changed the database schema is updated. Additionally, this component also defines the access permissions for each access table based on what the user creating the entity has specified.

- *Hibernate mappings persister* - this component is responsible to construct the mappings that define how data is mapped and stored in the database. Since we are using Hibernate, this information is created and stored as .hbm files. Any time the entities are changed the mappings are updated.
- *Entity definition persister* - this component provides functionality for storing the entity definitions.
- *XML Serializer* - This component is responsible to store and retrieve the entity definitions. It provide a level of abstraction over the way the definitions are stored. As a result, this will be the only component that is affected in case of any change of the place and format of the data is required. The default implementation of this component stores the entity definition in the file system as XML files.

5.4.5 Entity Manipulation Manager

This component is responsible to provide API which can be used by application layer components in order to perform CRUD operations over the already defined entities. Figure 5.6 shows the functional decomposition of the **Entity Manipulation Manager** module. It consists of the following components:

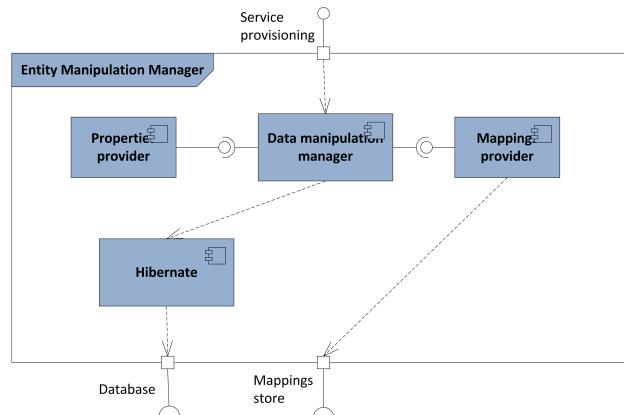


Figure 5.6: Functional decomposition of the Entities manipulation manager module

- *Data manipulation manager* - this component provides the API for manipulating data. This component manages the communication with the Hibernate framework. It is responsible to start/stop the framework and monitor its life cycle. It also acts as a level of abstraction over the framework and in case any change in future is required, this will limit the number of affected components.
- *Properties provider* - Keeps track of all common and user related options that are required for the correct operation of the Hibernate framework. The main properties this component is responsible to provide are the database login configuration for each user. It has to make sure that each request to the database

is executed with the correct database user so that the database can handle the access control properly.

- *Mappings provider* - This component is responsible to provide the "hbm" files that tell the Hibernate framework how to map the U-Sem entities(represented as Java collections) to the database.
- *Hibernate* - This component represents the Hibernate framework which is responsible to perform the actual interaction with the database in order to serve CRUD requests.

5.4.6 Concurrency View

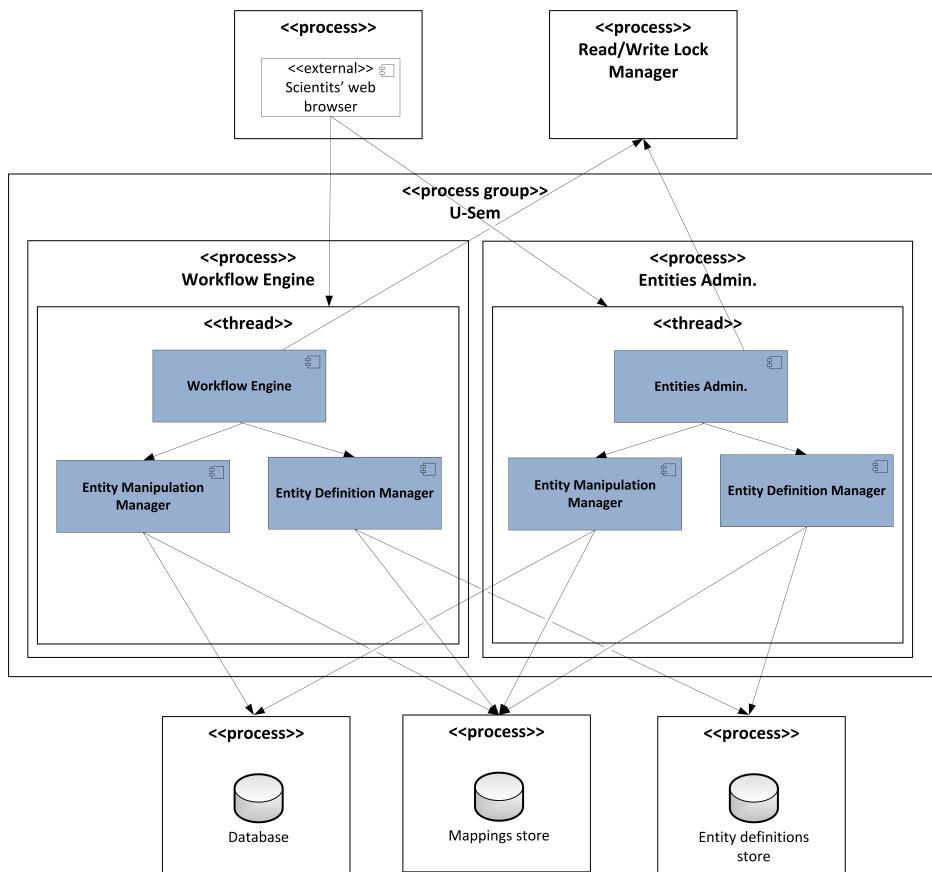


Figure 5.7: Context diagram of U-Sem

This section describes the concurrency structure of the feature. 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 5.7 illustrates the concurrency organization. The main functionality of the system is situated in the U-Sem process group. All U-Sem processes including the

storage processes operate concurrently. Workflow configuration and execution initiated by U-Sem clients and entity management by scientists can happen at the same time. This organization makes the solution flexible because if needed the **Workflow engine** process can be replicated independently from the **Administration**. However, this organization also introduces some problems that have to be solved.

Problems

Firstly, if the workflow engine is the middle of execution and the structure of the database is changed, then the workflow may fail unexpectedly and cause problems that are hard to detect and reproduce.

Secondly, every time the required resources for the data store interaction(entity definitions and mappings) are loaded the system has to make sure they are consistent with each other and also with the underling structure(schema) of the database. A problem can occur if the resources are loaded and modified simultaneously(simultaneous execution of workflows and entity definitions manipulations). It may happen that some of the files are loaded before the modification and others afterwards. This inconsistency can also lead to problems that are hard to detect and reproduce.

Solution

In order to solve the problem we propose a solution that is based on synchronization between the processes. The solution is based on the *Reader-Writer Lock* idea. It extends mutual exclusion locks by enabling concurrent read access to a shared resource but requires exclusive access for write operations [34].

Our solution maps to the *Reader-Writer Lock* idea as follows:

- The shared resource is the combination of the entity definitions, mapping and the database schema.
- The workflow engine acts as reader of the shared resource.
- The entity administration acts as writer of the shared resource.

As a result, multiple workflows can be executed simultaneously but when a change to the entities is needed it is executed exclusively. Therefore, the workflow engine is protected against loading the resources while they are inconsistent.

However, the standard *Reader-Writer Lock* in Java works only within the virtual machine². In our case we have to synchronize entire processes. Our research showed that there are already existing tools that provide this functionality [20]. The tool that is used in the proposed solution is Terracotta³. It introduces a process that is responsible for the lock and the other processes communicate with it in order to obtain the lock and use the resources.

²<http://docs.oracle.com/>

³<http://terracotta.org/>

5.5 Implementation

We implemented the proposed architecture in order to evaluate its applicability and capabilities. In this phase we basically implemented the system following the specification discussed in the previous section. Therefore, in this section we are discussing only on the most interesting parts of the implementation of the system.

5.5.1 Entities definition admin UI

In order to implement the endpoints(user interface) we used the **jQuery UI**, **Bootstrap** and **dynamic tree** technologies. The user interface consists of two main panels:

- **Entities list** - Illustrated on picture 5.8, this is the initial view exposed to the user. It provides a list of available entity definitions and enables users to create, update and delete entities.

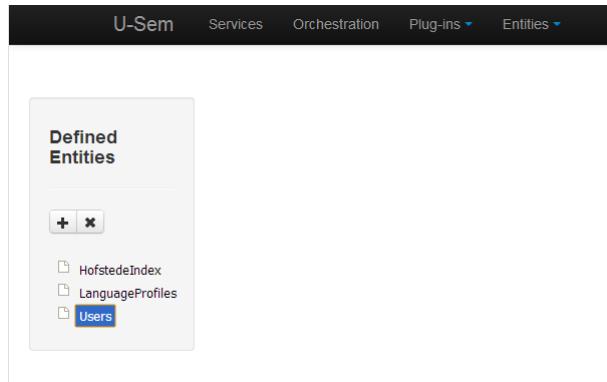


Figure 5.8: User interface for entity definitions management

- **Entity manipulation panel** - The implementation also provide a panel that is responsible to automate the process of defining entities. Shown on figure 5.9, this panel is used when an entity is created or updated.

5.5.2 RDFGears workflow configuration

We have also implemented two features that aim to benefit engineers when configuring workflows in RDFGears.

The first feature concerns the **Insert entity** component. According to the specification the engineer has to select and entity type that has to be stored and provide the corresponding data as inputs for the component. However, when selecting the entity type it is clear what are the inputs that will be required for the component because they have to correspond to the predefined entity structure. Therefore, instead of manually configuring the inputs of the component, the system automatically reads the entity type definition and sets up all the required inputs automatically.

The second feature is the preview panel for entity types. It represents a small dialog panel in the user interface that can be opened every time the engineer defining workflows need to refer to the structure of an entity type. As shown on figure 5.10,

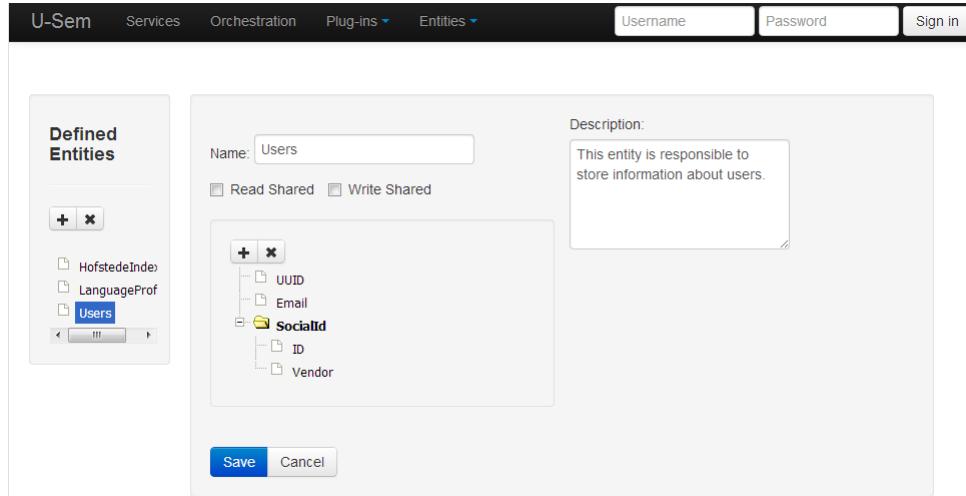


Figure 5.9: User interface for constructing entity definitions

it presents the entity types in the form of a tree. This feature is particularly valuable when users have to write queries because they can easily refer to the entity types and therefore, they do not have to remember them. It is also likely to prevent mistakes connected to not reproducing the entity types correctly in the queries.

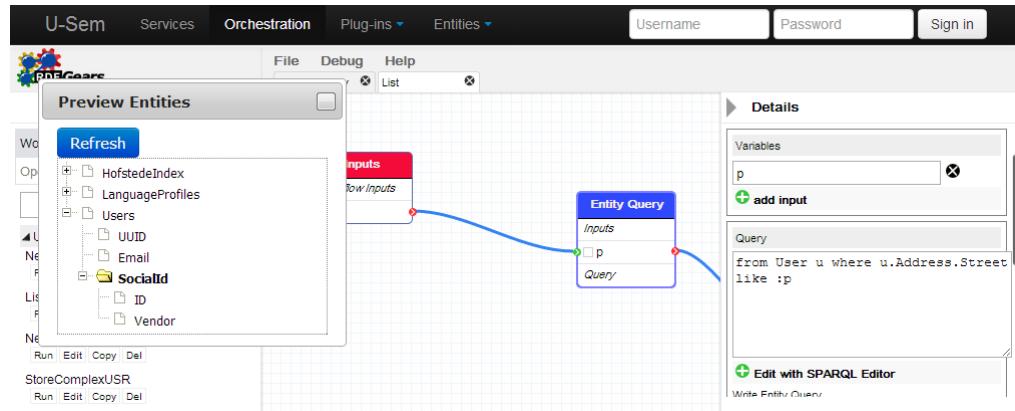


Figure 5.10: User interface that provides a quick preview of the entity definitions

5.5.3 Extended workflow compiler

We implemented the extension of the capabilities by extending some of the classes involved in the feature and keeping the original classes intact. In this way, in case of problem or if needed engineers can easily switch between the two implementations. We extended the following classes:

- *WorkflowNode* - the newly created node is responsible to act as a wrapper around the output node of the workflow. When executed it triggers the execution of not

only the output not but also to all the nodes with side effects defined in the workflow. The result of the execution is still the result of the execution of the output node. In this way we make sure that all nodes with side affects are executed as expected from the user. The already existing cashing mechanism ensures that each component is executed at most once which prevents any duplicate executions of components.

- *WorkflowLoader* - the new child class extends the implementation in two directions. Firstly, when the workflow is compiled the resulted data structure contains not only the components needed for the execution of the output node but also the components with the side effects and their dependent components. Secondly, the output node implementations is replaced by the extended WorkflowNode, containing references to the output component and all components with side effects.

5.6 Evaluation

In section ?? we discussed the issues solving which will theoretically save users time and efforts. In this section we want to evaluate how engineers benefit from the solution in a real world situation. In order to do that, we took an already existing user modelling service, implemented it using the newly proposed solution and at the end, discuss the improvements.

For the experiment we used the already implemented service for building language profiles based on Twitter. The basic idea of the workflow is to read all tweets for a particular user form the database, infer the languages (and level of knowledge) that the user knows, store the generated profile in the database and finally, provide it as a result of the execution of the service.

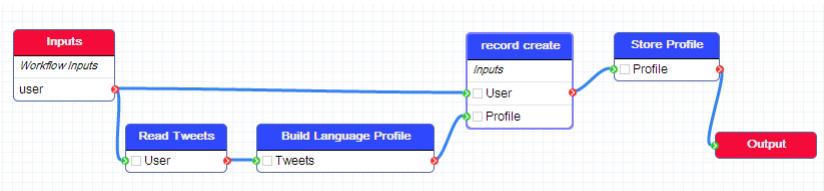


Figure 5.11: Workflow defining the service for building language profiles before the introduction of the feature

Figure 5.11 shows how the workflow looked like before the introduction of our solution. In order to implement that workflow scientists had to perform the following steps:

- Install and set-up a database.
- Create the necessary tables manually. This step is required every time the workflow is deployed on a U-Sem system.
- Implement specific components for queering the tweets and storing the generated profiles and deploy them to the U-Sem system.

- Make the store component return a copy of its input because it has to be part of the main branch otherwise it is optimized by the engine and not executed.

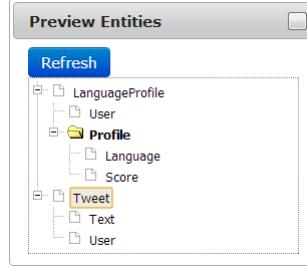


Figure 5.12: The entity definitions needed for building the service for building language profiles

Figure 5.13 shows how the workflow looks like after the introduction of our solution. In order to implement the workflow scientists had to perform the following steps:

- Define the entity types using the user interface illustrated on Figure 5.12.
- Reuse the components for querying and storing data by only providing the JPQL query and selecting the entity type for storing.
- Indicate the store component to have side effects and put it in a separate branch.
- Pack the workflow and entity types definitions into a plug-in for easy deployment.

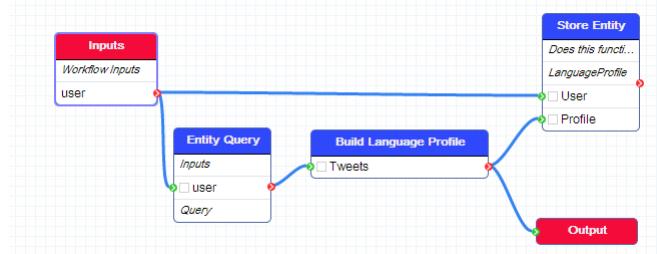


Figure 5.13: Workflow defining the service for building language profiles after the introduction of the feature

We believe that using the proposed solution saves a lot of time and efforts to the engineers defining the service for the following reasons:

- No installation and configuration of the database required
- No need to create the database schema manually. It is done automatically by the user interface

- No programming required for implementing storage and query components
- No need to convert RGL values so that they can fit into the database
- Queries are written in JPQL which is a higher level language compared to SQL and queries are likely to be faster and easier to write. Additionally, the user interface that shows the structure of the entities at any time makes writing queries easier (engineers do not have to know it by heart) and prevents mistakes that come from not knowing the structure of the entities.
- The workflow looks more intuitive because the **Store** component can be left in a separate branch and does not have to return a copy of the input data like in the initial situation.
- The workflow can be deployed on a U-Sem system by simply installing the plug-in. No configuration required.

5.7 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 the following items:

- One of the limitations that our approach introduces is the way semantics of data are defined. Currently, users can describe them only in text form (in the description field). The way entities are described is left to the engineer, there are no automated mechanisms that manage or assist the process. Therefore, more sophisticated (formal) approach for describing semantics of entities might be beneficial.
- The aim of this feature is to simplify the work with persistent data in U-Sem. However, introducing this abstraction over the storage functionality we also have reduced the flexibility to some extend. One of the side effects is the inflexibility in terms of transactions management(begin, commit, rollback). In the current situation users do not have any means to manage the database transactions and they are tied to the way the engine is configured. Currently, our research showed that this is not a problem but in future some users might need to have the power to control the transactions to the database. Therefore, introducing a mechanism that can enable that efficiently might be an interesting research topic.
- Most of the data manipulation components(except for the "Insert entity" component) require users to enter JPQL query. These queries can get quite complex and as a result users may make some mistakes when writing them. Currently, the solution does not provide any facilities that can validate these queries. Users are notified about the mistakes only when they try to execute the workflows and they fail. As a result, this process may cost a lot of time to users until they finally end up with the correct queries. Therefore, in future the system might be further improved by introducing functionality for auto-completion and validation of the JPQL queries.

- Currently, the RDF Gears engine does not provide any guarantees for the order of the execution of branches. This is not a problem for components that do not have side effects. However, with the introduction of the new components that interact with the database developing an mechanism to control the execution order of branches within a workflow makes sense and might be a helpful future addition to the solution.
- The system is currently targeted to be used mainly as a research tool and there are no requirements for supporting large number of users and high loads. However, if the system is to be used in such demanding environment then it might be worth to investigate and if needed improve the performance, scalability and availability properties of the solution. **maybe propose solutions like L2 cache and high availability db setups.**

Chapter 6

Conclusions and Future Work

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

6.1 Contributions

This thesis provides the following contributions:

The first group of contributions concern the Plug-in environment feature.

- Design and implementation of the dynamic component model for RDFGears based on OSGi.
- Design and implementation of the plug-in collaboration system.
- Design and implementation of a plug-in template that simplifies the creation of U-Sem plug-ins under Eclipse.

The second group concern the Data Management feature.

- Design and implementation of the multi-tenant ORM framework for dynamic, virtual entities based on Hibernate.
- Designing a mechanism that enables scientists to collaborate on data level.
- Extension for the RGL language and RDFGears engine that enables the system to deal with components that have side effects.
- Integration with the Plug-in environment feature that enables simplified deployment of services to U-Sem.

6.2 Conclusions

6.3 Discussion/Reflection

Apart from user modelling, the system can also be considered as an extension of RDFGears that can benefit engineers working in other fields **cite**. Additionally, because the features are designed to be loosely coupled from the rest of the system(implemented as separate sub-systems providing APIs) they may be used to extend and provide the functionality for other workflow engines.

6.4 Future work

The future work can be discussed from two perspectives. The first one addresses how each of the implemented features can be improved in the future. We already discussed these in the sections dedicated for each of the features. From the higher level perspective an obvious future work is the implementation of the other features that engineers indicated to bring important improvements for their everyday job. These features are

...

Bibliography

- [1] Fabian Abel, Ilknur Celik, Claudia Hauff, Laura Hollink, and Geert-Jan Houben. U-sem: Semantic enrichment, user modeling and mining of usage data on the social web. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.
- [2] Fabian Abel, Nicola Henze, Eelco Herder, and Daniel Krause. Interweaving public user profiles on the web. In *User Modeling, Adaptation, and Personalization*, pages 16–27. Springer, 2010.
- [3] James Frederick Allen. A plan-based approach to speech act recognition. Technical report, Dept. of Computer Science, University of Toronto, Canada, 1979.
- [4] Mark Assad, David J Carmichael, Judy Kay, and Bob Kummerfeld. Personisad: Distributed, active, scrutable model framework for context-aware services. In *Pervasive Computing*, pages 55–72. Springer, 2007.
- [5] Neil Bartlett. Osgi in practice. *Bd (January 11, 2009)*, 2009.
- [6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.
- [7] Peter Brusilovsky. Knowledgetree: A distributed architecture for adaptive e-learning. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 104–113. ACM, 2004.
- [8] Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. *The adaptive web: methods and strategies of web personalization*, volume 4321. Springer-Verlag New York Incorporated, 2007.
- [9] Xia Cai, Michael R Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 372–379. IEEE, 2000.

- [10] Zhenbang Chen, Zhiming Liu, Anders P Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Science of Computer Programming*, 74(4):168–196, 2009.
- [11] Marc Cheong and Vincent Lee. Integrating web-based intelligence retrieval and decision-making from the twitter trends knowledge base. In *Proceedings of the 2nd ACM workshop on Social web search and mining*, pages 1–8. ACM, 2009.
- [12] Philip R Cohen and C Raymond Perrault. Elements of a plan-based theory of speech acts. *Cognitive science*, 3(3):177–212, 1979.
- [13] Drew Dean, Edward W Felten, and Dan S Wallach. Java security: From hotjava to netscape and beyond. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 190–200. IEEE, 1996.
- [14] Gero Decker and Alistair Barros. Interaction modeling using bpmn. In *Business Process Management Workshops*, pages 208–219. Springer, 2008.
- [15] Oscar Dieste, Natalia Juristo, and Forrest Shull. Understanding the customer: what do we know about requirements elicitation? *Software, IEEE*, 25(2):11–13, 2008.
- [16] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. *Technical Reprot, University of Klagenfurt, Austria*, 1994.
- [17] E.Feliksik. Rdf gears, a data integration framework for the semantic web. Master’s thesis, TU Delft, Netherlands, 2011.
- [18] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-jvm: a java virtual machine for component isolation in osgi. In *Dependable Systems & Networks, 2009. DSN’09. IEEE/IFIP International Conference on*, pages 544–553. IEEE, 2009.
- [19] Li Gong, Gary W Ellison, and Mary Dageforde. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley Professional, 2003.
- [20] Soumeya Leila Hernane, Jens Gustedt, and Mohamed Benyettou. A dynamic distributed algorithm for read write locks. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 180–184. IEEE, 2012.
- [21] Ann M Hickey and Alan M Davis. A unified model of requirements elicitation. *Journal of Management Information Systems*, 20(4):65–84, 2004.
- [22] Mei Hui, Dawei Jiang, Guoliang Li, and Yuan Zhou. Supporting database applications as a service. In *Data Engineering, 2009. ICDE’09. IEEE 25th International Conference on*, pages 832–843. IEEE, 2009.

- [23] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2):W729–W732, 2006.
- [24] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-based software engineering. In *Theoretical Aspects of Computing–ICTAC 2005*, pages 70–95. Springer, 2005.
- [25] Thomas A Joseph and Kenneth P Birman. Reliable broadcast protocols. Technical report, DTIC Document, 1988.
- [26] Andreas M Kaplan and Michael Haenlein. Users of the world, unite! the challenges and opportunities of social media. *Business horizons*, 53(1):59–68, 2010.
- [27] Joachim Karlsson and Kevin Ryan. A cost-value approach for prioritizing requirements. *Software, IEEE*, 14(5):67–74, 1997.
- [28] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.
- [29] Gavin King, Christian Bauer, M Rydahl, Emmanuel Bernard, Steve Ebersole, and H FERENTSCHIK. Hibernate: Relational persistence for idiomatic java. Technical report, Technical Report, Available online at: http://www.hibernate.org/hib_docs/v3/reference/en/html_single, 2010.
- [30] Alfred Kobsa. Generic user modeling systems. *User modeling and user-adapted interaction*, 11(1-2):49–63, 2001.
- [31] Alfred Kobsa and Josef Fink. An ldap-based user modeling server and its evaluation. *User Modeling and User-Adapted Interaction*, 16(2):129–169, 2006.
- [32] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.
- [33] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 21–30. ACM, 2009.
- [34] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110. ACM, 2009.
- [35] Jeff Linwood and Dave Minter. *Beginning Hibernate*. Apress, 2010.
- [36] Shiyong Lu and Jia Zhang. Collaborative scientific workflows. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 527–534. IEEE, 2009.
- [37] Bertram Ludascher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

- [38] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.
- [39] Eric Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.
- [40] Frank Moisiadis. The fundamentals of prioritising requirements. In *Proc Systems Engineering, Test & Evaluation Conf, Sydney, Australia*, pages 108–119. Citeseer, 2002.
- [41] Lars E Olson, Carl A Gunter, and P Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 289–298. ACM, 2008.
- [42] Elizabeth J O’Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.
- [43] Jon Orwant. Heterogeneous learning in the doppelgänger user modeling system. *User Modeling and User-Adapted Interaction*, 4(2):107–130, 1994.
- [44] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [45] Pierre Parrend and Stéphane Frénot. Security benchmarks of osgi platforms: toward hardened osgi. *Software: Practice and Experience*, 39(5):471–499, 2009.
- [46] Gilda Pour. Component-based software development approach: new opportunities and challenges. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 376–383. IEEE, 1998.
- [47] Gilda Pour. Enterprise javabeans, javabeans and xml expanding the possibilities for web-based enterprise application development. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 31. Proceedings*, pages 282–291. IEEE, 1999.
- [48] Gilda Pour, Martin Griss, and John Favaro. Making the transition to component-based enterprise software development: Overcoming the obstacles-patterns for success. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pages 419–419. IEEE, 1999.
- [49] Elaine Rich. Building and exploiting user models. In *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 2*, pages 720–722. Morgan Kaufmann Publishers Inc., 1979.
- [50] Elaine Rich. User modeling via stereotypes. *Cognitive science*, 3(4):329–354, 1979.
- [51] Gr C Roper-Lowe and JA Sharp. The analytic hierarchy process and its application to an information technology decision. *Journal of the Operational Research Society*, pages 49–59, 1990.

- [52] Nick Rozanski and Eóin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley Professional, 2011.
- [53] Robert C Seacord. Software engineering component repositories. In *Proceedings of the International Workshop on Component-Based Software Engineering*, 1999.
- [54] Marojahan Sigiro. Dynamic user interface supporting type analysis for rdfgears. Master's thesis, TU Delft, Netherlands, 2011.
- [55] Vladimir Silva. *Practical Eclipse Rich Client Platform Projects*. Apress, 2009.
- [56] Saurabh Srivastava, Michael Hicks, Jeffrey S Foster, and Patrick Jenkins. Modular information hiding and type-safe linking for c. *Software Engineering, IEEE Transactions on*, 34(3):357–376, 2008.
- [57] Mengtao Sun and Gang Tan. Jvm-portable sandboxing of java's native libraries. In *Computer Security—ESORICS 2012*, pages 842–858. Springer, 2012.
- [58] Ke Tao, Fabian Abel, Claudia Hauff, and Geert-Jan Houben. Twinder: a search engine for twitter streams. In *Web Engineering*, pages 153–168. Springer, 2012.
- [59] Andre LC Tavares and Marco Tulio Valente. A gentle introduction to osgi. *ACM SIGSOFT Software Engineering Notes*, 33(5):8, 2008.
- [60] Wolfgang Wahlster and Alfred Kobsa. *User models in dialog systems*. Springer, 1989.
- [61] Petia Wohed, Wil MP van der Aalst, Marlon Dumas, Arthur HM ter Hofstede, and Nick Russell. On the suitability of bpmn for business process modelling. In *Business Process Management*, pages 161–176. Springer, 2006.
- [62] Eoin Woods and Nick Rozanski. The system context architectural viewpoint. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 333–336. IEEE, 2009.
- [63] Jing Wu, Yi Mu, Jennifer Seberry, and Chun Ruan. Access policy sheet for access control in fine-grained xml. In *Embedded and Ubiquitous Computing—EUC 2005 Workshops*, pages 1273–1282. Springer, 2005.
- [64] Yong-chao ZHAO, Ning Zheng, and Ying-long GE. Design and deploy of the connection pool in java. *Application Research of Computers*, 6:077, 2004.

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 serifed 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