TÉCNICO LISBOA

Collaborative Platform for Analysis of Software Systems

Catarina Isabel Carvalho Santana

Thesis to obtain the Master of Science Degree in

Information and Software Engineering

Advisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof./Dr. Lorem Ipsum
Advisor: Prof./Dr. Antnio Manuel Ferreira Rito da Silva
Members of the Committee: Prof./Dr. Lorem Ipsum

October 2015

Acknowledgments

“In the end, I’m able to look back without shame or regretful nostalgia and think, *You did something great. And something new will come around. Or not. Either way, do the work you love. And love yourself. That’s all you can do in this world in order to be happy*”

- Felicia Day in *“You’re Never Weird on the Internet (Almost)”*

Abstract

The Software Architectures course teaches students the most important concepts on design and architecture of software systems and makes students apply these concepts to real, complex software systems. Organizing knowledge and applying theory to practice is not an easy task, and students often need to ask questions and discuss. State-of-the-art on collaborative, social software and knowledge structuring is analyzed and a social platform is proposed to solve the problems adjacent to this course.

Keywords: social software, knowledge structuring, collaborative platform, reputation systems, tagging systems, ontology, taxonomy

Resumo

O teu resumo aqui...

Keywords: as tuas palavras chave

Contents

List of Tables

List of Figures

Chapter 1

Introduction

NEEDS MOAR WORK.

Analyzing and discussing big, real, open-source and highly complex software system is a very important part of the Software Architectures course. In the course context, students must apply concepts and techniques for design and analysis of software architectures to descriptions of real systems.

However, applying these concepts and techniques is not a very easy task, and often students have questions and doubts regarding these descriptions. These questions sometimes require not only consulting the course bibliography, but also discussing with peers or asking questions to teachers.

The following document is organized as follows: Section **??** provides a more detailed description of the problem presented last paragraph and presents a possible solution for the problem. Section 4 presents the state-of-the-art on the components of the solution. Section **??** describes a possible solution for the problem and Section **??** describes how the implemented solution will be assessed.

Chapter 2

Problem Description

The course of Software Architectures teaches students the most important concepts in the field of software architectures and applies these concepts to real-life software systems.

The practical component of this course (where the theory is applied) is done by analyzing documents/articles that describe the architectures of real-life systems and applying the concepts learned in theory lessons: extracting stakeholders, scenarios, tactics, views, etc. This analysis is either done by students with the help of the teacher, during practical classes, or in work groups of usually three students, that read, discuss and analyze software descriptions together and present their results to both the teacher and the rest of the class.

Understanding the analysis done and how it was done is very important, since it means students can apply the concepts learned not only in the written exam to pass the course, but also in the future, in other real-life software systems. However, there are several issues regarding the application of theory concepts in this course:

- Each year, there are around one-hundred students signed in the course. All these students have roughly the same Computer Science background knowledge. However, not all students have the same maturity: while some may quickly understand what is taught in theory lessons, others may need some more time to assimilate what was taught.
- The output of the analysis done to software descriptions (the scenarios, etc. extracted) is not available in a consistent way: the analysis done in class is available for students if they took their own notes, and the analysis done by groups is available if:
    1. Students took notes of their colleagues' presentations
    2. Groups share their presentation slides among them. Slides may include errors pointed out by the teacher, but not corrected.
- The case description is usually a fairly long document (from ten to twenty pages approximately). The task of carefully reading and understanding all the text and do a mapping between the concrete descriptions in the text and the abstract concepts learned in theory classes is not easy, as the parts of the text that map to concepts are not always evident.
- The architectural elements extracted from a single software description are usually scattered along the whole text, and it is not evident the connection between all the elements.

Chapter 3

Objectives

To solve the mentioned problems, we intend to develop a platform, where students and teachers collaborate in the analysis and synthesis of the case descriptions, ending on a structured representation of the case descriptions, that is hooked on the concrete description.

The platform will provide ways for annotating the text on the case descriptions, which will help students organizing their thoughts and creating the structure.

To motivate the students to participate in this platform and to increase the collaborative component, elements of social software will be used (such as reputation and tagging systems), to promote, among other things, collaboration, mutual aid, discussion, learning, and even some healthy competition between students.

The existence of this collaborative platform provides not only a way for students to discuss, ask questions and consolidate their knowledge, but also a unique place where their study materials are stored and organized, facilitating their studies.

Chapter 4

Related Work

The platform to develop can be thought of a social software, where different people communicate and collaborate to provide a structured representation of a software description. The next subsections provide an overview on the state-of-the-art of two main aspects:

- **Collaborative work**, presenting the state-of-the-art on social and collaborative aspects of software. This includes literature on:
  - The Honeycomb Framework (a framework that generalizes the most important components of social software);
  - Persuasive Software (systems that have impact on users behavior and thoughts, which is the case of the platform to develop);
  - Roles in Social Networks (describing the types of users of social software);
  - Reputation Systems (attribution of scores to users to provide a motivational component on the platform).
- **Knowledge Structuring**, presenting the state-of-the-art on ways of structuring information. This includes literature on:
  - Collaborative tagging systems (assigning keywords to documents or parts of documents);
  - Semi-structured content (providing a way of structuring knowledge without such strict rules as, for example, an ontology or a taxonomy);
  - Ontology Learning (extracting knowledge from text).

4.1 Honeycomb Framework

The term “Social Software” is used in very different contexts [1], and therefore, there are different definitions for it, given by different authors [2, 3, 4, 5, 6].

The definition of Social Software as “systems that allow people, in their particularities and diversity, to communicate (interact, collaborate, exchange ideas and information) mediating and facilitating any kind of social relationship and favoring the emergence of a collective wisdom and a bottom-up organization” [1] applies to our collaborative platform, in the sense that it should allow a large number of students, with different personalities and opinions, to ask questions to their colleagues, give opinions, receive feedback, etc, in order to correctly identify the architectural elements in a description of a software system, and therefore reach a complete and correct analysis of that system.

The Honeycomb Framework was proposed to illustrate the seven elements that give a functional definition for social software [7]:

- **Identity:** the unique identifier of a user within the system;
- **Presence:** resources that allow knowing if certain identity is online;
- **Relationship:** way to determine how users can relate/are related to others;
- **Reputation:** way of knowing the status of a user in the system;
- **Groups:** possibility to form communities of users that have common interests;
- **Conversation:** resources for communication among the users (synchronous and/or asynchronous);
- **Sharing:** possibility of sharing objects that are important to the users (videos, images, etc);

The Identity appears at the center of the framework, because it is the most basic requirement of any social system.

Although this framework identifies the major elements of a social system, it does not, in fact, specifies whether a system is social or not, a software system may implement all of these elements and still not be social.

Analogously, a system can be social with just a couple of elements implemented. So far, the only way to know if a system is social is to check if it complies with the definition (or with one of them).

In the platform to develop it is possible to identify the following elements of the framework:

- **Identity:** The identity is the central element of the framework and it is essential to the platform, as each student is a unique person and, therefore, must have a unique identification inside the platform. Since each student has already got a unique student number inside the whole university, the student number should be used for identification. The name of the student should also be added for purposes of identification, since it helps other users knowing who are they communicating with (it is easier to know who a student is if his name is available).
- **Reputation:** To assess the quality and relevance of the participations in the platform, a reputation system must be used, as discussed in 4.4.
- **Groups:** One of the evaluation methods of the Software Architectures' course is the realization of group works, where students form work groups, and each group will have to identify architectural elements in a description of a software system. The notion of these work groups must be present in the platform, since each group is a restricted set of people that are working towards the same goal: Correctly understand and identify the architectural elements in the description.
- **Relationship:** Although there is already an implicit relationship inside the platform (all students are colleagues of each others), this relationship has a specialization. Since students are divided in work groups, students inside a work group have a different relationship they are group colleagues, and this relationship must be differentiated. As the teacher will also be a member of the platform, the relationship student-professor must always be present as well: a student does not have the same kind of interactions he has with a colleague when interacting with a teacher therefore this kind of relationship must be present in the platform.
- **Conversation:** The existence of discussions around elements to add/already added to the platform provides a way of communication between the users.

- **Sharing:** Within the discussions around elements, it should be possible to share different kinds of objects (articles, links, etc), since they may be helpful to understand or complement what is being discussed. Since there are groups inside the platform, there should be present different levels of sharing:

    1. *Group level:* Shared objects are only available for a single work group. This way, groups can share documents, links, etc., that are important for completing their assignments, or help understand the description document;
    2. *General level:* Documents are shared inside a discussion, an added element, or as an answer to a question asked, and are available to all students in the platform;

- **Presence:** Considering that all communications are asynchronous, the notion of presence is not strictly necessary in the platform. However, it could be added, as a feature, a visible list of people online (logged in the platform) similar to the ones present in many discussion forums.

4.2 Persuasive Software Design Patterns

The term “persuasive technology” is used to describe computer systems that have an impact on user’s thoughts and may even lead to changes in their behavior [8, 9].

There are three different possible outcomes for a persuasive system:

- Reinforcement - making current attitudes resistant to change;
- Changing Outcome - changes in a person’s response to an issue;
- Shaping outcome - formulate a pattern for a situation where one did not exist before [10];

The main users of the collaborative platform to develop will be MSc students attending the ’Software Architecture’ course. Even though they are familiar with some of the terms used in the context of the course, new terms are introduced as the course progresses. This platform is persuasive in the sense that it must change the way its users will look at a description of a Software System: reading is not enough, it is necessary to relate what is read with the terms and concepts learned in the classes. Therefore, the most important outcome for this platform is the Changing Outcome: When a description of a system is presented, the students most not only read it, but also relate it to the concepts learned and use the platform to try and extract the correct scenarios and views from it.

The concept of social influence shows that in daily activities, human beings often influence or are influenced by the actions of other people. One’s opinion might be a reflection of someone else’s opinion, which is also directly or indirectly influenced by cultural norms, mass media, social networks, etc [11]. Therefore, social influence is a change in one’s behavior (or attitudes or beliefs), caused by external pressures [12].

In our context, we want the students to contribute to the platform. Either by adding new elements (scenarios, views), discussing the elements added by their colleagues, or asking/answering questions. The platform should provide a way to do this, by allowing:

- Insertion and edition of new elements to the conceptual structure, done by any student;
- Comments (and responses to comments) on an element inserted;
- Discussion Forums and Q&A systems

While some students will want to start contributing to the platform right away, some of them will not initially feel compelled to that. However, as their colleagues and friends add contributions, and since one's actions are somehow influenced by other people, these students may probably start contributing to the platform as well. Even in a social context, because people do not live in isolation, students may talk, ask/answer questions and discuss in person about the system being analyzed in the platform. The outcome of these social interactions may be a new, relevant, contribution to the platform.

Four design patterns are proposed for persuasive systems, in order to introduce social influence in software features: Social Learning and Facilitation (SLF), Competition (COM), Cooperation (COO) and Recognition (REC) [13].

4.2.1 Social Learning and Facilitation (SLF):

The concept of Observational Learning, introduced by the Social Learning Theory, states that people's behavior and skills are augmented through the act of observing others [14], whilst the concept of Social Facilitation states that people's awareness of being observed/evaluated by others also has an influence on their behaviors [15]. Therefore, the Social Learning and Facilitation pattern has the main purpose of enabling and enhancing the design of software features that allow to visualize the presence of other people, with the motivation that it is easier for individuals to pursue their goals if there is a clear awareness of other people pursuing the same goal and facing the same issues [13].

In the context of our collaborative platform, it is necessary to persuade the students to add contributions as they may, at first, not feeling confident enough to add their own contributions, whether they are new elements, comments, questions/answers, etc. The platform should provide a way to visualize the elements recently added or that are currently under discussion as students see their colleagues contributing towards the same goal, they may want to add their own contribution.

Considering that the users of the platform will be students, the Social Learning should have more emphasis over the Social Facilitation. Students have different rhythms of learning, and sometimes a concept may not be correctly understood at first. Asking questions or discussing elements of the analysis will help in the correct assimilation of concepts, and therefore it must be encouraged.

The Social Facilitation comes in the sense that moderation is necessary: while students are encouraged to contribute and ask questions in order to learn, they should strive to add relevant comments and ask relevant questions, as their contributions are seen by everyone.

4.2.2 Competition (COM):

Human beings have a natural drive for competition. Therefore, it can be used as a social support guideline, to motivate users to adopt attitudes and behaviors [9]. The Competition pattern has the main purpose of using competitive elements, such as ranks, scores and levels, that allow users to compare their performance with others, and adjust their target goals based on these. However, as some people might see competition as a motivational factor for improvement, other people may see it as a source of anxiety therefore, participation in a competition must always be voluntary [13].

Adding competitive elements to a social system will, as said above, trigger the human being's natural drive for competition. Adding a score for each student and a rank hierarchy based on the range of scores will start a hopefully healthy competition between students. Each contribution should have a positive score, given either by the teacher or by the colleagues. Less correct contributions should have a lower score, but positive nonetheless. When a student adds something to the platform, it means that he is making an effort to apply the concepts learned in the classes. As said before, not all students can do this correctly at first, and therefore if one of them specifies an incorrect scenario, for example, it

shouldn't have a negative score: trying and failing is a way of learning. In addition, when another student adds a contribution to correct his colleagues, he is also gaining score points, and therefore, increasing the competition.

Even this kind of competition, that is supposed to be healthy and motivating, can be a source of anxiety for some people. Therefore, it should be optional for students to be assigned a score and receiving points, or even to show their identity when contributing (add anonymity to the platform).

4.2.3 Cooperation (COO)

Besides competition, reciprocity is also another human beings' natural drive [16, 17], and cooperation between people has been proven to cause positive impact on user behaviors [18]. The Cooperation pattern has the main purpose of providing software features that allow users to engage in mutual goals and provide ways for them, supporting each other in reaching their goals. The motivation for this pattern is that it is easier to cooperate if there is a clear awareness of what needs to be done, what other people are doing, and how far they are in achieving their goals. Group discussion supports these needs and encourages giving and receiving support on the matter. However, some people prefer to perform alone, and therefore cooperation must not be forced [13].

The students that will use this platform will be working towards a main goal: correctly understanding and correlating what was learned in the Software Architectures classes with real examples of software systems. But concerning the analysis of a single software system, there are several concepts and structures to identify on a single description. Students may be having some trouble identifying these elements, and therefore the system must provide a way of defining goals. A goal, for example, could be "Add a new scenario for Availability".

The goals system should work similarly to a discussion forum: Students can add, delete or change goals (changing the description or the title, for example), and for each goal it is possible to send messages to discuss it and ask or answer questions. As discussion around the goal progresses, students collaborate with each other by answering questions asked by their colleagues or correcting what some other student wrote before. Discussions can converge, then, to a correct identification of architectural elements in the platform. A goal should be able to be marked as "in progress" or "complete", to indicate that discussions around that goal are still in progress or have finished. Goals can also be split in sub-goals if the discussion around a single goal becomes too big and complex, it might be wise to be able to split the discussion in sub-goals, each with its own discussion so that when all sub-goals are complete, the main goal will also be nearly or fully complete.

4.2.4 Recognition (REC)

In addition to competition and cooperation, recognition is introduced as another interpersonal motivational factor. This happens because people take pleasure in having their efforts recognized and appreciated by others, in order to feel fulfilled for their achievements [17]. The Recognition pattern has the purpose of providing software features that enable users to get recognition from their peers. The motivation for this pattern is that users, when working towards a goal, they sometimes need a reason to focus on that same goal. Having their efforts recognized may be a good reason to keep the good work, and therefore systems should provide opportunities for public recognition of top achievers [13].

As mentioned before, the users of this platform are students that are pursuing a Masters degree. And just like any other human being, students like to feel that their efforts are recognized. Not only recognition leads to a sense of fulfillment, but also it is a motivational factor to keep up the good work.

Since the system should have a score system, this score should be used as a way of creating a weekly spotlight: by keeping track of the amount of points scored by each student along the week, as the week ends, the list of the Top 10 students with most points gathered in that week should be visible when entering the platform. The list of all students, in decreasing order of weekly points, could also be consulted.

Recognition is then achieved when students see their names in the spotlight. In addition to that, there may be some (hopefully healthy) competition for a spot in the Top 10 - as all, or at least most students will want to have their names on the spotlight.

4.3 Roles in Social Networks

A survey on the state-of-the-art regarding identification of roles in social networks ([19]) starts by stating the difference between a "role" and a "position", by giving a definition for both of these concepts:

- A *position* is a "well-defined place" of an individual in a social structure, and is usually associated with certain attitudes, mental health, knowledge, etc., of individuals [19, 20].
- A *role*, in a social structure, is a set of expectations for an individual in a certain position. For example, the role of "secretary" is associated to what secretaries are expected to do [19, 21].

The survey divides the roles in two categories:

- *Non-explicit roles*: roles that are not defined a-priori
- *Explicit roles*: predefined types of actors in the social network, such as "experts" or "influencers".

Methodologies for inferring *non-explicit* roles analyze the structure of the social network and communications between users, in order to identify patterns in communication that allow the identification of roles. Common approaches are based on blockmodels [22, 23] and probabilistic bayesian models [24, 25, 26].

Methodologies for identifying *explicit* roles in a social network compare the activity of the user inside a social network against the criteria that characterizes each role. For example, techniques for identification of "experts" (users that answer correctly to questions asked) inside forums [27] analyze, among other criteria, the number of answers given by the user and how many different people a user answers.

In the platform to develop, it is possible to define a priori the two main roles:

- Student: it is the role assigned to most of the users. Students are the most active users, and their contributions to the platform include tagging parts of the document, adding and editing parts of the semi-structure, asking and answering questions and rate other students' contributions.
- Teacher: this role is assigned to a very small number of users in the platform, but there must be at least one teacher present. Their number of contributions is smaller and they are mostly for correcting (through adding comments to the elements added, or answering questions) the contents added by the students to the platform. Teachers can also give ratings to the contents of the platform. It is up to the teachers to define the weight of their ratings on a case by case strategy.

The methodologies for identification of explicit roles in a social network [27, 28] have in consideration social networks around the web, with hundreds or even thousands of users and where the roles cannot be easily assigned. Given the small amount of users expected in the platform, the roles of Student and Teacher will be assigned in the moment of registration, and there is no need for this methodologies.

A methodology similar to the ones described for identification of experts [27] could possibly be added to the platform. By analyzing user interactions and the ratings saved in the reputation system, a role similar to “expert” could be assigned to some students to identify the ones that do the most relevant contributions to the platform. Students with the role of “experts” could have some responsibilities assigned to them, in order to help other students understanding and applying the concepts. However, given the small number of users in the platform, the number of interactions to analyze may not be enough to do a correct inference.

4.4 Reputation Systems

Reputation systems are of extreme importance for certain kinds of applications, namely e-commerce websites (Amazon, eBay, etc.). In these websites, buyers and sellers execute money transactions and therefore, the reputation of a seller will denote the degree of trust that possible buyers will have in them [29].

Most of the existing literature focuses on these types of applications, and analyses them according to certain defined criteria [29, 30].

For the platform to develop, reputation does not play such an important role as it does for e-commerce and similar applications. For this context, reputation will add a motivational component to the platform: Students rate their colleagues’ contributions and ratings are aggregated to calculate a reputation score for each student. Students should strive for getting and keeping a high reputation score. Contributions to the platform include parts of the description that are identified as semantically significant for the software architecture, elements added to the semi-structure, questions and answers added to the Q&A section, comments added (in the discussion forums, or to the elements added).

4.4.1 Motivation:

A study was conducted [31], concerning the lack of participation in Moknowpedia, a Wiki system: People were not motivated enough to overcome any obstacles (lack of time, lack of a structure for the article, etc.) and start contributing to the wiki. A reputation system was added to the wiki in order to solve these problems and improve both content quality and quantity. The system used MediaWiki, a widely used wiki software, and CollabReview, a Java-based web application for reputation management [32].

Results show an increase of 62% in the number of article revisions and an increase of 42% in the number of viewed articles. In the end of the four-month evaluation phase, during which the number of users fluctuated between 18 and 16, 237 reviews had been added to the databases. Overall, the users accessed the wiki more often, read more articles and made more contributions.

This shows the importance of a reputation system in a collaborative system, even if its role is only motivational, and given the similarities between Moknowpedia and the platform to develop (small number of users with similar backgrounds), it is expected that the presence of a reputation system in the platform leads to significant numbers of contributions.

4.4.2 Requirements and Features:

A framework for analysis of reputation systems is proposed in [29], where the general requirements for reputation systems are elicited and the corresponding features needed for their fulfillment. Since these requirements try to include all possible kinds of contexts for reputation systems, not all requirements apply to all systems. Given the context of the platform, the following requirements are identified for its reputation system:

- **R1 (Ratings should discriminate user behavior)** and **R2 (Reputation should discriminate user behavior):**

  The ratings (score that a student assigns to a participation from other student) and the reputation score (the score assigned to a student after processing all ratings given by other students) should have a range of possible values that discriminate all the possible user behaviors, from a student with very few and/or less relevant contributions to a student with many and/or very relevant ones.

- **R3 (The reputation system should be able to discriminate "incorrect" ratings):**

  Despite the small size of the platform to develop, malicious users are present everywhere. Intentionally inaccurate ratings can be given to contributions, in order to increase or decrease a user's reputation score. Therefore, some management for this kind of situations should be present in the platform.

- **R11 (Users should not be able to directly modify ratings)**, **R12 (Users should not be able to directly modify reputation values)** and **R13 (Users should not be responsible to directly calculate their own reputation):**

  When being processed, data should not be accessible to users. The reputation score of a student should be calculated by the system and neither this value, nor the values of the ratings attributed should be modifiable by the users.

To satisfy the identified requirements, the reputation system should implement the following features:

- **F1 (Trust/Distrust):** The metrics for reputation and ratings should discriminate the range of all possible user behaviors, from reliable to questionable. In this context, a range of values should be used, where the lowest value represents, as said above, a student with very few and/or less relevant contributions, and the highest value represents a student with many and/or very relevant ones.

- **F2 (Absolute Reputation Values):** The reputation score of a student should not be calculated with respect to another students. Scores should be absolute and calculated independently for each user by the reputation system.

- **F3 (Origin/Target):** To prevent students from self-rating themselves and to control malicious ratings (for example, a student with a lower reputation score could assign lower ratings to their colleagues' participation in order to lower their scores), the origin and the target (students) of each rating given should be identified in the system. By comparing origin and target self-ratings are prevented, and by assigning a weight proportional to the origin's reputation score (a rating given by a student with a lower reputation score will have a smaller weight when processing) malicious users are controlled.

4.4.3 Components of a Reputation System

In [33, 30], reputation systems are divided in four components and it is defined a set of criteria for evaluation:

- **Input:** the process of collecting reputation information from information sources

- **Processing:** the procedure of computing and aggregating the reputation information

- **Output:** the dissemination of the reputation information

- **Feedback Loop:** collection of feedback of the output (review of the review). Does not apply to the context of the platform.

1. **Input:** Concerning input, a series of evaluation criteria is defined [30], from which the following apply to the platform:

   - *Collection channel:* The collection channel is the way how a reputation system collects information. This platform will use a collection channel of type 1 *(CC1)*: ratings are left directly in the platform, by providing a way of rating for each element that can be added.
   - *Information Source:* The evaluators that provide the ratings to the system. The information source in this context are the students logged in the platform, and is measured by:
     - Information Source Scale (whether a reputation system has restrictions on the sources): In this context there is a type 2 Information Source Scale *(ISC2)*, which means only registered users can assign ratings. This makes sense, since only students registered in the platform are familiarized enough with the contents to provide ratings.
     - Granularity (how information sources relate with the target):
       This concerns people with very different backgrounds being sources of information in very different domains. Some reputation systems can require a certain reputation score in a certain domain in order for a user to be able to provide ratings. Since there is only a main domain in this platform, the granularity of the information source is of type *GRN1*: There are no restrictions on evaluators. This means every student can rate their colleagues' contributions.
   - *Reputation Information:* The information collected can be classified by:
     - Breadth: Number of properties collected. In this context, only a single property is collected: the ratings assigned.
     - Format: The format of the reputation information. In this reputation system, the format should be of type *IPF1*: A rating scale, that provides a range of possible scores to assign to elements in the platform.
   - *Collection Costs:* The time cost of collecting and processing a single rating. Since the reputation system is small and not a very relevant part of the whole platform, the time costs for collecting and processing ratings are not relevant.

2. **Processing:** Concerning the Processing component, a set of criteria is defined for its evaluation [33], from which the following applies to the platform:

   - *P1(Target rating algorithm)*: The algorithm used to aggregate the ratings assigned to students. There are many algorithms that can be used, from the most simple (summation, average, percentage) to the most complex (Bayesian system and fuzzy models). Given the simplicity of the reputation system to develop, a weighted average algorithm is the algorithm that fits best for the context. Assigning weights when calculating the average is important because, as said above, users with lower reputation scores have a probability of also becoming malicious users in the platform, by, for example, giving low ratings to other users in order to decrease their reputation scores.
   - *P4 (Update Frequency)*: how often the system updates the reputation information. Again, give the simplicity and the small amount of users, information should be updated as soon as the ratings are assigned.

- *P6 (Algorithm complexity)*: refers to the complexity of each algorithm. The weighted average algorithm is a very simple algorithm, with low complexity.
- *P7 (System complexity)*: refers to the complexity of the whole system. The reputation system to develop is, as mentioned before, very simple and therefore has very low complexity.

The remaining criteria did not apply to the reputation system to develop due to its simplicity. For example, criteria P3 (Feedback aggregation algorithms) identifies how feedback ratings are aggregated and is concerned with the information collected from the Feedback Loop (reviews of the reviews). Since this Feedback Loop does not apply to the reputation system to develop, neither does P3.

3. **Output:** Concerning the Output component, there is also a set of defined criteria [33] for both the reporting and dissemination of reputation information For dissemination, the following criteria apply:

   - **O1** (Set of end users): this refers to the users that will be able to see the reputation score assigned to the users of the system. In this case, the set of end users is U_r (the users registered in the platform).
   - **O2** (Access methods): this criteria focuses on whether reputation systems provide alternative ways for users to get the reputation information other than the website. In this platform there is no need for an alternate dissemination of information, since reputation is a minor component of the whole platform. Therefore, no other access methods should be provided.

   Concerning information reporting, two kinds of information are defined [33]:

   - Aggregated (the results of the Processing component)
   - Individual (all individual ratings)

   While individual information is necessary for some kinds of applications (for example, Amazon shows the information collected in every single review, which is important because users are mostly likely to describe the reason for the rating score attributed), it does not add anything relevant to the context of the reputation system to develop, since only a single rating value is collected. Therefore, only the criteria for aggregated information applies:

   - **O4** (Descriptive dimensions): this criteria identifies how many dimensions are used to illustrate the aggregated information. Only a single dimension is used, the weighted average of the ratings, calculated by the Processing component of the reputation system.

   O3 (Timeliness) concerns how the reputation system shows the evolution of reputation information, we do not consider relevant to the system to develop.

4.5 Collaborative Tagging

Collaborative tagging is the practice of allowing anyone to freely attach keywords or tags to content [34].

While in some document repositories or digital libraries there is an authority responsible for assigning and organizing documents by keywords, in collaborative tagging anyone is able to freely attach keywords or tags to the contents. This is very useful when there is no authority for organizing content or simply there is too much content for a single authority to organize: It is the case of the web [34].

Tagging-based systems contrast with taxonomies: while taxonomies organize contents into unambiguous categories that are within more general categories, tagging is neither exclusive nor hierarchical, and therefore it alows to identify content as being about a great variety of things simultaneously [34].

However, both these methods of organizing information have problems with the word semantics, namely polysemous words (a single word with many, related, senses), synonyms (multiple words with the same/closely related sense), and the basic level variation: different people consider different levels of specificity for describing the same entity for example, a person may describe a dog as 'dog' or as 'beagle') [35].

The tags added to content may describe the content itself, or describe the category in which the content falls [36]. It is possible to identify several functions for the tags, within the system that uses the tagging system [34].

In this platform we are aiming for using tagging as a way of identifying the main parts of a software system within a software description, namely stakeholders, scenarios and their parts, tactics, etc.

For that, and to avoid problems of word semantics, tags added to the text of the description must belong to a closed taxonomy of Software Architecture terms, which will contain all terms necessary to describe an Architecture, organized hierarchically.

It will be possible to elicit relationships between tags in the text (for example, two tags X and Y added to two different parts of the text might have some kind of relationship like is-tactic-for-scenario(X,Y)), which will then facilitate the construction of semi-structured contents.

4.6 Semi-Structured Content

Knowledge can be obtained from many different resources, ranging from unstructured (for example, language models obtained from plain text) to structured ones (for example, ontologies) [37].

Unstructured resources are as simple as collections of text, images and other media contents. Thanks to the Web, it is possible to collect huge amounts of unstructured information such as raw text, which allowed major advances in the Natural Language Processing field. However, unstructured knowledge has its limitations: the resources do not provide all the knowledge necessary for complex inference chains [38] and the information is not ontologized not included within a semantic network of unambiguously defined concepts and their semantic relations [37].

Structured resources are machine-readable, and there are several kinds, such as Thesauri (collections of related terms), Taxonomies (hierarchically structured classification of terms) and Ontologies (knowledge model that includes concepts, relations between them and even axioms and rules). These kinds of resources provide information of the highest quality, since their contents were developed with the contribution of experts. However, they require a huge amount of effort in creating and maintaining them, which is very difficult and time-consuming. Also not all topics are covered by the experts, and information might even be slightly culturally-biased. And since these resources are manually annotated, it is hard to keep updating them and they might not contain the lexicalizations of the concepts in different languages [37].

Semi-structured contents try to create a middle-ground between these two types. The most important example of their use is the Wikipedia.

Wikipedia consists of a repository of webpages, each of them containing an entry about a certain concept. There are relations between the different kinds of pages, for example, the redirections (several concepts that redirect for the same article), internal hyperlinks (links for articles about concepts that were included in the text), interlanguage links (links for the same article written in different languages) and category pages (used to classify entries). Pages can also contain infoboxes (tables summarizing

the most important information).

Wikipedia then relies on large amounts of mannually-input knowledge, provided via massive online collaboration. Information is ontologized with high quality thanks to the collaborative editing of articles and it's possible to keep information continuously updated and achieve a wide coverage for almost all domains.

To sum up, using semi-structured resources provides the best of both worlds: high quality information with wide coverage of almost all domains [37].

The main goal of this platform is to allow students to extract a semi-structured representation of a software system from a description in plain-text - have all the parts that constitute a software system (Stakeholders, Scenarios, Tactics, Views, etc) described in a semi-structure (similar to a Wikipedia page), instead of scattered along a ten or more pages article.

The idea is to create templates for all the possible entities that can be added to the semi-structured representation: a template for describing stakeholders, another for describing scenarios, views, styles and so long. Each template should, then, allow a correct description of a component. For example, a scenario, which captures and expresses quality requirements, is defined by a stimulus (a condition arriving at the system), a source of stimulus (entity that generates it), an artifact (the part of the system being stimulated, an environment (the conditions of the system), the response (activity after the stimulus) and a response measure (some way of measuring the response). Whilst the stimulus and the response are mandatory in a scenario, it may not contain all the parts described.

When adding a component to the semi-structure, students will follow the template and fill it with their own words (for example, for describing each part of a scenario). However, all this information should be somehow linked with the parts of the text that describe the scenario.

By using terms from a taxonomy to tag parts of the text and allowing elicitation of relationships between terms (described below) and linking from the semi-structure to a tag in the text, this problem is solved and people are able to create a bridge between the text in the description and the semi-structured content, even if the information for a single component is scattered along the text.

In the field of Artificial Intelligence and Natural Language Processing, there are methods for extraction of machine-readable information from semi-structured contents, namely thesauri and relationships extraction, and for enrichment of structured information with semi-structured content (that also extracts relationships for taxonomy and ontology induction) [37]. In this platform there is no need for Artificial Intelligence techniques, but the concept of relationship extraction is very important. As said before, students are allowed to tag parts of the text from the description with terms from a taxonomy of Software Architecture concepts - namely tag a part of the text as a scenario, a stimulus, a response measure, etc. But this tagged text may be related: a part of text tagged as 'stimulus' may be a stimulus for another part marked as 'scenario' (there is a relationship of the kind is-stimulus-of-scenario). Providing method for elicitation of these relationships allows for a more deep understanding of the description text, by linking the components added to the semi-structure with the tags and relationships elicited from the text.

And since this is a collaborative platform, it is expected that having many people working together in semi-structuring a software description will increase the quality of the final product. As students discuss and correct each others' work, the semi-structure extraction becomes more complete, and students also have an opportunity to learn from their mistakes - since sometimes it is not easy to relate the concepts learned in a classroom with real-life applications.

4.7 Ontology Learning

Research on Ontology learning from text has evolved over the years and there are several open challenges for this field [39]. Ontologies are defined as “effectively formal and explicit specifications in the form of concepts and relations of shared conceptualizations” [40], and can be thought of as a directed graph, with concepts as nodes and relations as edges.

Techniques for ontology learning can be classified in statistics-based, linguistics-based, logic-based or hybrid. Over the years there was also an increased interest in techniques for ontology learning from social data. Existant literature investigates the problem of ontology learning from user-defined tags [41], discusses the requirements for automatic ontology learning from social data [42], presents a tripartite ontology model [43], and describes an approach to complement corpus-based ontology learning with tags [44].

Regarding the context of the platform, all the concepts used to tag the document contents (the concepts of Software Architectures) will be defined a priori and organized according to the relations between them (IS-A relations and others).

Software descriptions analyzed in the platform do not contain these concepts. However, the students' motivation is to find parts of the text that correspond to the concepts predefined and tag them accordingly (for example, a single sentence of the text can identify a stakeholder, or a whole paragraph can identify the stimulus of a scenario).

Since the relations between concepts are already defined, it should be possible for students to elicit these relations between the tagged parts of the text. This facilitates the comprehension of the text contents and the creation of the semi-structure as mentioned previously.

Chapter 5

Solution

To solve the problems elicited throughout this document, it was developed a Web Application. This kind of application facilitates collaboration between its users, and therefore was the more adequate choice for the system to develop. The developed application features:

- An authentication system, where users can login into the system. This authentication system provides an unique identity for a user inside the application, and also provides distinction between types of users, as there are users of type STUDENT and users of type TEACHER, with different permissions.
- Document Management, only available for teachers, where software description articles can be added or removed;
- Document parsing into a view;
- Creation of annotations in the document text. As the name says, an annotation is a part of the text that is marked and highlighted. The AnnotatorJS [1] library provides tools not only to mark up parts of text, but also to associate tags and user text to that selected text.
- Templates for a set of Software Architectures concepts. The parts of marked text described before can be associated to these templates, thus making it easier to co-relate the theoretical concepts with the practical applications. The concepts used in these templates are represented as entities of the domain model.

The next chapters will give an overview of the domain, and explain the architecture and implementation of the system.

[1] http://annotatorjs.org/

Chapter 6

Domain Model

To understand the architectural and implementation decisions taken during the solution development, it is necessary to understand the Software Architectures Domain Model. Section 6.1 lists and describes the concepts talked in the Software Architectures classes, Section 8.1 presents an abstract domain model where it is possible to see how these concepts relate and Section 6.3 gives an idea on how the information from the domain model can be represented in templates.

6.1 Concepts

Before describing the domain model of the developed solution, it is necessary to provide a small introduction to the most important concepts from the Software Architectures course.

6.1.1 Scenarios

A scenario is used to capture and express the quality requirements of a system. The considered qualities are:

- Availability - concerning the uptime of a system;
- Interoperability - how easily can the system interoperate with other system;
- Modifiability - the cost of changing the system;
- Performance - how fast the system responds to events;
- Security - resistance to unauthorized usage whilst providing service to legitimate users;
- Testability - how easy it is to check system faults through testing;
- Usability - how easy it is for a user to accomplish a task, and the kind of support offered by the system;

A scenario consists of six parts [45]:

- **Source of Stimulus:** Some entity (a human, a computer or any other actuator) that generates the stimulus;
- **Stimulus:** A condition that arrives at the system;
- **Environment:** The system condition when the stimulus occurs;

- **Artifact:** Part of the system that was stimulated;
- **Response:** Activity undertaken after the arrival of the stimulus;
- **Response Measure:** when the response occurs, it should be measurable in some way, so the requirement can be tested;

Each Scenario uses a set of Tactics, which are design decisions used to achieve the quality requirements expressed in them. Each quality requirement has a set of commonly used tactics. For example, to assure the Security of a system, tactics such as detecting service denial or message delays (to detect attacks), or revoking access to the system (to react to an attack) are used.

6.1.2 Views

A view is a representation of a set of system elements and the relationships associated with them [46]. This set of elements and relationships is constrained by viewtypes.

A Viewtype defines the element types and relationship types used to describe the architecture of a software system from a particular perspective. Viewtypes refine into styles.

An architectural style is a specialization of element and relation types, together with a set of constraints on how they can be used.

Views can fall into three viewtype categories:

- **Module Viewtype:** document the system principal units of implementation.

  The elements of this viewtype are the *Modules*, which is are implementation units.

  Relationships between modules can be of type *"Is-part-of"*, which defines a part-whole relationship, *"Depends on"*, which defines dependency relations, and *"Is-a"*, which defines a generalization/specialization relationship.

- **Component & Connector Viewtype:** document the system units of execution.

  The elements are the Components, which are the principal processing units and data stores, and the Connectors, which are pathways of interaction between components.

  The relationships can be of type *"Attachment"*, which associate components to connectors, and *"Interface"* Delegation, which associates component ports to other ports from an "internal architecture"- and similarly for the connector.

- **Allocation Viewtype:** document the relationships between a system's software and its development and execution environments.

  The elements are the *Software Element* (elements from the Module and Component & Connector viewtypes) and the *Environmental Element*.

  The relationships are of type *"Allocated-to"*, which means that a software element is allocated to an environmental element.

6.2 Model

Figure 6.1 shows the concepts and relations described in Section 6.1 in a Domain Model Diagram.

The scenario elements are represented as the model entities 'Source of Stimulus', 'Stimulus', Artifact', 'Environment', 'Response' and 'Response Measure' respectively, and are associated to the Scenario entity. A Scenario can only have at most one instance of each element, hence the "0..1" cardinality.

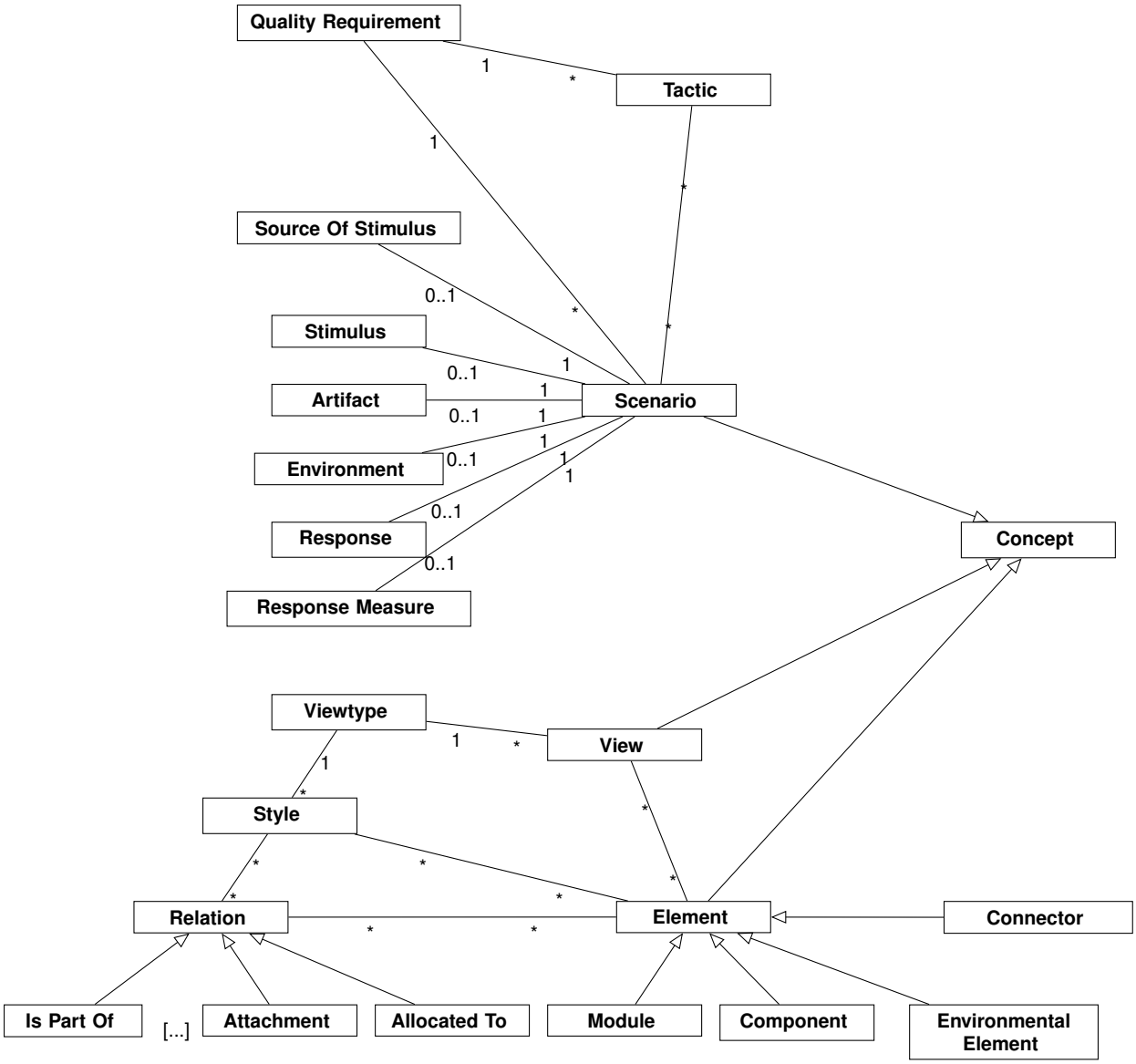

Figure 6.1: Domain Model showing the Software Architectures concepts and how they are related

Similarly, each scenario element can only be associated with a single scenario, and therefore there is a “1” cardinality in the diagram. Each Scenario captures a single Quality Requirement and has a set of Tactics. As mentioned, each quality requirement has a set of commonly used tactics for its achievement.

A View is associated with a Viewtype, which refines into a style. The View contains a set of elements, which are related to each other via specific relations.

The Scenario, View and Element are considered the main concepts of the domain, and they have a dedicated template in the developed solution, as it will be described in the next section. These templates will contain information about the concept and how it is related with other concepts in the domain model.

6.3 Templates

The structured representation of the Software Architectures concepts described in Sections 6.1 and 8.1 is represented in the developed solution by using specific templates for these concepts.

Although we can elicit a wide set of concepts, it does not make sense to have a template for each and

every one of them. It is easier to see the relations between concepts if they are in the same template.

This is the case for the Scenarios. It makes sense to see all the elements of a Scenario together, so it is possible to see, for example, who/what generated the stimulus and what part of the system was stimulated. Therefore, the Scenario is considered one of the main concepts, and it has its own dedicated template, in which are present the quality attribute, the elements and the tactics.

Figure 6.2 shows a Schema for the Scenario template. All the Scenario elements, the quality requirement and the tactics are present in the template, making it possible to see, as mentioned before, how all these concepts are related.

```
<html>
<body>

  <div id="ScenarioIdentification">
    <span>Scenario Name</span>
    <span>Scenario Quality Requirement</span>
  </div>

  <div id="ScenarioDetails">
    <span>Scenario  Description</span>

    <div id="Scenario Elements">
      <div id="Source of Stimulus">Source of Stimulus Description</div>

      <div id="Stimulus">Stimulus Description</div>

      <div id="Artifact">Artifact Description</div>

      <div id="Environment">Environment Description</div>

      <div id="Response">Response Description</div>

      <div id="Response Measure">ResponseMeasure Description</div>

      <div id="Tactics">
        <div>Tactic1  Description</div>
        <div>Tactic2  Description</div>
      </div>
    </div>
  </div>
</body>
</html>
```

Figure 6.2: Schema for the Scenario template

The Views and their elements are also another case of main concepts. It makes sense to aggregate all the system elements and their relationships from a view in a separate template, representing the part of the system being described in that view. However, since a single element can be present in more than one view, it also makes sense to have a specific template for the elements, so their individual properties can be seen.

Figure 6.3 shows a Schema for the Module template, which is the element from the Module Viewtype. Besides the element details, its relationships with other elements (other Modules, in this specific example) are present in the template.

Figure 6.4 shows the schema for the Module Viewtype Views template. Although the template allows inserting a description for the view, its most important feature is the inclusion of other templates in it. As mentioned, a view aggregates a set of system elements and the relationships between them. As there is already a template for the element, the view will include a set of element templates - a template per element, with its respective information.

```
<html>
<body>
  <div id="Module Identification">
    <span>Module Name</span>
  </div>

  <div id="Module Details">
    <div id="Views"> Views Including the Module</div>

    <div id="Description">
      <span>Name</span>
      <span>Responsibilities</span>
      <span>Interfaces</span>
      <span>Implementation Information</span>

      <div id="Relations">
        <div>Is-part-of relations</div>
        <div>uses relations</div>
        <div>Is-a relations</div>
        <div>Crosscuts relations</div>
        <div>One-to-one relations</div>
        <div>One-to-Many relations</div>
        <div>Many-to-many relations</div>
        <div>Aggregation relations</div>
      </div>

    </div>
  </div>
</body>
</html>
```

Figure 6.3: Schema for the Module template

```
<html>
<body>
  <div id="View Identification">
    <div>View Name</div>
    <div>Viewtype</div>
    <div>Style</div>
  </div>

  <div>View Description</div>

  <div id="Elements">
    <div id="Module1Template">...</div>
    <div id="Module2Template">...</div>
    (...)
  </div>
</body>
</html>
```

Figure 6.4: Schema for a Module Viewtype View template

Chapter 7

Architecture Analysis

Chapter 6 introduced the domain model of the Software Architectures concepts present in the developed solution, and the templates in which they are included.

In this chapter, it is shown how these templates are filled with information extracted from a software description article.

The next sections will present two entities from the system: the Document and the Annotation. These entities have a major role in the identification of Software Architectures concepts from the domain model, and the enrichment of the templates for those concepts.

7.1 Document

The Document entity in this system corresponds to an article which describes a software system and is read and analysed in the practical classes of the Software Architectures course. This entity saves the article text and its source, so it can be parsed locally in the application

Figure 7.1 shows the Document entity in the system. The “title” attribute corresponds to the title of the article, the “url” saves the article’s original URL, and the “content” saves the article’s contents.

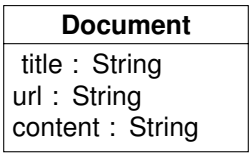

Figure 7.1: The Document Entity

As the main purpose of this application is to create a structured representation of the document using the templates described in Section 6.3, the instances of the main concepts, from which the templates will be generated, must be associated to the Document, as shown in Figure 7.2. This way, generating a structured representation of the document is as easy as extracting the main concepts that are connected to it and including their respecive templates in the structured representation. A concept describes a part of one and only one document, hence the ’1’ cardinality.

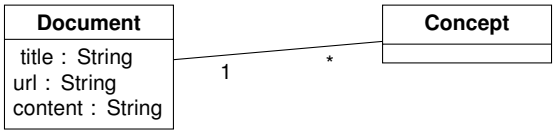

Figure 7.2: Association between Document and Concept entities

7.2 Annotation

An annotation consists of a portion of text selected from the article, enriched with a tag and other information. This selected text can correspond partially or totally to the specification of a Software Architecture concept. For example, the description of the Source of Stimulus of a Scenario may be found in a single paragraph of the article, or in a set of single sentences scattered along the whole article.

Figure 7.3 shows the Annotation entity in the system. The “annotation” field saves a Json representation of the annotation data such as the quote from the text and the given tag. The “tag” field stores the tag given to the annotation. This tag corresponds to a Software Architectures concept, such as “Stimulus” or “Module”.

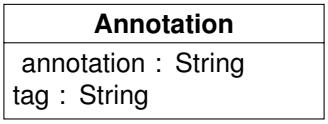

Figure 7.3: The Annotation Entity

Annotations are added to the system by selecting a portion of text in the parsed document and setting it as an annotation. Therefore, an annotation belongs to one, and only one document. In Figure 7.4 it is possible to see the association between the Annotation and Document entities, which reflects the mentioned constraints.

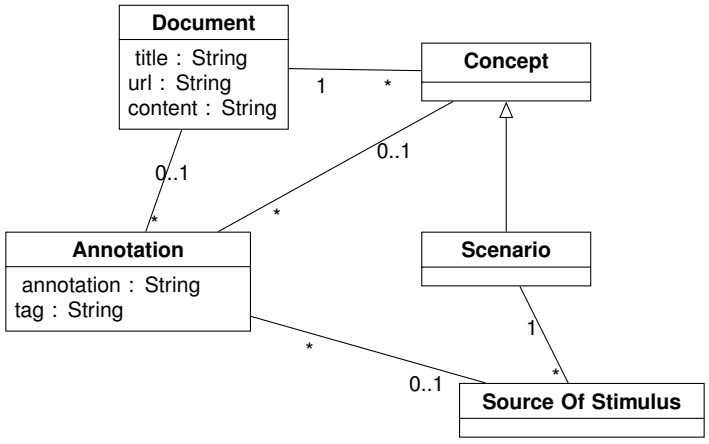

Figure 7.4: Associations between Document, Annotation and the domain model concepts

Annotations are used to enrich the templates described in Section 6.3. Associating the annotations with the respective instances of the concepts allows for that enrichment. All the concepts, not only the main ones, can have associated annotations.

As mentioned in the previous section, an annotation can partially or fully describe a concept, therefore a concept can be associated with multiple annotations.

Figure 7.4 shows how the Annotation is associated with the domain concepts. There is an association not only with the main concepts of the domain model, but also all the other concepts represented in 6.1. The figure exemplifies these associations with only the Scenario and the Source of Stimulus, but all the other concepts have similar associations.

7.3 Interface Flow

The associations between the entities described in the previous sections are created after a set of interactions with the system. These interactions are:

1. After navigation to the document page, a portion of text is selected. The AnnotatorJS interface is prompted, and the user selects a tag to describe the selected text;
2. Upon submission, an instance of “Annotation” is created and associated with the instance of the document;
3. By mouse hovering over the highlighted text, another AnnotatorJS interface is prompted, and the user can associate this annotation to a domain entity by clicking “Add this annotation to the structured representation”;
4. Upon clicking the link, a modal window is shown and, according to the tag given to the annotation, the user can select either an existant Scenario, View or Module to add this annotation to, or create a new instance.

   Initially there are no created instances, and upon creation the instance is associated with the document. When a new Scenario is created, their elements are instantiated as well;
5. After selection, the annotation is associated with the selected element. In the case of Scenarios, if the annotation describes any of the scenario elements, it will be associated with the instance of the corresponding element.

   For example, an annotation tagged as “Stimulus” will be associated with the instance of the Stimulus entity that is associated with the selected Scenario.
6. The user is then redirected to the template of the selected element. Inside the template, the user can delete the annotation, move it to other element, delete the element itself and add text descriptions to the template.

   In the Scenario template the user can also add Tactics, in the Module template the user can add relations with other Modules, and in the View template the user can add elements to the view.

Figures 7.5, 7.6 and 7.7 show the Schemas presented in Section 6.3 of Chapter 6, now including the annotations and the user text.

```
<html>
<body>

  <div id="Scenario Identification">
    <span>Scenario Name</span>
    <span>Scenario Quality Requirement</span>
  </div>

  <div id="Scenario Details">
    <div id="Scenario Description Annotations"></div>
    <div id="Scenario Description Text"></div>

    <div id="Scenario Elements">
      <div id="Source of Stimulus">
        <div id="Source of Stimulus Annotations"></div>
        <div id="Source of Stimulus Description Text"></div>
      </div>

      <div id="Stimulus">
        <div id="Stimulus Annotations"></div>
        <div id="Stimulus Description Text"></div>
      </div>

      <div id="Artifact">
        <div id="Artifact Annotations"></div>
        <div id="Artifact Description Text"></div>
      </div>

      <div id="Environment">
        <div id="Environment Annotations"></div>
        <div id="Environment Description Text"></div>
      </div>

      <div id="Response">
        <div id="Response Annotations"></div>
        <div id="Response Description Text"></div>
      </div>

      <div id="Response Measure">
        <div id="Response Measure Annotations"></div>
        <div id="Response Measure Description Text"></div>
      </div>

      <div id="Tactics">
        <div id="Tactic1">
          <span>Tactic Name</span>
          <div id="Tactic1 Annotations"></div>
          <div id="Tactic1 Description Text"></div>
        </div>

        <div id="Tactic2">
          <span>Tactic Name</span>
          <div id="Tactic2 Annotations"></div>
          <div id="Tactic2 Description Text"></div>
        </div>
        (...)
      </div>
    </div>
  </div>
</body>
</html>
```

Figure 7.5: Scenario Schema enriched with annotations and user text

```
<html>
<body>
  <div id="Module Identification">
    <span>Module Name</span>
  </div>

  <div id="Module Details">
    <div id="Views"> Views Including the Module</div>

    <div id="Description">
      <div id="Name">
        <div id="Module Name Annotations"></div>
      </div>

      <div id="Responsibilities">
        <div id="Module Responsibilities Annotations"></div>
      </div>

      <div id="Interfaces">
        <div id="Module Interfaces Annotations"></div>
      </div>

      <div id="Implementation">
        <div id="Module Implementation Information Annotations"></div>
      </div>

      <div id="Relations">
        <div>Is-part-of relations</div>
        <div>uses relations</div>
        <div>Is-a relations</div>
        <div>Crosscuts relations</div>
        <div>One-to-one relations</div>
        <div>One-to-Many relations</div>
        <div>Many-to-many relations</div>
        <div>Aggregation relations</div>
      </div>

      <div id="Module Description Text"></div>

    </div>
  </div>
</body>
</html>
```

Figure 7.6: Module Schema enriched with annotations and user text

```
<html>
<body>
  <div id="View Identification">
    <div>View Name</div>
    <div>Viewtype</div>
    <div>Style</div>
  </div>

  <div id="View Description">
    <div id="View Annotations"></div>
    <div id="View Description Text"></div>
  </div>

  <div id="Elements">
    <div id="Module1Template">...</div>
    <div id="Module2Template">...</div>
    (...)
  </div>
</body>
</html>
```

Figure 7.7: Module Viewtype view Schema enriched with annotations and user text

Chapter 8

Implementation

The application developed follows a Model-View-Controller architecture.

This architecture is composed of three elements [47]:

- The Model, which captures and stores information about the application domain;
- The View, which generates output based on the model state;
- The Controller, which provides interaction between the Views and the Model. It is able to modify the Model and update the corresponding View.

The standard interaction cycle in this architecture is the following [47, 48]:

- A user interacting with the application sends a request to the Controller.
- The Controller communicates with the Model to apply the desired changes.
- The Model is updated.
- The Controller notifies the corresponding View so it can be updated with the new version of the Model.

The next sections will describe the solution implementation based on these three elements.

8.1 Model

The Model of the implemented system is based on the Domain Model presented in chapter 6, but a few modifications were made to facilitate programming.

The developed application uses the Fénix Framework, which allows the creation of a transactional and persistent domain model for applications [49, 50]. The domain model is specified in the Domain Modeling Language, which is a domain-specific language created for this framework. The framework completely hides the database from the programmer, who can focus on the application development in Java.

8.1.1 Annotations

As seen in Chapter 7, the Document is one of the most important entities, because it aggregates all the annotations created over the text and all the Software Architectures concepts elicited.

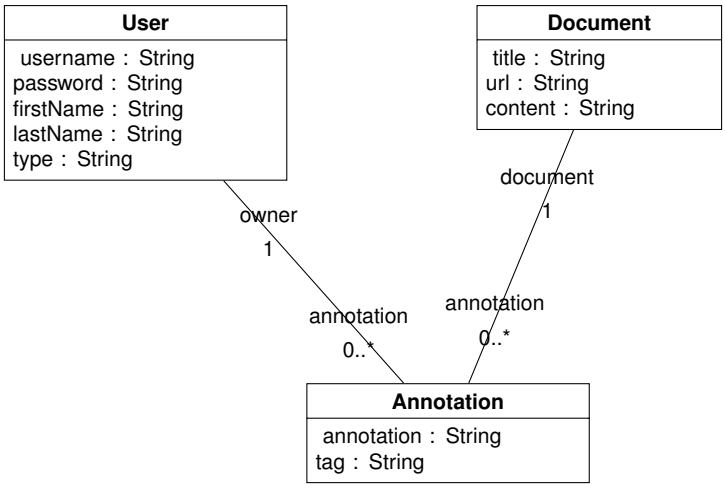

Figure 8.1: Document, Annotations and Users in the implemented Model

Similarly to what is shown in Figure 7.4 of Chapter 7, in the implemented model, the Document entity is connected to the Annotation. However, an annotation is created by an user authenticated in the system, and this information is also present in the domain, as shown in figure 8.1. The “User” entity saves users registered in the system. When a user creates an annotation, it is associated to him.

8.1.2 Scenarios

The Scenario is a main concept of the domain, and therefore is connected to the Document, as seen in Figure 7.2.

Scenarios are represented in the implemented model in a very similar way to what is shown in Figure 6.1. However, there is a difference in the implemented model regarding how Quality Requirements and Tactics are represented.

In Figure 6.1, it is shown an association between Scenario and Quality Requirement, Scenario and Tactic and Tactic and Quality Requirement. The meaning of these associations is to represent what is explained in Section 6.1.1: A Scenario has a Quality Requirement from a set of pre-existent ones, and a set of Tactics also from a set of existent ones, which are specific for the Quality Requirement.

However, in the implemented model, there are no pre-existent Tactics nor Quality Requirements. The existent Quality Requirements and their specific Tactics are specified in a Java class, which returns them as necessary. Figure 8.2 shows the Scenario, Tactic and Quality Requirement entities, and how they are related in the implemented model.

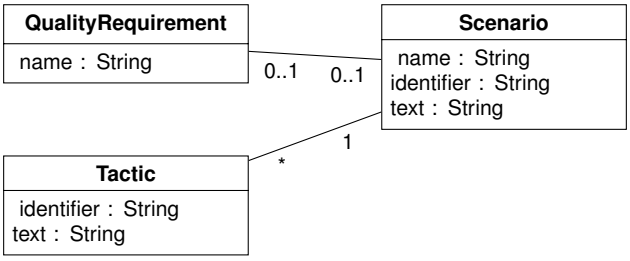

Figure 8.2: Quality Requirements and Tactics detail in the implemented Model

To create a Scenario, it is necessary to select a Quality Requirement from a list, which is retrieved from the class mentioned. When the Scenario is created, a new Quality Requirement is also created, with the selected one stored into the “name” attribute, and associated with that Scenario. Inside the

Scenario template, Tactics can be added. To add a tactic it is necessary to select it from a list of specific tactics for the quality requirement. That list is, again, retrieved from the class mentioned before. When a specific tactic is selected, a new Tactic is created in the model with the specific tactic name stored in the "identifier" attribute. Given these implementation details, there was no need to create the association between Tactic and QualityRequirement nor to have a Quality Requirement associated with many Scenarios.

To facilitate programming, all Scenario elements are a subclass of a "ScenarioElement" class, as seen in the example in Figure 8.3. This superclass aggregates all the methods common to the Scenario elements, making it easier to perform actions such as updating the text, associating annotations or presenting the elements in a view. A few of those methods are shown in the figure. However, this class is not associated with the Scenario entity. Its purpose is only to facilitate programming by aggregating common methods.

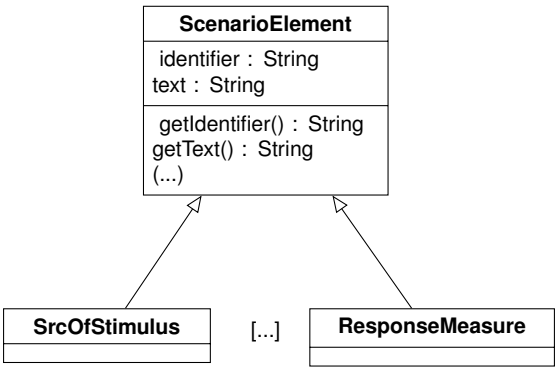

Figure 8.3: The ScenarioElement entity

The "identifier" parameter of the entities, seen in Figures 8.2 and 8.3 stores the name of the concept that corresponds to that entity, and often corresponds to the name of the entity itself. For example, the identifier in all instances of Scenario is "Scenario" and the identifier in all instances of "SrcOfStimulus" is "Source Of Stimulus". This was done to facilitate certain programming issues such as getting the concept name from a "ScenarioElement" instance. The "name" attribute in the Scenario gives it an identification other than just the quality attribute, and the "text" attribute in the Scenario and ScenarioElement corresponds to the description text that can be added to the templates.

All domain entities have associated annotations, as it was mentioned previously in Section 7.2. The implemented model is similar to what is shown in Figure 7.4, with the Scenario and ScenarioElement entities being associated with the Annotation.

8.1.3 Modules

The Modules are the elements of the Module Viewtype Views. Figure 8.4 shows how a Module is represented in the implemented model.

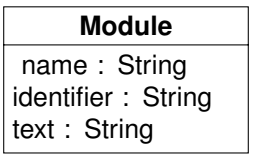

Figure 8.4: The Module entity

The Module entity has a "name" attribute, which provides identification and also gives an idea on

what are the Module functions, an “identifier”, as explained in the previous section, and a “text” attribute, which corresponds to text added to the template.

As it is one of the main concepts of the system, the Module is associated with the corresponding document upon its creation as demonstrated in the example in Figure 7.4. The Module is also associated with the set of Annotations that describe it.

As seen in the abstract domain model present in Figure 6.1 of Chapter 6, the elements of a view have a certain type of relationship with each others. Although the abstract domain model in Figure 6.1 contains an entity “Relation” to represent how the view elements are connected, in the case of the Modules, these relations have no other information than the Modules connected.

Therefore, in the implemented model, there are no entities representing the relationships between modules. Instead, there were defined different associations of the Module entity to itself, in order to represent the different relations.

Each Module Viewtype Style specifies relation types refined from the ones presented in 6.1.2. There are, then, eight types of relations between Modules[46] defined in the Domain Modeling Language:

- *Is-Part-Of* - a Module can be part of one and only one parent Module. Figure 8.5 shows how this relation was implemented. A Module can have only one other Module set as parent, but can have many children Modules;

```
relation moduleIsPartOfModule {
    Module playsRole parent{ multiplicity 0..1; }

    Module playsRole child{ multiplicity 0..*; }
}
```

Figure 8.5: Is-Part-Of relation between Modules in the implemented model

- *Uses* - a specialization of the *depends-on* relation, where a Module depends on the correct functioning of other Modules to satisfy its own requirements. The implementation is shown in Figure 8.6. A Module can use and be used by many other Modules;

```
relation moduleUsesModule {
    Module playsRole uses{ multiplicity 0..*; }

    Module playsRole usedBy{ multiplicity 0..*; }
}
```

Figure 8.6: Uses relation between Modules in the implemented model

- *Is-A* - a relation of generalization, in which a Module is a generalization, or parent, of other modules. Implementation is shown in Figure 8.7;

```
relation moduleIsAModule {
    Module playsRole isA { multiplicity 0..*; }

    Module playsRole superModule { multiplicity 0..*; }
}
```

Figure 8.7: Uses relation between Modules in the implemented model

- *Crosscuts* - an Aspect Module, which implements a crosscutting concern of the system, is bound to a Module that is affected by that crosscutting concern. Implementation is shown in Figure 8.8;

```
relation moduleCrosscutsModule {
    Module playsRole crossCuts { multiplicity 0..*; }

    Module playsRole crossCuttedBy { multiplicity 0..*; }
}
```

Figure 8.8: Uses relation between Modules in the implemented model

- *One-To-One, One-To-Many, Many-to-Many* - logical associations between Data Entity Modules, similar to the UML associations. Implementation of these relations is shown in Figure 8.9.

```
relation moduleOneToOneModule {
    Module playsRole oneToOne { multiplicity 0..*; }
    Module playsRole oneRelation { multiplicity 0..*; }
}

relation moduleOneToManyModule {
    Module playsRole oneToMany { multiplicity 0..*; }
    Module playsRole oneToManyRelations { multiplicity 0..*; }
}

relation moduleManyToManyModule {
    Module playsRole manyToMany { multiplicity 0..*; }
    Module playsRole manyToManyRelations { multiplicity 0..*; }
}
```

Figure 8.9: One-to-One, One-To-Many and Many-To-Many relation between Modules in the implemented model

To note: the '0..*' cardinality in the one-to-one, one-to-many and many-to-many relations means that a Module can have several relations of this type in a Data Model Style view. For example, if a Module instance with name "Department" has a "One-To-Many" relation with Modules "Employee" and "Room", it means that a visual representation of these Modules would be the one in figure 8.10.

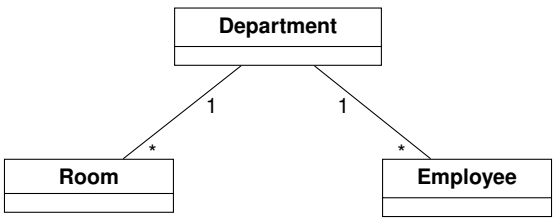

Figure 8.10: Example of the One-To-Many relation between Modules

- *Aggregation* - an aggregation relation. Figure 8.11 shows how this relation is implemented. A Module can be an aggregator of other several Modules, and be aggregated by many others;

8.1.4 Components and Connectors

Components and Connectors are the elements of the Component & Connector Viewtype Views. Figure 8.12 shows how these concepts are represented in the implemented model.

A Component has a set of Ports, and a Connector as a set of Roles. The Ports of a Component are connected to the Roles of a Connector by the *Attachment* relation explained in Section 6.1 of Chapter 6. The implementation of this relation will be explained further on. The "name" and "text" attributes in the entities correspond to the identification name of the entity and the descriptive text that can be added,

```
relation moduleAggregatesModule {
    Module playsRole aggregator { multiplicity 0..*; }

    Module playsRole aggregated { multiplicity 0..*; }
}
```

Figure 8.11: Aggregation relation between Modules in the implemented model

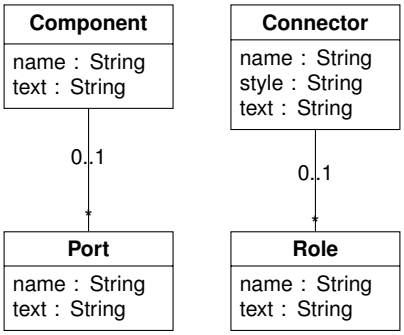

Figure 8.12: Component and Connector entities in the implemented model

respectively. The “style” attribute in the Connector refers to the Component & Connector styles. Each style has a unique type of connector. Specifying the style of a connector will make it easier to figure which Components will attach to it.

The Component and Connector are main concepts of the system, and therefore are associated with the corresponding document upon creation. They and their respective Ports and Roles are also associated to a set of Annotations that describe them. The associations of the Component and Connector to the Document and Annotation are similar to the ones in the example of Figure 7.4.

Components are related with Connectors by the *Attachment* relation. A Component has, as mentioned, a set of ports, and these ports can be attached to roles of Connectors.

Similar to the Module relations, the *Attachment* relation in the implemented model has no more information than the Port and Role involved. Therefore, it did not make sense to create an entity to represent this relation. Instead, a new relation between the Port and Role entity is created in the Domain Modeling language, as shown in Figure 8.13.

```
relation portIsAttachedToRole {
    Port playsRole port { multiplicity 0..1; }

    Role playsRole role { multiplicity 0..1; }
}
```

Figure 8.13: Attachment relation between Component Ports and Connector Roles in the implemented model

8.1.5 Views

The abstract domain model presented in Figure 6.1 of Chapter 6 shows how a View is related to the Viewtype, the Styles and the elements.

A View is, as the Scenario, the Element and the Relation, considered a main concept, and therefore has its own dedicated template. Figure 8.14 shows how the View is represented in the implemented model.

In the abstract domain model of Figure 6.1 the View is associated with the Viewtype entity which, in

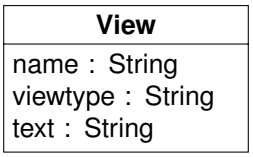

Figure 8.14: View entity in the implemented model

turn, is associated with a Style entity. Similar to what was explained for the Scenarios with the Quality Requirements and Tactics, the abstract domain model assumes there are a set of pre-defined Viewtype entities, each associated with a pre-defined set of Styles.

In the implemented domain model, however, there are no pre-defined Styles nor Viewtypes. When a view is added to the system, a viewtype must be chosen, and it is stored in the "viewtype" attribute. This attribute is used to distinguish what kind of elements can be added to the view. For example, if the "viewtype" attribute of a View is set to "Module Viewtype", then only Modules are added to that View.

The styles of a view are not explicitly stated, but they are implicit by the elements that are added to the View. For example, a Module Viewtype View containing a set of Modules with "uses" relations between each others implies that there is a *Uses* style in the view, and a Component & Connector Viewtype View containing Components attached to Connectors with a certain style implies that that style is present in the view. This verification is done by the Controller, as described in Subsection 8.2.6.

A View is composed by a set of Elements, as seen in Figure 6.1. In the implemented domain model, the View has a similar association with the Module, Component and Connector entities. Figure 8.15 shows how these relations are implemented in the Domain Modeling Language.

```
relation viewHasModules {
    View playsRole view { multiplicity 0..*; }
    Module playsRole module { multiplicity 0..*;}
}

relation viewHasComponents {
    Component playsRole component { multiplicity 0..*; }
    View playsRole view { multiplicity 0..*; }
}

relation viewHasConnectors {
    Connector playsRole connector { multiplicity 0..*; }
    View playsRole view { multiplicity 0..*; }
}
```

Figure 8.15: Relation between View and the view elements: Module, Component and Connector

A View is created by associating an Annotation with it and, as it is a main concept, it is associated with the respective Document, as demonstrated in the example in Figure 7.4.

8.2 Controller

This section describes the Controller part of the implemented system. The application developed uses the Spring[1] Framework, which allows the definition of one or more Java classes to act as Controllers and handle requests.

A Java class is set to be a Controller by annotating it with the *@Controller* annotation, or *@RestController* if the controller's goal is to provide a REST API. Figure 8.16 shows an example from the Document Controller described in 8.2.2, with a method that handles requests for the URL *'/selectDoc/id'*. Method

[1] https://spring.io/

showDocument() retrieves the Document from the domain model with the specified *id* and adds it and the *id* itself as Model attributes, so the template will be able to use them. The template “docTemplate” will be then showed to the user.

```
@Controller
public class DocumentController {
    // ...
    @RequestMapping(value = "/selectDoc/{id}", method = RequestMethod.GET)
    public String showDocument(@PathVariable String id, Model m) throws IOException {
        m.addAttribute("docId", id);
        Document d = FenixFramework.getDomainObject(id);
        m.addAttribute("doc", d);
        return "docTemplate";
    }
}
```

Figure 8.16: Example of a Controller class defined using the Spring Framework

8.2.1 Annotation Controller

The AnnotationController class provides a way to manage the annotations added to a document. It communicates with the Domain Model to create, remove or update Annotations. This controller is in fact a RestController, which means it provides a REST API, used to retrieve a JSON representation of the annotations stored in the model and display it along with the document text. The endpoints provided by this controller are described in Table 8.1.

Name	Request Method	Endpoint	Description
INDEX	GET	/selectDoc/{docId}/ store/annotations	Returns the set of annotations associated with the document with id *docId*
READ	GET	/selectDoc/{docId}/ store/annotations/{id}	Returns the annotation with the specific *id*
CREATE	POST	/selectDoc/{docId}/ store/annotations	Creates a new annotation, stores it in the model associated with the document with id *docId*, and redirects to the Read endpoint
UPDATE	PUT	/selectDoc/{docId}/ store/annotations/{id}	Updates the annotation with the given *id* and redirects to the READ endpoint
DELETE	DELETE	/selectDoc/{docId}/ store/annotations/{id}	Removes the association between the annotation with the given id and the document with id *docId*. The response is a HTTP/1.0 204 NO CONTENT.

Table 8.1: REST API provided by the Annotation Controller

The **INDEX** endpoint has the particularity of iterating through all the Annotations in the system. If the annotation is associated to another domain entity besides the Document (for example, a Scenario, or a Module), information about the domain entity will be added to the annotation body, and will be shown in the interface. This way, it is possible to have some information about the domain entity even before checking its template.

Figure 8.17 shows an example of how information is added to the annotation in the *INDEX* endpoint. The method *getAnnotations()*, which receives the requests for this endpoint, iterates over the annotations associated with the document with id *docId*, parses the JSON of each annotation into an instance of the AnnotationJS class, checks which domain element is associated with the annotation and adds information accordingly to the field "text" of the annotation. The example in the figure shows how information about the viewtype is added to annotations connected with a View. The list of all retrieved AnnotationJS instances is then parsed into a JSON string and sent as response.

```
@RequestMapping(value = "/selectDoc/{docId}/store/annotations",
method = RequestMethod.GET)

public String getAnnotations(@PathVariable String docId) {
    Document d = FenixFramework.getDomainObject(docId);
    List<AnnotationJ> anns = new ArrayList<AnnotationJ>();
    Gson g = new Gson();
    for(Annotation a : d.getAnnotationSet()) {
        AnnotationJ ann = g.fromJson(a.getAnnotation(), AnnotationJ.class);
        if(a.isViewAnnotation() && a.getView() != null) {
            View v = a.getView();
            ann.setText("View: " + v.getViewtype());
        }

        //...

        anns.add(ann);
    }
    String resp = g.toJson(anns);
    return resp;
}
```

Figure 8.17: Example of how the INDEX endpoint adds information to the annotation body

8.2.2 Document Controller

The DocumentController class handles the requests to view, add or remove a document from the system.

Adding and removing documents from the system is a feature that only Teachers are authorized to use. The controller receives a POST request containing the URL to the article. The Java library JSoup[2] extracts the HTML from the given URL and the controller processes it and stores a Document in the database. Processing of the extracted HTML includes turning relative URLs into absolute ones for href and src attributes. The Controller will also check if the database already contains a document with the given URL before adding a new entry.

When a teacher removes a Document from the system, all Annotations and Domain Model entities associated with that Document such as Scenarios or Views are removed from the system as well.

Upon Document visualization, there are three other operations handled by this controller:

- When a user wants to associate an Annotation with a domain entity such as a Scenario, a request is sent to the Document Controller containing information about the annotation unique ID and its tag. Based on the tag, the controller then redirects to another controller, which will provide means for the user to add and/or select an entity to associate with the annotation.

  The reason for this redirect is the existent of a logic division of the Controllers. Each Controller class handles a set of requests related with a certain entity of the domain model. The AnnotationController class handles operations over Annotations, the ScenarioController handles operations over Scenarios and their elements, etc.

[2] http://jsoup.org/

Therefore, it's only logical to redirect this request to the Controller that handles operations over the respective domain entity. Figure 8.18 shows the method in DocumentController that performs the redirect.

```
@RequestMapping(value = "/addAnnotationToStructure/{docId}/{annotationId}/{tag}")
public RedirectView addAnnotationModal(@PathVariable String docId,
        @PathVariable String annotationId, @PathVariable String tag) {
    RedirectView rv = null;
    if (Utils.allScenarioConcepts().contains(tag)) {
        rv= new RedirectView("/addAnnotationToScenarioStructure/" + docId + "/"
                + annotationId);
    } else if (tag.contains("Module")) {
        rv = new RedirectView("/addAnnotationToModuleTemplate/" + docId + "/"
                + annotationId);
    }else if (tag.contains("View")) {
        rv = new RedirectView("/addAnnotationToViewTemplate/" + docId + "/"
                + annotationId);
    }else if (tag.contains("Component")) {
        rv = new RedirectView("/addAnnotationToComponentTemplate/" + docId + "/"
                + annotationId);
    }else if (tag.contains("Connector")) {
        rv = new RedirectView("/addAnnotationToConnectorTemplate/" + docId + "/"
                + annotationId);
    }
    return rv;
}
```

Figure 8.18: Redirecting requests to associate annotations with domain entities in the DocumentController

- When visualizing the Annotation information in the document, it is possible to navigate to the template of the entity to which this annotation is associated by clicking in a link. A request is then sent to the Document Controller, which again checks the Annotation tag, and redirects to the correct Controller. Figure 8.19 shows the DocumentController class method which performs the redirect according to the annotation tag.

```
@RequestMapping("/viewTemplate/{docId}/{connectedId}/{annotationId}")
public RedirectView viewTemplate(@PathVariable String docId,
        @PathVariable String connectedId, @PathVariable String annotationId) {
    Annotation a = FenixFramework.getDomainObject(annotationId);
    RedirectView rv = new RedirectView();
    if(a.isScenarioAnnotation()) {
        rv.setUrl("/viewScenario/"+docId+"/"+connectedId+"#"+annotationId);
    }else if( a.getTag().contains("Module")) {
        rv.setUrl("/viewModule/"+docId+"/"+connectedId+"#"+annotationId);
    }else if(a.getTag().contains("View")) {
        rv.setUrl("/viewView/"+docId+"/"+connectedId+"#"+annotationId);
    }else if(a.getTag().contains("Component")) {
        rv.setUrl("/viewComponent/"+docId+"/"+connectedId+"#"+annotationId);
    }else if(a.getTag().contains("Connector")) {
        rv.setUrl("/viewConnector/"+docId+"/"+connectedId+"#"+annotationId);
    }
    return rv;
}
```

Figure 8.19: Redirecting a request to visualize an entity template in the DocumentController

- The Document interface also allows the user to navigate to the structured representation of the document, which is an aggregation of all the templates of all the entities added to the system. When the link to navigate to the structured representation is clicked, a request is sent to the Controller, which retrieves information about Scenarios, Views and Elements to be presented to the user.

Figure 8.20 shows the method *viewStructuredRepresentation()*, which handles the request to see the Structured Representation of the Document. This method retrieves all the main elements from the database (scenarios, views, modules, components and connectors) and adds them as model attributes, so they can be used by the template. The template “structuredRepresentation” is then returned and showed to the user.

```
@RequestMapping("/viewStructuredRepresentation/{docId}")
public String viewStructuredRepresentation(@PathVariable String docId,
        Model m) {
    Document d = FenixFramework.getDomainObject(docId);
    m.addAttribute("scenarios", d.getScenarioSet());
    m.addAttribute("views", d.getViewSet());
    m.addAttribute("modules", d.getModuleSet());
    m.addAttribute("components", d.getComponentSet());
    m.addAttribute("connectors", d.getConnectorSet());
    m.addAttribute("docId", docId);
    m.addAttribute("title", d.getTitle());
    return "structuredRepresentation";
}
```

Figure 8.20: Retrieving the Structured Representation template in the DocumentController

8.2.3 Scenario Controller

The Scenario Controller provides means to add and remove Scenarios from the document and link or unlink annotations from a Scenario or its elements.A Scenario is created when a user wants to link an Annotation to a Scenario and is prompted the interface to add a new or choose an existing one. When the Controller receives the request to add a new Scenario, it does not only adds a new Scenario to the database, but also adds a new SrcOfStimulus, Stimulus, Artifact, Environment, Response and ResponseMeasure to the databse, all associated with the newly created Scenario.

```
@RequestMapping(value = "/addNewScenario/{docId}/{annotationId}/{scenarioName}/
{qualityRequirement}")
public RedirectView addNewScenario(@PathVariable String docId,
@PathVariable String annotationId, @PathVariable String scenarioName,
        @PathVariable String qualityRequirement,@RequestParam String move) {
    Document d = FenixFramework.getDomainObject(docId);
    addScenarioToDocument(d,qualityRequirement ,scenarioName);
    //...
}
    @Atomic(mode = TxMode.WRITE)
private void addScenarioToDocument(Document d, String qualityRequirement, String scenarioName) {
    Scenario s = new Scenario();
    s.setName(scenarioName);
    s.setIdentifier("Scenario");
    QualityRequirement qr = new QualityRequirement();
    qr.setName(qualityRequirement);
    s.setQualityRequirement(qr);
    SrcOfStimulus src = new SrcOfStimulus();
    src.setIdentifier("Source Of Stimulus");
    //...
    ResponseMeasure rm = new ResponseMeasure();
    rm.setIdentifier("Response Measure");
    s.setSrcOfStimulus(src);
    //...
    s.setResponseMeasure(rm);
    d.addScenario(s);
}
```

Figure 8.21: Creating a new Scenario in ScenarioController

Figure 8.21 shows the method *addNewScenario*, which handles the request for adding a new Sce-

nario to the Document and calls the method *addScenarioToDocument()*, which creates the new Scenario with the given name, creates and associates with it the selected Quality Requirement and the Scenario elements and associates it with the respective document.

When the user chooses which Scenario to link the Annotation with, the Controller receives a request containing both the unique ID of the Scenario and the unique ID of the Annotation. With these IDs, it can verify the tag associated with the Annotation, and add the Annotation either to the Scenario or to the corresponding element.

For example, if the tag associated with the Annotation is “Source Of Stimulus”, then it will be linked to the “SrcOfStimulus” instance that is associated with the Scenario with the specified ID.

Figure 8.22 shows the method *addAnnotationToScenario()*, which associates an Annotation with a Scenario or one of its elements accordingly.

```
@Atomic(mode = TxMode.WRITE)
private void addAnnotationToScenario(Scenario s, Annotation a) {
    String tag = a.getTag();
    if (tag.equals("Scenario Description") || tag.equals("Tactic")) {
        updateAnnotation(s.getExternalId(), a);
        s.addAnnotation(a);
    }
    if (s.getElements().get(tag) != null) {
        ScenarioElement em = s.getElements().get(tag);
        updateAnnotation(s.getExternalId(), a);
        em.addAnnotation(a);
    }
}
```

Figure 8.22: Associating an Annotation with a Scenario in ScenarioController

A Scenario is initially created without any tactics. These are added in the Scenario Template, and their creation is handled by the Scenario Controller. Upon receiving the corresponding request, a new “Tactic” is added to the database containing the tactic’s name, and is associated with the corresponding Scenario. When an Annotation with the tag “Tactic” is added to a Scenario, it is initially associated with the “Scenario” entity, as it can be seen in Figure 8.22.

Inside the template, these annotations can be associated with the added tactics. Figure 8.23 shows how a request to associate an Annotation with a specific Tactic is handled. To note, in the *linkAnnotationToTactic()* method, the association between the Scenario and the Annotation is removed to create the new association with the Tactic.

```
@RequestMapping(value = "/linkToTactic/{docId}/{scenarioId}/{tacticId}/{annotationId}")
public RedirectView addAnnotationToTactic(@PathVariable String docId,
        @PathVariable String scenarioId, @PathVariable String tacticId,
        @PathVariable String annotationId) {
    //...
    linkAnnotationToTactic(s, t, a);
    //...
}

@Atomic(mode = TxMode.WRITE)
private void linkAnnotationToTactic(Scenario s, Tactic t, Annotation a) {
    s.removeAnnotation(a);
    t.addAnnotation(a);
}
```

Figure 8.23: Associating an Annotation with a specific Tactic in ScenarioController

When user text is added to a Scenario or one of its elements, a request is sent to this Controller. It will then update the Model accordingly and show the updated Model in the corresponding View.

Figure 8.24 shows how the Scenario *text* field is updated. The methods for updating the Scenario elements (including Tactics) are very similar to the example in the Figure.

```
@RequestMapping(value = "/setScenarioText/{docId}/{scenarioId}", method = RequestMethod.POST)
public RedirectView setScenarioText(@RequestParam String text,
        @PathVariable String docId, @PathVariable String scenarioId) {
    Scenario scen = FenixFramework.getDomainObject(scenarioId);
    updateText(scen, text);
    RedirectView rv = new RedirectView("/viewScenario/" + docId + "/"
            + scenarioId + "#" + scen.getName() );
    return rv;
}

@Atomic(mode=TxMode.WRITE)
private void updateText(Scenario scen, String text) {
    scen.setText(text);
}
```

Figure 8.24: Updating the Scenario text in the ScenarioController

8.2.4 Module Controller

Similarly to the Scenario Controller, the Module Controller handles the creation and deletion of Modules, which are the elements of Module Viewtype Views. It also handles the linkage of Annotations to existent Modules and the addition of user text to a Module.

The implementation of the methods that handle these operations is similar to the implementation done in the Scenario Controller. But unlike the Scenario, the Module has no other elements, therefore creating a Module only creates a Module in the database, and annotations are always linked to that Module.

As explained in Chapter 6, the Elements of a View are related to each others, and in the Subsection 8.1.3, we can see that, since the relations between Modules have no information other than the Modules involved, they are represented in the model as different associations of the “Module” entity to itself.

The Module Controller handles adding and removing other Modules to/from the possible relations. The requests are unique for each relation, meaning the request to add a Module to the *Uses* relation of a certain Module will be in the form */setModuleUses/...* and the request to add *One-To-One* relations with other Modules to a certain Module will be in the form */setModuleOneToOne/...*. Each request modifies the corresponding association accordingly.

```
@RequestMapping(value = "/setModuleParent/{docId}/{moduleId}/{parentId}")
public RedirectView addModuleParent(@PathVariable String moduleId,
        @PathVariable String parentId, @PathVariable String docId) {
    Module mod = FenixFramework.getDomainObject(moduleId);
    Module parent = FenixFramework.getDomainObject(parentId);
    addParent(mod, parent);
    RedirectView rv = new RedirectView("/viewModule/" + docId + "/"
            + moduleId+"#isPartOf");
    return rv;
}
@Atomic(mode = TxMode.WRITE)
private void addParent(Module mod, Module parent) {
    mod.setParent(parent);
}
```

Figure 8.25: Example of adding a “Is-Part-Of” relation between two modules

Figure 8.25 shows how the ModuleController class adds a “Is-Part-Of” relation between two modules. Inside a Module template it is possible to select another Module to be the parent and associate them with

a “Is-Part-Of” relation. The *addModuleParent()* method handles the request to create this association. The request contains the *id* of the module in which the relation will be created, and the *id* of the Module chosen to be the parent. After setting the relation, the user is redirected to the Module template.

8.2.5 Component and Connector Controllers

Components and Connectors are the elements of the Component & Connector Viewtype Views. The ComponentController and ConnectorController classes handle the creation and deletion of Components and Connectors from the system. The two classes have very similar methods.

A Component has a set of Ports, which can be attached to Connector Roles. Adding and removing Ports to a component is very similar to the methods implemented to adding and removing tactics from a Scenario. Initially a Component has no Ports, and these can be created inside the Component template, as shown in Figure 8.26.

```
@RequestMapping(value = "/addPortToComponent/{docId}/{componentId}/{portName}")
public RedirectView addPortToComponent(@PathVariable String docId,
        @PathVariable String componentId, @PathVariable String portName) {
    Component comp = FenixFramework.getDomainObject(componentId);
    addPortToComponent(comp, portName);
    RedirectView rv = new RedirectView("/viewComponent/" + docId + "/"
            + componentId + "#" + comp.getName());
    return rv;
}

@Atomic(mode = TxMode.WRITE)
private void addPortToComponent(Component comp, String portName) {
    Port p = new Port();
    p.setName(portName);
    comp.addPort(p);
}
```

Figure 8.26: Adding Ports to a Component in ComponentController

Annotations with the tag “Component Port” are initially associated with the Module, but can be associated with a specific port, as seen in Figure 8.27.

```
@RequestMapping(value = "/moveAnnotationToPort/{docId}/{componentId}/{portId}/{annotationId}")
public RedirectView moveAnnotationToPort(@PathVariable String docId,
        @PathVariable String componentId, @PathVariable String portId,
        @PathVariable String annotationId) {
    Component comp = FenixFramework.getDomainObject(componentId);
    Port p = FenixFramework.getDomainObject(portId);
    Annotation a = FenixFramework.getDomainObject(annotationId);
    moveAnnotationToPort(comp, p, a);
    RedirectView rv = new RedirectView("/viewComponent/" + docId + "/"
            + componentId + "#" + comp.getName());
    return rv;
}

@Atomic(mode = TxMode.WRITE)
private void moveAnnotationToPort(Component comp, Port p, Annotation a) {
    comp.removeAnnotation(a);
    p.addAnnotation(a);

}
```

Figure 8.27: Associating an Annotation with a Port in the ComponentController

The Connector has a set of Roles, which are attached to the Component Ports. The implementation of the Roles in a Connector is similar to the implementation of the Ports: Initially a Connector has no

Roles, which can be added inside the template, and annotations can be associated with the created Roles. The methods for creating Roles and linking annotations to them in the ConnectorController class are very similar to the ones represented in the Figures 8.26 and 8.27, but the requests are sent to */addRoleToConnector* and */moveAnnotationToRole/* and instead of Component and Port instances there are Connectors and Roles.

As mentioned in Section 8.1.4, component ports can be associated with the connector roles by a relation of *attachment*. Inside a Component template, it is possible to attach each of the ports to a Role from an existent Connector.

Figure 8.28 shows how an attachment relation is created between a Port and a Role.

```
@RequestMapping(value = "/attachPortToConnectorRole/{docId}/{portId}/{roleId}")
public RedirectView attachPortToRole(@PathVariable String docId,
        @PathVariable String portId, @PathVariable String roleId) {
    Port p = FenixFramework.getDomainObject(portId);
    Role r = FenixFramework.getDomainObject(roleId);
    attachPortToRole(p,r);
    RedirectView rv = new RedirectView("/viewComponent/" + docId + "/"
            + p.getComponent().getExternalId() + "#" + p.getName());
    return rv;
}

@Atomic(mode=TxMode.WRITE)
private void attachPortToRole(Port p, Role r) {
    p.setRole(r);
}
```

Figure 8.28: Attachment of Component Ports to Connector Roles in the ComponentController

8.2.6 View Controller

The ViewController class handles the creation and deletion of Views from two possible Viewtypes: Module and Component & Connector.

Similar to the other domain entities, a new View is created when the user wants to associate an annotation with the tag “View” with a domain entity. Figure 8.29 shows how a request to add a new Module Viewtype View is handled.

```
@RequestMapping(value = "/addNewView/{docId}/{annotationId}/Module Viewtype/{viewName}")
public RedirectView addNewModuleViewTypeView(@PathVariable String docId,
        @PathVariable String annotationId, @PathVariable String viewName,
        @RequestParam String move) {
    Document d = FenixFramework.getDomainObject(docId);
    addModuleViewtypeToDocument(d, viewName, "Module Viewtype");
    RedirectView rv = new RedirectView();
    if (move.equals("yes")) {
        rv.setUrl("/moveAnnotationView/" + docId + "/" + annotationId);
    } else {
        rv.setUrl("/addAnnotationToViewTemplate/" + docId + "/"
                + annotationId);
    }
    return rv;
}
```

Figure 8.29: Adding a new View in the ViewController

As there are two possible viewtypes for the views added to the application, the ViewController distinguishes between two different templates when receiving a request to view the View’s template: One for views from the Module Viewtype, to which only Modules can be added, and one for views from the

Component & Connector Viewtype, to which only Components and Connectors can be added. This distinction is done inside the method *viewViewTemplate()* of the ViewController class.

Figure 8.30 shows how the method checks the *viewtype* attribute of a View object to add the correct attributes to the template model and return the correct template. The controller adds the existent Modules, Component and Connectors as template model attributes so the template can show them to the user who wants to add elements to a view. The set of existent Views is also added as attribute, to show in case the user wishes to move an annotation to other view. The UsedIds instance added as attribute is a class containing an empty *List<String>*, which will be used to save the *ids* of the elements selected to add to a view.

```
@RequestMapping(value = "/viewView/{docId}/{viewId}")
public String viewViewTemplate(Model m, @PathVariable String docId,
        @PathVariable String viewId) {
    m.addAttribute("docId", docId);
    View v = FenixFramework.getDomainObject(viewId);
    Document d = FenixFramework.getDomainObject(docId);
    if(v.getViewtype().equals("Module Viewtype")) {
        m.addAttribute("view", v);
        m.addAttribute("views", d.getViewSet());
        m.addAttribute("modules", d.getModuleSet());
        m.addAttribute("used", new UsedIds());
        return "viewMVTTemplate";
    }else if(v.getViewtype().equals("Component & Connector Viewtype")) {
        m.addAttribute("view", v);
        m.addAttribute("views", d.getViewSet());
        m.addAttribute("components", d.getComponentSet());
        m.addAttribute("connectors", d.getConnectorSet());
        m.addAttribute("used", new UsedIds());
        return "viewCCTemplate";
    }
    return null;
}
```

Figure 8.30: Request to see a View's Template in the ViewController

After the user selects a set of Modules, Components or Connectors to add to a view, a request is sent to the ViewController, which handles the association of these elements with the respective view.

Figure 8.31 shows how Modules are associated with a View, but these methods are similar for the Components and Connectors. A POST request is sent to */addModulesToView/{docId}/{viewId}*, con-

```
@RequestMapping(value = "/addModulesToView/{docId}/{viewId}", method = RequestMethod.POST)
public RedirectView addModulesToView(@PathVariable String docId,
        @PathVariable String viewId, @ModelAttribute UsedIds modules) {
    View v = FenixFramework.getDomainObject(viewId);
    if(v.getViewtype().equals("Module Viewtype")) {
        addModulesToView(v, modules);
    }
    RedirectView rv = new RedirectView("/viewView/" + docId + "/" + viewId);
    return rv;
}
@Atomic(mode = TxMode.WRITE)
private void addModulesToView(View v, UsedIds modules) {
    for (String id : modules.getUsed()) {
        Module m = FenixFramework.getDomainObject(id);
        v.addModule(m);
    }
}
```

Figure 8.31: How Modules are added to a View in ViewController

taining in its body the instance of the *UsedIds* class that was added as model attribute (see Figure 8.30), with the *ids* of the selected elements saved in the list. Method *addModulesToView()* makes sure the

view has the correct viewtype for the elements to be added, and then method *addModulesToView(View v, UsedIds modules)* iterates over the list of selected *ids* and associates the Modules with the respective *ids* to the view.

8.3 Views

This section describes how the information from the Model is displayed to the user. The developed application uses Thymeleaf[3], a Java template engine for displaying dynamic templates. Tymeleaf can be fully integrated with the Spring Framework. In each template, it is possible to use the Thymeleaf syntax to access and display the Model Attributes added by the Controllers.

8.3.1 Concepts Templates

It was developed an unique template for each of the main Concepts of the Domain Model: The View, Scenario, Module, Component and Connector.

Each template displays the information about the concepts contained in the respective java objects, which were added as Model Attributes by the Controllers. They are roughly structured as follows:

1. Header with the concept name and type (if it is a Module, a Component, etc.). Figure 8.32 shows how this is implemented in the template for the Scenarios, however the implementation in the other concept templates is very similar.

```
<div class="panel-heading">
  <span id="scenarioName">
    <b><span th:text="'Scenario for '
      +${scenario.getQualityRequirement().getName()}+': '">
    </span>
    <span th:text="${scenario.getName()}"></span></b>
  </span>
  <!-- ... -->
</div>
```

Figure 8.32: Header with the Scenario name and Quality Requirement in the Scenario Template

   The *th:text* is a Thymeleaf attribute which sets the text of the HTML element to what is defined inside it. In this case, the attribute concatenates a piece of text with *${scenario. getQualityRequirement(). getName()}*. The *${}* syntax is used to access a variable, in this case the "scenario", which is a Java object added to the Model Attributes by the ScenarioController. This example accesses the Scenario's quality attribute and name to display;

2. Annotations associated with the concept. Figure 8.33 shows how Thymeleaf is used to iterate over the annotations associated with the scenario instance and display their information in the template.;

3. Concept's description text, if existent, and tools for adding or editing this text. Figure 8.34 shows how the text and the editor are represented in the Scenario Template. The editor is inside a collapsible * *, which is initially hidden but can be toggled by clicking the "Add/Edit Text" link. When the user clicks the "Submit" button, the added/edited text is retrieved from the *textarea* element using javascript. To note that it is possible to add inline thymeleaf expressions to javascript code. The retrieved text is sent in an Ajax POST request to the respective Controller. When the request is finished, the window is reloaded to show the updated text.

[3] http://www.thymeleaf.org/

```
<div>
  <ul class="list-group" style="margin-top:10px">
    <li class="list-group-item" th:each=" ann : ${scenario.getJsonAnnotations()}">
      <p> <b> Document Quote: </b> <i> <span th:text="'(...)' +
       ${ann.quote} + '(...)'"> </span> </i> </p>
      <p> <b> Comment: </b> <span th:text="${ann.text}"> </span> </p>
      <p> <b> Tag: </b> <span th:text="${ann.tag}"> </span> </p>
    </li>
  </ul>
</div>
```

Figure 8.33: Annotations represented in the Scenario Template

```
<div id="EditText">
  <a href="#toggleEditor" data-toggle="collapse">
    <span data-toggle="tooltip" data-original-title="Add/Edit Text">
    </span>
  </a>
  <div th:id="'textViewer'+${element.getExternalId()}"
    th:text="${element.getText()}">
  </div>
  <div class="collapse">
    <textarea  th:id="'text'+${element.getExternalId()}"
      th:text="${element.getText()}"></textarea>
    <button th:id="'submitText'+${element.getExternalId()}" type="submit">
      Submit
    </button>
  </div>
</div>

<script type="text/javascript" th:inline="javascript">
  $("#submitText"+/*[[${element.getExternalId()}]]*/).click(function(){
    var elemtext = document.getElementById("text"
        +/*[[${element.getExternalId()}]]*/).value;
    var ident = /*[[${element.getIdentifier()}]]*/;
    var loc = /*[[@{/setScenarioText/} + ${docId}
        + '/'+${element.getExternalId()}]]*/;
    var content = {text: elemtext}
    $.ajax(loc,{
      method: "POST",
      data: content
    }).done(function(data, textStatus, jqXHR){
      window.location.hash="#"+ident;
      window.location.reload(true);
    });
  });
</script>
```

Figure 8.34: Description text display and edition tools in the Scenario Template

Chapter 9

Evaluation

Evaluation here...

Chapter 10

Future Work

future work...

Chapter 11

Conclusion

The success of social software is very dependent on the people that use it, since it is focused on the interactions and collaboration between their users.

Although the implemented platform is more focused on extracting and structuring knowledge from an unstructured source, i.e., a plain text article, it allows a set of users, authenticated within the platform, to visualize the same document and collaborate together in building a structured representation of said document.

...conclusoes dos testes...

Bibliography

[1] R. Pereira, M. C. C. Baranauskas, and S. R. P. da Silva, "Social software building blocks: Revisiting the honeycomb framework," in *Information Society (i-Society), 2010 International Conference on*, pp. 253–258, IEEE, 2010.

[2] C. Shirky, "A group is its own worst enemy," in *The Best Software Writing I*, pp. 183–209, Springer, 2005.

[3] R. Klamma, M. A. Chatti, E. Duval, H. Hummel, E. T. Hvannberg, M. Kravcik, E. Law, A. Naeve, and P. Scott, "Social software for life-long learning," 2007.

[4] B. E. Kolko, E. Johnson, and E. Rose, "Mobile social software for the developing world," in *Online Communities and Social Computing*, pp. 385–394, Springer, 2007.

[5] D. L. Wiley, "Tagging: People-powered metadata for the social web," 2008.

[6] M. A. Chatti, M. Jarke, and D. Frosch-Wilke, "The future of e-learning: a shift to knowledge networking and social software," *International journal of knowledge and learning*, vol. 3, no. 4, pp. 404–420, 2007.

[7] G. Smith, "Social software building blocks," *Retrieved November*, vol. 5, p. 2010, 2007.

[8] B. J. Fogg, "Persuasive technology: using computers to change what we think and do," *Ubiquity*, vol. 2002, no. December, p. 5, 2002.

[9] H. Oinas-Kukkonen and M. Harjumaa, "Persuasive systems design: Key issues, process model, and system features," *Communications of the Association for Information Systems*, vol. 24, no. 1, p. 28, 2009.

[10] H. Oinas-Kukkonen and M. Harjumaa, "Towards deeper understanding of persuasion in software and information systems," in *Advances in Computer-Human Interaction, 2008 First International Conference on*, pp. 200–205, IEEE, 2008.

[11] P. Mavrodiev, C. J. Tessone, and F. Schweitzer, "Quantifying the effects of social influence," *Scientific reports*, vol. 3, 2013.

[12] R. E. Guadagno and R. B. Cialdini, "Preference for consistency and social influence: A review of current research findings," *Social Influence*, vol. 5, no. 3, pp. 152–163, 2010.

[13] M. Oduor, T. Alahäivälä, and H. Oinas-Kukkonen, "Persuasive software design patterns for social influence," *Personal and Ubiquitous Computing*, vol. 18, no. 7, pp. 1689–1704, 2014.

[14] A. Bandura and D. C. McClelland, "Social learning theory," 1977.

[15] R. B. Zajonc *et al.*, *Social facilitation*. Research Center for Group Dynamics, Institute for Social Research, University of Michigan, 1965.

[16] R. B. Cialdini, “Influence: The psychology of persuasion,” 1993.

[17] T. W. Malone and M. R. Lepper, “Making learning fun: A taxonomy of intrinsic motivations for learning,” *Aptitude, learning, and instruction*, vol. 3, no. 1987, pp. 223–253, 1987.

[18] A. Stibe, H. Oinas-Kukkonen, and B. White, “Exploring the effects of social influence on user behavior targeted to feedback sharing,” in *Proceedings of the IADIS WWW/Internet Conference (ICWI), Madrid, Spain*, 2012.

[19] M. Forestier, A. Stavrianou, J. Velcin, and D. A. Zighed, “Roles in social networks: Methodologies and research issues,” *Web Intelligence and Agent Systems*, vol. 10, no. 1, pp. 117–133, 2012.

[20] S. P. Borgatti and M. G. Everett, “Notions of position in social network analysis,” *Sociological methodology*, vol. 22, no. 1, pp. 1–35, 1992.

[21] S. Nadel, “The theory of social structure.,” 1957.

[22] S. P. Borgatti and M. G. Everett, “Two algorithms for computing regular equivalence,” *Social Networks*, vol. 15, no. 4, pp. 361–376, 1993.

[23] R. L. Breiger, S. A. Boorman, and P. Arabie, “An algorithm for clustering relational data with applications to social network analysis and comparison with multidimensional scaling,” *Journal of Mathematical Psychology*, vol. 12, no. 3, pp. 328–383, 1975.

[24] M. Steyvers, P. Smyth, M. Rosen-Zvi, and T. Griffiths, “Probabilistic author-topic models for information discovery,” in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 306–315, ACM, 2004.

[25] A. McCallum, X. Wang, and A. Corrada-Emmanuel, “Topic and role discovery in social networks with experiments on enron and academic email.,” *J. Artif. Intell. Res.(JAIR)*, vol. 30, pp. 249–272, 2007.

[26] A. Daud, J. Li, L. Zhou, and F. Muhammad, “A generalized topic modeling approach for maven search,” in *Advances in Data and Web Management*, pp. 138–149, Springer, 2009.

[27] J. Zhang, M. S. Ackerman, and L. Adamic, “Expertise networks in online communities: structure and algorithms,” in *Proceedings of the 16th international conference on World Wide Web*, pp. 221–230, ACM, 2007.

[28] N. Agarwal, H. Liu, L. Tang, and P. S. Yu, “Identifying the influential bloggers in a community,” in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pp. 207–218, ACM, 2008.

[29] S. Vavilis, M. Petković, and N. Zannone, “A reference model for reputation systems,” *Decision Support Systems*, vol. 61, pp. 147–154, 2014.

[30] L. Liu, M. Munro, and W. Song, “Evaluation of collecting reviews in centralized online reputation systems.,” 2010.

[31] S. Dencheva, C. R. Prause, and W. Prinz, “Dynamic self-moderation in a corporate wiki to improve participation and contribution quality,” in *ECSCW 2011: Proceedings of the 12th European Conference on Computer Supported Cooperative Work, 24-28 September 2011, Aarhus Denmark*, pp. 1–20, Springer, 2011.

[32] C. R. Prause and S. Apelt, “An approach for continuous inspection of source code,” in *Proceedings of the 6th international workshop on Software quality*, pp. 17–22, ACM, 2008.

[33] L. Liu and M. Munro, “Systematic analysis of centralized online reputation systems,” *Decision support systems*, vol. 52, no. 2, pp. 438–449, 2012.

[34] S. A. Golder and B. A. Huberman, “Usage patterns of collaborative tagging systems,” *Journal of information science*, vol. 32, no. 2, pp. 198–208, 2006.

[35] J. W. Tanaka and M. Taylor, “Object categories and expertise: Is the basic level in the eye of the beholder?,” *Cognitive psychology*, vol. 23, no. 3, pp. 457–482, 1991.

[36] T. Coates, “Two cultures of fauxonomies collide,” *http://www. plasticbag. org/archives/2005/06/two_cultures_of_fauxonomies_collide/. Last access: May*, vol. 8, p. 2008, 2005.

[37] E. Hovy, R. Navigli, and S. P. Ponzetto, “Collaboratively built semi-structured content and artificial intelligence: The story so far,” *Artificial Intelligence*, vol. 194, pp. 2–27, 2013.

[38] P. Domingos, “Toward knowledge-rich data mining,” *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 21–28, 2007.

[39] W. Wong, W. Liu, and M. Bennamoun, “Ontology learning from text: A look back and into the future,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 4, p. 20, 2012.

[40] T. R. Gruber, “A translation approach to portable ontology specifications,” *Knowledge acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[41] J. Tang, H.-f. Leung, Q. Luo, D. Chen, and J. Gong, “Towards ontology learning from folksonomies.,” in *IJCAI*, vol. 9, pp. 2089–2094, 2009.

[42] K. Kotis and A. Papasalouros, “Automated learning of social ontologies,” *Ontology Learning and Knowledge Discovery Using the Web: Challenges and Recent Advances*, 2011.

[43] P. Mika, “Ontologies are us: A unified model of social networks and semantics,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 1, pp. 5–15, 2007.

[44] A. Weichselbraun, G. Wohlgenannt, and A. Scharl, “Augmenting lightweight domain ontologies with social evidence sources,” in *Database and Expert Systems Applications (DEXA), 2010 Workshop on*, pp. 193–197, IEEE, 2010.

[45] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI series in software engineering, Addison-Wesley, 2003.

[46] P. Clements, *Documenting Software Architectures: Views and Beyond*. SEI series in software engineering, Addison-Wesley, 2003.

[47] G. E. Krasner, S. T. Pope, *et al.*, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[48] T. Reenskaug and J. Coplien, “The dci architecture: A new vision of object-oriented programming. artima developer,” 2009.

[49] J. Cachopo and A. Rito-Silva, “Combining software transactional memory with a domain modeling language to simplify web application development,” in *Proceedings of the 6th international conference on Web engineering*, pp. 297–304, ACM, 2006.

[50] J. M. P. Cachopo, *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Universidade Técnica de Lisboa, 2007.