Collaborative Platform for Analysis of Software Systems

Catarina Isabel Carvalho Santana

Instituto Superior Técnico,
catarina.santana@tecnico.ulisboa.pt

Abstract. The Software Architectures course at Instituto Superior Técnico teaches students the most important concepts on design and architecture of software systems and helps students to apply these concepts to real and 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 was analyzed and a social platform was developed to help solving these problems.

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

1 Introduction

Analyzing and discussing big, real, open-source and highly complex software system is a very important part of the Software Architectures course at Instituto Superior Técnico. 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. This thesis focuses on providing a solution for this problem with the use of social software and knowledge structuring strategies.

This document is organized as follows: Section 2 gives a more detailed description about the problem of applying theoretical concepts to practical examples in the context of the Software Architectures course. Section 3 elicits the main goals of this thesis. Section 4 presents the state-of-the-art in the areas of Social Software and Knowledge Structuring. Section 5 gives a small description of the developed solution. Sections 6, 7 and 8 describe the solution domain model, architectural details and implementation. Finally, Section 9 will show the assessment of the developed solution and 10 will describe what can still be added to it.

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 where they 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 students took notes of their colleagues' presentations or 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.

3 Objectives

To solve the problems mentioned in Section **??**, a collaborative platform was planned and developed, 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 goal of the platform is to provide ways for annotating the text on the case descriptions, which will help students organizing their thoughts and creating the structure, and use elements of social software, as a way of promoting collaboration, mutual aid, learning and even some competition between students.

The existence of this collaborative should provide 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.

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 sections 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 and **Reputation Systems**, which is the 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, which consists of assigning keywords to documents or parts of documents, **Semi-structured content**, which consists of providing a way for structuring knowledge without such strict rules as, for example, an ontology or a taxonomy and **Ontology Learning**, extracting knowledge from text.

4.1 Honeycomb Framework

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" [21] applies to our collaborative platform, as it should allow students with different personalities and opinions to participate by identifying architectural elements, asking questions, etc. in order to 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 [24]: **Identity**, the unique identifier of a user within the system, which applies to the platform to develop in the sense that each student is a unique person and must have a unique identification in the platform; **Presence**, resources that allow knowing if certain identity is online, which are not strictly necessary in the platform; **Relationship**, way to determine how users can relate/are related to others, which applies to the platform as students are related to each other both as colleagues as and group elements, and to the teachers; **Reputation**, way of knowing the status of a user in the system, which could be used to assess the quality and relevance of the contributions to the platform; **Groups**, possibility to form communities of users that have common interests, which applies to the platform as students form work groups; **Conversation**, resources for communication among the users (synchronous and/or asynchronous). In the context of the platform, conversation should be done asynchronously with comments and discussions; **Sharing**, possibility of sharing objects that are important to the users (videos, images, etc), which applies to this context as students may want to share documentation or other media with their colleagues or their group;

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, 20].

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, and **Shaping Outcome**, formulate a pattern for a situation where one did not exist before [19]. 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 describes a change in one's behavior (or attitudes or beliefs), caused by external pressures [12]. In our context, students may not contribute to the platform at first, but may feel compelled to when seeing their colleagues' contributions. 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) [18].

The **Social Learning and Facilitation** pattern has the main purpose of using 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 [18]. In the context of our collaborative platform, students may not contribute at first, but may feel compelled to as they see their colleagues contributing.

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. Some people may see this competition as a source of anxiety, so participation must be voluntary [18]. Adding a score for each student and a rank hierarchy based on the range of scores will start a hopefully healthy competition between them. Anonymity should be an option in the platform, to remove sources of anxiety.

The **Cooperation** pattern has the main purpose of providing software features that allow users to engage in mutual goals and support each other. The motivation for this pattern is that it is easier to cooperate if people are engaged in mutual objectives. However, some people prefer to perform alone, so cooperation must not be forced [18]. The students in the 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 there are several concepts and structures to identify on a single description and students may have difficulties in identifying them, so dividing the main goal in several smaller goals and have discussions around them would be useful.

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 sometimes need a reason to focus on reaching their goals. Having their efforts recognized may be a good reason to keep the good work[18]. As the system should have a score system, it could be used as a way of creating a weekly top ten of the students who achieved most points in that week. Recognition is then achieved when students see their names highlighted.

4.3 Roles in Social Networks

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 [9, 17]. There are two categories of roles: *Non-explicit roles*, which are not defined a-priori but can be inferred, and *Explicit roles*, which are predefined types of actors in the social network, such as "experts" or "influencers".

In the platform to develop, it is possible to define a priori the two main roles: the **Students**, which are the most active users in the platform as they contribute to the analysis of the software descriptions, and the **Teachers** which corresponds to a very small number of users in the platform. Their number of contributions is smaller and they are mostly for adding correction to the contents added by the students to the platform.

4.4 Reputation Systems

Reputation systems are of extreme importance for certain kinds of applications, namely e-commerce websites, where money transactions are executed and the reputation of a seller will denote the degree of trust that possible buyers will have in them [25].

For the platform to develop, reputation does not play such an important role, but will add a motivational component to the platform: Students rate their colleagues' contributions and should try to get and keep a high reputation score. The motivation behind the importance of including a reputation system in the platform is the study conducted [6], concerning the lack of participation in Moknowpedia, a Wiki system. A reputation system was added to the wiki in order to solve these problems and improve both content quality and quantity[22], and results showed an increase of 62% in the number of article revisions and an increase of 42% in the number of viewed articles.

Requirements and Features of the Reputation System: A framework for analysis of reputation systems is proposed in [25], where the general requirements for reputation systems are elicited and the corresponding features needed for their fulfillment. Given the context of the platform, its reputation system requires that **Ratings and Reputation should discriminate user behavior**, as these scores should allow to identify students with few and/or less relevant contributions and students with many and/or very relevant ones, **the reputation system should be able to discriminate "incorrect" ratings**, as malicious users are present everywhere and can give intentionally inaccurate ratings, **Users should not be able to modify ratings, reputation values and calculate their own reputation**, as this data should not be accessible to them, and the score should be calculated by the reputation system.

To satisfy the identified requirements, the reputation system should feature a range of values to represent **trust and distrust**, where low values indicate students with few and/or less relevant contributions and high values indicate students with many and/or very relevant ones, **absolute reputation values**, calculated independently for each user, and identification of **origin and target** of the ratings, to prevent self-ratings and identify potential malicious users

Components of a Reputation System: Reputation systems are divided in four components[15, 16], and each have a set of defined criteria for their evalutation: **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 and the **Feedback Loop**, the collection of feedback of the output (review of the review), which does not apply to the context of the platform.

Concerning the **Input** component, ratings are collected in the platform and are given by the students that are logged in. Only a single property is collected, the rating assigned.Concerning the **Processing** component, a weighted average algorithm is the best choice to aggregate ratings in this simple system, and information should be updated as soon as the ratings are assigned. Both the algorithm and the system have a very low complexity. Concerning the **Output** component, the end users of the information are the students registered in the platform, and this information is only available inside the platform. The reputation information consists in the result of the weighted average algorithm.

4.5 Collaborative Tagging

Collaborative tagging is the practice of allowing anyone to freely attach keywords or tags to content [10]. Tagging-based systems contrast with taxonomies as they are neither exclusive nor hierarchical, and allow to identify content as being about a great variety of things simultaneously [10]. The tags added to content may describe the content itself, or describe the category in which the content falls [5]. It is possible to identify several functions for the tags, within the system that uses the tagging system [10].

In this platform we are aiming for using tags from a closed taxonomy as a way of identifying the main parts of a software system within a software description, namely stakeholders, scenarios and their parts, tactics, etc., and use the tagged text to create a structured description of the document.

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) [13]. However, unstructured resources do not allow for complex inference chains[7] and information is not ontologized[13], and structured resources require a huge amount of effort to create and maintain[13]. Semi-structured contents try to create a middle-ground between these two types. The most important example of their use is the Wikipedia, which is a repository of webpages, related with each other by links. Massive online collaboration allow for updated high quality and wide coverage of contents. To sum up, using semi-structured resources provides the best of both worlds: high quality information with wide coverage of almost all domains [13].

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, and have all the parts that constitute a software system described in a structure similar to a Wikipedia, instead of scattered along a ten or more pages article. This will be accomplished by creating templates for the concepts and associating information from the document to them. Student collaboration in the platform will lead to good quality structured descriptions of the articles.

4.7 Ontology Learning

Research on Ontology learning from text has evolved over the years and there are several open challenges for this field [26]. Ontologies are defined as "effectively formal and explicit specifications in the form of concepts and relations of shared conceptualizations" [11], 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.

Regarding the context of the platform, all the concepts used to tag the document contents (the concepts of Software Architectures) and their will be defined a priori. Software description documents do not contain these concepts, but the students' motivation is to find parts of the text that correspond to the concepts predefined and tag them accordingly. This facilitates the comprehension of the text contents and the creation of the semi-structure as mentioned previously.

5 Solution

To solve the problems elicited throughout this document, it was developed a Web Application to be used by students and teachers, both in class and home environments. 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, **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, which are parts of selected text annotated with tags, and **templates** for a set of Software Architectures concepts, which aggregate the corresponding annotations and describe the concepts.

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

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 6.2 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.

Scenarios: A scenario is used to capture and express the quality requirements of a system. The considered qualities are **Availability**, **Interoperability**, **Modifiability**, **Performance**, **Security**, **Testability** and **Usability**.

A scenario consists of six parts[1]: The **Source of Stimulus**, an entity that generates the stimulus; The **Stimulus**, a condition that arrives at the system; The **Environment**, the system condition when the stimulus occurs; The **Artifact**, the part of the system stimulated; The **Response**, what is done in response to the stimulus; And the **Response Measure**, the way of measuring the response. 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.

Views: A view is a representation of a set of system elements and the relationships associated with them [4]. 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, which are specializations of element and relation types, together with a set of constraints on how they can be used.Views can fall into three viewtype categories:

The **Module Viewtype**, which documents the system principal units of implementation. The elements are the *Modules* and relationships between them can be of type part-whole, dependency and generalization/specialization; The **Component & Connector Viewtype**, which documents the system units of execution. The elements are the Components and the Connectors, and the relationships can be of type *"Attachment"*, associating components to connectors, and *"Interface"* Delegation, associating component ports and connector roles to other ports and roles, respectively; and the **Allocation Viewtype**, which documents the relationships between a system's software and its development and execution environments. The elements are the *Software Element* (Module, Components and Connector) and the *Environmental Element*. The relationships are of type *"Allocated-to"*, where software element is allocated to an environmental element.

6.2 Model

Figure 1 shows the concepts and relations described in Section 6.1 in a Domain Model Diagram.

The scenario elements are associated to the Scenario, and each Scenario captures a single Quality Requirement and has a set of Tactics. A View is associated with a Viewtype, which refines into a style. The View contains a set of elements, which have relations with each other. The Scenario, View and Element are considered the main concepts of the domain, and they have a dedicated template in the developed solution. 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 6.2 is represented in the developed solution by using specific templates for these concepts.

It did not make sense to have a template for each and every one of them, as 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 considered one of the main concepts and has its own dedicated template, in which are present the quality attribute, the elements and the tactics.

Figure 2 shows a Schema example 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.

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. The templates for the View and View Elements follow the same idea of the Scenario schema. The template for the elements represents the element information and also its relations with other elements. The View template has the particularity that, besides including information about the view, also includes the templates of the elements present in it.

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 section, it is shown how these templates are filled with information extracted from a software description article. The next subsections 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.

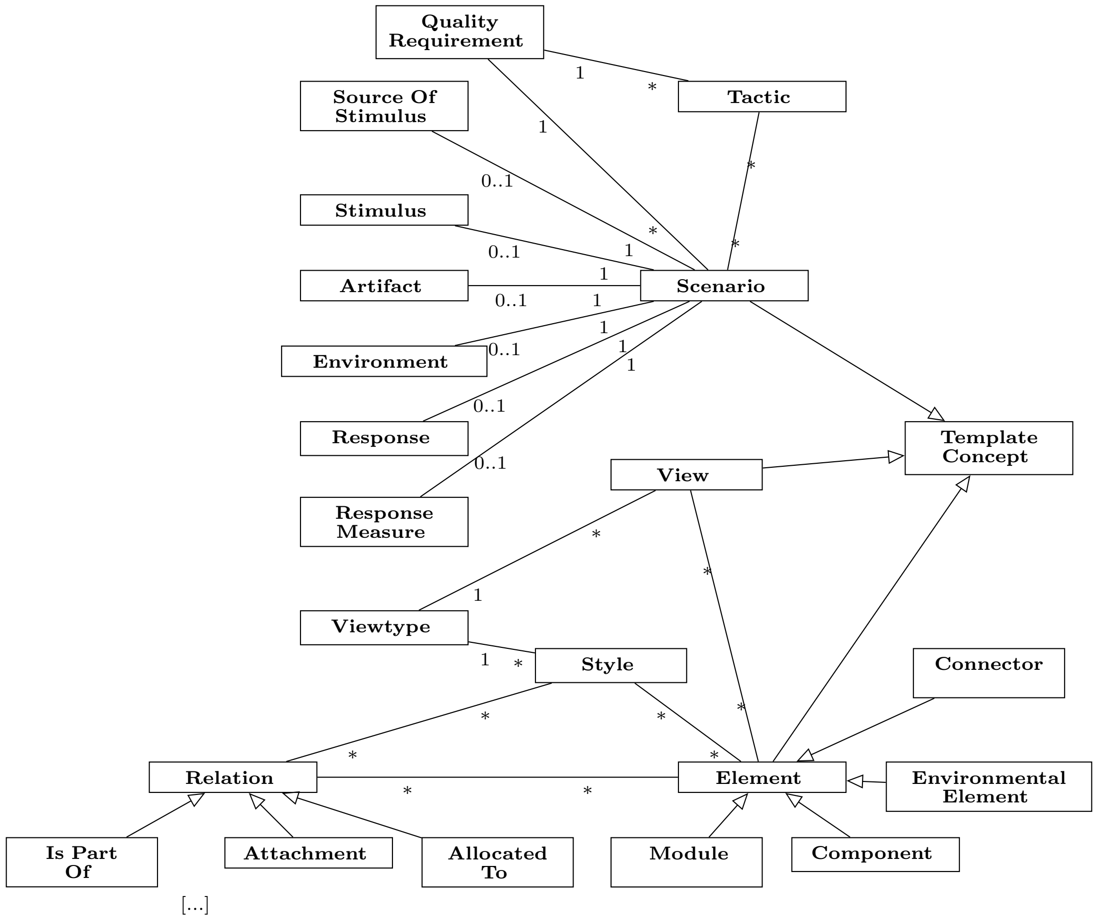


Fig. 1. Domain Model showing the Software Architectures concepts and how they are related

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 3 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.

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 4. 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.

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 5 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”.

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 6 it is possible to see the association between the Annotation and Document entities, which reflects the mentioned constraints.

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.

```
<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>
```

Fig. 2. Schema for the Scenario template

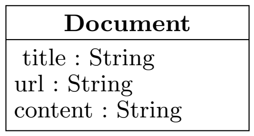


Fig. 3. The Document Entity

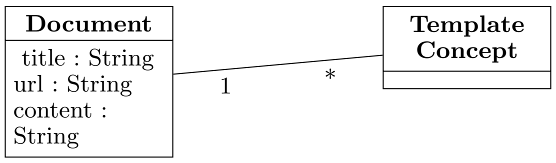


Fig. 4. Association between Document and Concept entities

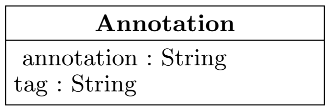


Fig. 5. The Annotation Entity

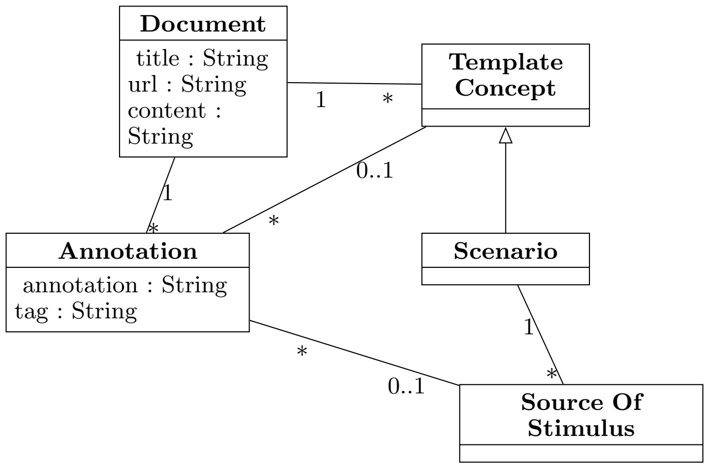


Fig. 6. Associations between Document, Annotation and the domain model concepts

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 6 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 1. The figure exemplifies these associations with the Scenario, which is a main concept, and the Source of Stimulus, but all the other concepts and main concepts - all the Scenario Elements, Modules, Components, etc. - 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, 8 and 9 show the Schemas presented in Section 6.3 of Chapter **??**, now including the annotations and the user text.

8 Implementation

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

This architecture is composed of three elements [14]:

- 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 [14, 23]:

- 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 **??**, 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 [2, 3]. 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.

```
<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>
```

Fig. 7. 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>
```

Fig. 8. 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>
```

Fig. 9. Module Viewtype view Schema enriched with annotations and user text

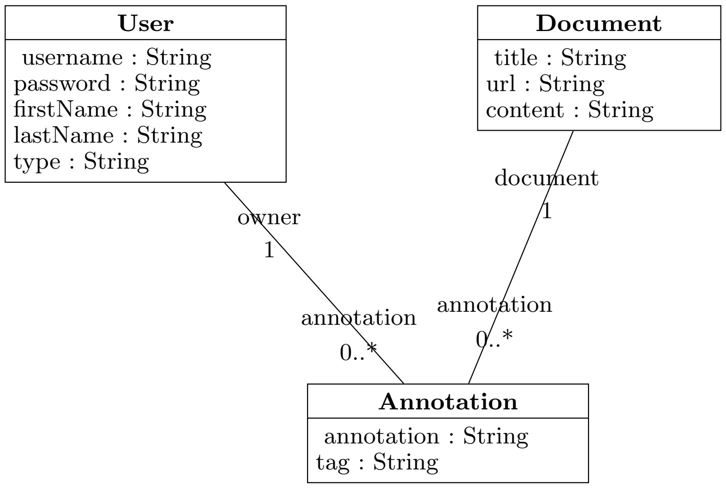


Fig. 10. Document, Annotations and Users in the implemented Model

Annotations As seen in Chapter **??**, 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.

Similarly to what is shown in Figure 6 of Chapter **??**, 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 10. The "User" entity saves users registered in the system. When a user creates an annotation, it is associated to him.

Scenarios The Scenario is a main concept of the domain, and therefore is connected to the Document, as seen in Figure 4.

Scenarios are represented in the implemented model in a very similar way to what is shown in Figure 1. However, there is a difference in the implemented model regarding how Quality Requirements and Tactics are represented.

In Figure 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: 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 11 shows the Scenario, Tactic and Quality Requirement entities, and how they are related in the implemented model.

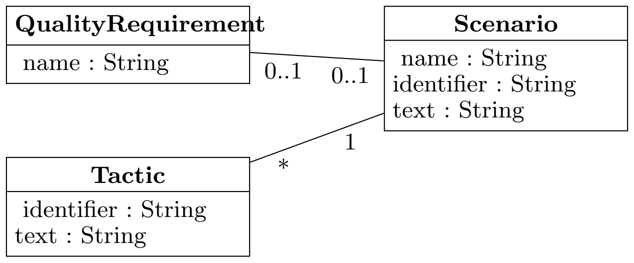


Fig. 11. Quality Requirements and Tactics detail in the implemented Model

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. Tactics can be added once inside the Scenario Template. 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 12. 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.

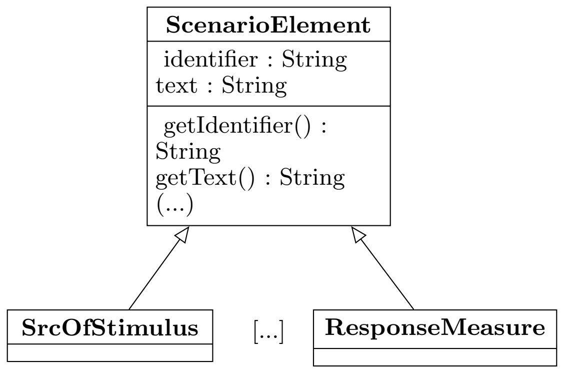


Fig. 12. The ScenarioElement entity

The "identifier" parameter of the entities, seen in Figures 11 and 12 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 6, with the Scenario and ScenarioElement entities being associated with the Annotation.

Modules The Modules are the elements of the Module Viewtype Views. Figure 13 shows how a Module is represented in the implemented model.

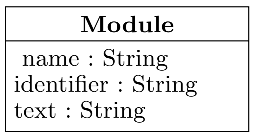


Fig. 13. The Module entity

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 6. The Module is also associated with the set of Annotations that describe it.

As seen in the abstract domain model present in Figure 1 of Chapter **??**, the elements of a view have a certain type of relationship with each others. Although the abstract domain model in Figure 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. There are, then, eight types of relations between Modules[4] defined in the Domain Modeling Language:

- *Is-Part-Of* - a Module can be part of one and only one parent Module. Figure 14 shows how this relation was implemented. A Module can have only one other Module set as parent, but can have many children Modules;
- *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 15. A Module can use and be used by many other Modules;

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

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

Fig. 14. Is-Part-Of relation between Modules in the implemented model

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

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

Fig. 15. Uses relation between Modules in the implemented model

- *Is-A* - a relation of generalization, in which a Module is a generalization, similar to a superclass, of other modules. The implementation of this relation is similar to the implementation of the "Uses" relation in Figure 15, as a Module can be a generalization of many Modules and a specialization of many others (multiple inheritance).
- *Crosscuts* - an Aspect Module, which implements a crosscutting concern of the system, is bound to a Module that is affected by that crosscutting concern. The Implementation of this relation is similar to the one shown in Figure 15, as a Module can be a crosscutting concern of many Modules, and be crosscutted by other Aspect Modules.
- *One-To-One, One-To-Many, Many-to-Many* - logical associations between Data Entity Modules, similar to the UML associations. The implementation of these relations is, again, similar to the implementation of the "Uses" relation in Figure 15. 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 16.

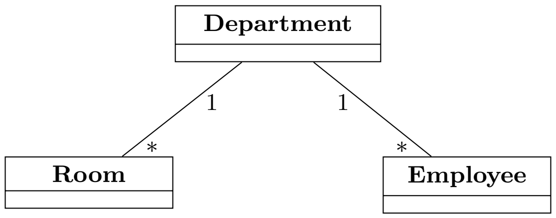


Fig. 16. Example of the One-To-Many relation between Modules

- *Aggregation* - an aggregation relation. The implementation of this relation is similar to Figure 15, as a Module can be an aggregator of other several Modules, and be aggregated by many others;

Components and Connectors Components and Connectors are the elements of the Component & Connector Viewtype Views. Figure 17 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 **??**. 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, 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. Both the Component and Connector entities and also the Port and Role are associated to a set of Annotations that describe them. The associations of these entities to the Document and Annotation are similar to the ones in the example of Figure 6.

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 18.

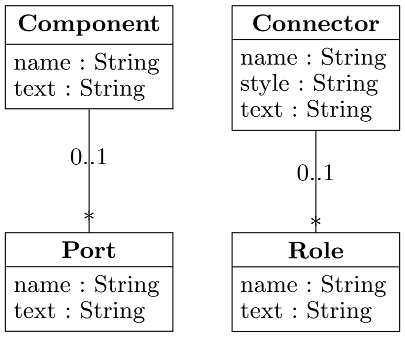


Fig. 17. Component and Connector entities in the implemented model

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

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

Fig. 18. Attachment relation between Component Ports and Connector Roles in the implemented model

Views The abstract domain model presented in Figure 1 of Chapter **??** 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 19 shows how the View is represented in the implemented model.

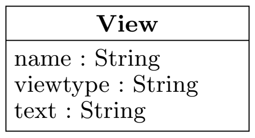


Fig. 19. View entity in the implemented model

In the abstract domain model of Figure 1 the View is associated with the Viewtype entity which, in 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.

A View is composed by a set of Elements, as seen in Figure 1. In the implemented domain model, the View has a similar association with the Module, Component and Connector entities. Figure 20 shows how these relations are implemented in the Domain Modeling Language.

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 6.

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.

[1] https://spring.io/

```
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..*; }
}
```

Fig. 20. Relation between View and the view elements: Module, Component and Connector

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 21 shows an example from the Document Controller described in 8.2, with a method that handles requests for the URL *'/selectDoc/id'*. Method *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.

```java
@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";
    }
}
```

Fig. 21. Example of a Controller class defined using the Spring Framework

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 1.

The **INDEX** endpoint has the particularity of iterating through all the Annotations associated with the Document with id *docId*. 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 22 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.

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

[2] http://jsoup.org/

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 1. REST API provided by the Annotation Controller

```java
@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;
}
```

Fig. 22. Example of how the INDEX endpoint adds information to the annotation body

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.
  Therefore, it's only logical to redirect this request to the Controller that handles operations over the respective domain entity. Figure 23 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;
}
```

Fig. 23. 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 24 shows the DocumentController class method which performs the redirect according to the annotation tag.
– 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 25 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.

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.

Figure 26 shows the method *addNewScenario*, which handles the request for adding a new Scenario 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.

```
@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;
}
```

Fig. 24. Redirecting a request to visualize an entity template in the DocumentController

```
@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";
}
```

Fig. 25. Retrieving the Structured Representation template in the DocumentController

```
@RequestMapping(value = "/addNewScenario/{docId}/{annotationId}/{scenarioName}/
{qualityRequirement}")
public RedirectView addNewScenario(/*...*/) {
    addScenarioToDocument(document,qualityRequirement ,scenarioName);
    //...
}
    @Atomic(mode = TxMode.WRITE)
private void addScenarioToDocument(Document d, String qualityRequirement,
String scenarioName) {
    Scenario s = new Scenario();
    QualityRequirement qr = new QualityRequirement();
    s.setQualityRequirement(qr);
    //...
    d.addScenario(s);
}
```

Fig. 26. Creating a new Scenario in ScenarioController

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 27 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")) {
        //associate annotation with the Scenario s
    }
    if (s.getElements().get(tag) != null) {
        //associate annotation with a ScenarioElement
    }
}
```

Fig. 27. 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 27.

Inside the template, these annotations can be associated with the added tactics. Figure 28 shows how a request to associate an Annotation with a specific Tactic is handled. 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(/*...*/) {
    //...
    linkAnnotationToTactic(s, t, a);
    //...
}
```

Fig. 28. 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 29 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(/*...*/) {
    updateText(scenario, text);
    //...
}

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

Fig. 29. Updating the Scenario text in the ScenarioController

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 **??**, the Elements of a View are related to each others, and in the Subsection 8.1, 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(/*...*/) {
    addParent(module, parent);
    //...
}
@Atomic(mode = TxMode.WRITE)
private void addParent(Module module, Module parent) {
    module.setParent(parent);
}
```

Fig. 30. Example of adding a "Is-Part-Of" relation between two modules

Figure 30 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.

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 31.

```
@RequestMapping(value = "/addPortToComponent/{docId}/{componentId}/{portName}")
public RedirectView addPortToComponent(/*...*/) {
    addPortToComponent(component, portName);
    //...
}

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

Fig. 31. 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 32.

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

```
@RequestMapping(value = "/moveAnnotationToPort/{docId}/{componentId}/
    {portId}/{annotationId}")
public RedirectView moveAnnotationToPort(/*...*/) {

    moveAnnotationToPort(component, port, annotation);
    //...
}
```

Fig. 32. Associating an Annotation with a Port in the ComponentController

to them in the ConnectorController class are very similar to the ones represented in the Figures 31 and 32, 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, 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 33 shows how an attachment relation is created between a Port and a Role.

```
@RequestMapping(value = "/attachPortToConnectorRole/{docId}/{portId}/{roleId}")
public RedirectView attachPortToRole(/*...*/) {
    attachPortToRole(port,role);
    //...
}
@Atomic(mode=TxMode.WRITE)
private void attachPortToRole(Port p, Role r) {
    p.setRole(r);
}
```

Fig. 33. Attachment of Component Ports to Connector Roles in the ComponentController

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 34 shows how a request to add a new Module Viewtype View is handled.

```
@RequestMapping(value = "/addNewView/{docId}/{annotationId}/Module Viewtype/{viewName}")
public RedirectView addNewModuleViewTypeView(/*...*/) {

    addModuleViewtypeToDocument(document, viewName, "Module Viewtype");
    //...
}
```

Fig. 34. 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 35 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.

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.

```
@RequestMapping(value = "/viewView/{docId}/{viewId}")
public String viewViewTemplate(/*...*/) {
    //...
    if(view.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")) {
        //add components and connectors information
    }
    return null;
}
```

Fig. 35. Request to see a View's Template in the ViewController

Figure 36 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*}, containing in its body the instance of the *UsedIds*

```
@RequestMapping(value = "/addModulesToView/{docId}/{viewId}",
    method = RequestMethod.POST)
public RedirectView addModulesToView(/*...*/@ModelAttribute UsedIds modules) {
    if(view.getViewtype().equals("Module Viewtype")) {
        addModulesToView(v, modules);
    }
    //...
}
@Atomic(mode = TxMode.WRITE)
private void addModulesToView(View v, UsedIds modules) {
    for (String id : modules.getUsed()) {
        //associate module with the view
    }
}
```

Fig. 36. How Modules are added to a View in ViewController

class that was added as model attribute (see Figure 35), 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 and, in each template, it is possible to use the its syntax to access and display the Model Attributes added by the Controllers.

A great advantage of Thymeleaf is the possibility of defining template fragments, which are pieces of code that can be defined in a separate file and be included in any template as desired. One of the main uses of fragments for the developed application is the page header, which is included in all templates and contains a small menu and the logout link. Figure 37 shows how a fragment is defined, in this case with the example of the header. Each fragment must have a unique name, which is "headerFragment" in this example. All html code of that fragment is then defined inside the *section* tags.

Including a predefined fragment in a template is as easy as Figure 38 shows. Sending a request to */headerFragment* will retrieve the whole file where the fragment was defined, and by specifying the fragment name, the html code defined inside *section* tags in the separate files will be included. This means that several fragments can be developed in one single file, and included individually in the templates by specifying their unique name.

The result of including the header fragment in a template is shown in Figure 39.

Another use for this Thymeleaf feature is in the templates defined for the main concepts - View, Scenario, Module, Component and Connector. Each of these templates are defined in separate <*section*> tags inside a file. This was done

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

```
<section layout:fragment="headerFragment">
  <nav class="navbar navbar-inverse" id="headernavbar">
    Header definition here
  </nav>
</section>
```

Fig. 37. Defining the Header fragment in Thymeleaf syntax

```
<section layout:include="@{/headerFragment} :: headerFragment"></section>
```

Fig. 38. Including a fragment in Thymeleaf syntax

because there are other templates where the information about the concepts is included, and it is much easier to include a fragment containing the whole information than writing the code again.

For example, inside a View template, information about the Elements included in the View must be shown, which is simply a matter of including the template of each element. Other example of the usefulness of these fragments is when showing the structured representation of a document, which consists of showing all the templates of the concepts associated with it. Again, this is as simple as including all the necessary templates.

Figure 40 shows the Scenario template. The templates use the information added by the Controllers to the Model Attributes, as seen in Figure 25 and display it in a structured way. The information added by the Controllers is actually Java objects retrieved from the Model. Thymeleaf allows to treat these attributes as Java objects, so it is possible to obtain the value of the class attributes, or call methods on that object.

For example, to obtain the Scenario title including the Quality Requirement and the Scenario name as it can be seen in Figure 40, it is used the Thymeleaf syntax seen in Figure 41. The *${}* syntax allows to access the variables, and the *th:text* attribute is in fact a Thymeleaf attribute which will set the text inside the *span* tag to the result of the variable access done inside the quotes. As it shown, it is also possible to concatenate the values returned with other static text to obtain the desired results.

Thymeleaf also provides automatic binding of forms information to provided objects. For example, when a user wants to add Modules to a Module Viewtype View, a modal window is prompted, listing all the existent Modules associated with the respective Document, as it is shown in Figure 42. The Modules are listed inside a form, with checkboxes that allow the user to select more than one Module to add.

In Figure 35, it is possible to see that the Controller creates and adds a new instance of the *UsedIds* as a model attribute. This instance is used by Thymeleaf to bind the value of the selected checkboxes in the form to the *List<String>* attribute of the class. Figure 43 shows how the form is created. A checkbox is added for each Module in the set. The *th:object* attribute defines that the variable “used” is the object to which the results will be bound, and the *th:field* attribute in the *input* tag defines that the value of the checkbox, defined by the *th:value* attribute, if it is chosen, must be bound to the field “used” of the class “UsedIds”, which is the list of type String.

When the form is submitted, Thymeleaf will add the selected values to the list, and send the object in the body of the POST request to the Controller, which will process them as seen in Figure 36.

9 Evaluation

The success of a software system is often dependent on the opinion of the people that will use it. Systems attractive and easy to use according to the target audience background are more likely to be highly used.

The developed system tries to make the task of reading, understanding and structuring a software description article into an easy one, providing a simple and clean interface.

As the main goal of the developed application is to be used by students and teachers, in both classroom and home environments, it was asked to the students enrolled in the Software Architectures course to test the application, namely the features for annotating text and structuring Scenarios from the annotations created.

The participating students were asked to fill a small survey afterwards to register their opinions. This survey consisted of two questions asking to evaluate the Usability and the adequacy of the application regarding Scenario creation in a scale of one to ten, and three open and optional questions, asking what did the student like the most about the application, what improvements could be done to it, and to register any other feedback the student may had.

A total of [NUMBER] students tested and evaluated the application.

Regarding the usability of the application, Figure ... shows the graph containing the ratings given by the students.

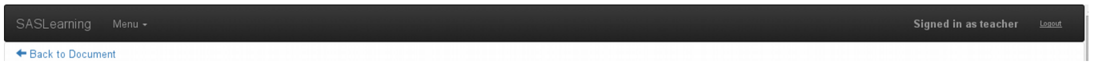
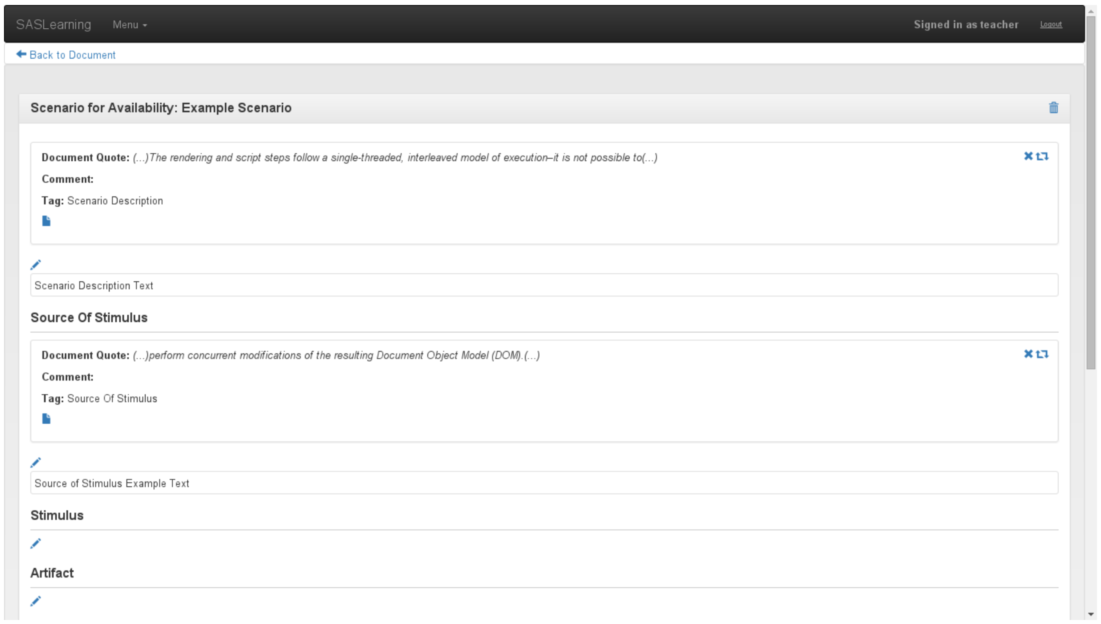


Fig. 39. How the header looks when included in a template


SASLearning Menu Signed in as teacher Logout

Back to Document

Scenario for Availability: Example Scenario

Document Quote: *(...)The rendering and script steps follow a single-threaded, interleaved model of execution–it is not possible to(...)*

Comment:

Tag: Scenario Description

Scenario Description Text

Source Of Stimulus

Document Quote: *(...)perform concurrent modifications of the resulting Document Object Model (DOM).(...)*

Comment:

Tag: Source Of Stimulus

Source of Stimulus Example Text

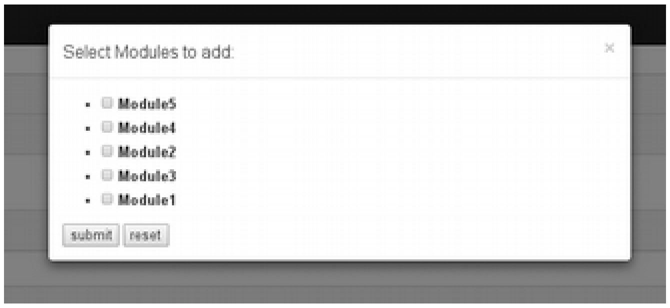
Stimulus

Artifact

Fig. 40. Example of the Scenario Template

```
<span th:text="'Scenario for '+${scenario.getQualityRequirement().getName()}+': '">
</span>
<span th:text="${scenario.getName()}"></span>
```

Fig. 41. How Thymeleaf accesses variables


Select Modules to add:

- Module5
- Module4
- Module2
- Module3
- Module1

submit reset

Fig. 42. Example of a Modal window containing a form

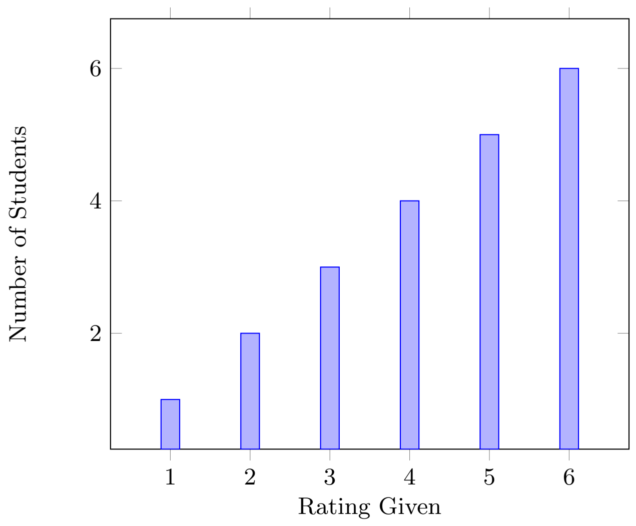
```
<form action="#" th:action="@{/addModulesToView/}" method="post">
  <ul>
    <li th:each="module : ${modules}" th:object="${used}">
      <input type="checkbox" th:field="*{used}"
        th:value="${module.getExternalId()}"></input>
      <label th:text="${module.getName()}"></label>
    </li>
  </ul>
</form>
```

Fig. 43. Example of a form with object for binding results

Conclusions....

Regarding the adequacy of the application to create Scenarios, Figure ... shows the graph with the ratings given by the students.

Conclusions...



Regarding the open questions... Introduzir conclusoes se as houver.

10 Future Work

The analysis of the state-of-the-art on Social Software and Knowledge Structuring done in Chapter **??** elicits a set of features that should be present in the solution.

However, developing a platform that provides structuring knowledge from an unstructured source and all the described social features was a very ambitious goal for the time span available.

The most important feature for this platform was, in fact, the knowledge structuring part, as the main goal of this thesis is to provide a solution to facilitate the analysis of the software description article and help the students to find the correlation between the concepts learned in the theoretical classes and these articles analyzed in the practical ones.

Therefore, the developed solution focused on the association of tags from a closed vocabulary to parts of text, and the creation of a semi-structured representation of the software description articles, where the elements of the Software Architectures are clearly distinguished and described by the corresponding parts of text and user text.

Finding the correct way to make these associations and representations was not a easy task, and there are still details of the developed solution that could be improved and new features that could be added in the future.

10.1 Allocation Viewtype

The Module and Component & Connector Viewtypes are the most common views present in a software description article, and were the ones that got most attention during development.

Allocation viewtype elements should be added to the platform in a future version, so Allocation Viewtype Views can be added to the structured representation of the article.

10.2 Graphical Representation of Views

The developed solution allows for the creation and representation of Modules, Components, Connectors and the Views that include these elements.

Module and Component & Connector Viewtypes are only represented textually in the platform. However, a good way to represent these views so that the relations between the elements are visible is to represent them in graphical diagrams.

Features to create or upload graphical representations of Views would be very useful for the platform as a way to improve comprehension about the Views and their elements.

10.3 Social Elements

Sections ??, ?? and ?? of Chapter ?? describing the Related Work list a series of social features that could be useful in the platform.

The developed platform contains social elements and allows for collaboration, as students are individually identified within the system and work together over a shared article. However, there are several other social software elements that could enrich the platform as a social system:

– **Groups:** This is the most important social aspect that could be added to the platform, as besides reading and analyzing articles in the practical classes, students must also read and analyze other articles for Group Assignments. The existence of features for the registration of work groups and having articles with group-access only could make it easier for the students to work on their group assignments.
– **Q&A and/or Discussion Forums:** It may not always be clear for a student which parts of the text correspond to which concepts. Although they can simply ask the teacher or a colleague, it could also be very useful to have a central place to ask questions and discuss decisions. Having a discussion forum and/or a Q&A system could also improve collaboration between students.
– **Reputation System:** Although its role would only be motivational, it is important that students feel motivated to participate in the platform, and a reputation system, assigning scores to, for example, annotations or comments, could be useful in this platform.
– **Spotlight:** Having a spotlight of the students with most participations and/or the highest reputation scores would provide not only extra motivation, but also promote some healthy competition between students.

11 Conclusions

In the context of the Software Architectures course, relating theory concepts with descriptions of real systems is a very hard job, specially for students that are learning these concepts for the first time. It is clear that a way for students to discuss and collaboratively structure their knowledge is needed.

This document presents the state-of-the-art on Social Software and Knowledge Structuring: The Honeycomb Framework for social software, design patterns for persuasive software, the roles of users in social software, reputation systems, collaborative tagging, semi-structured contents and ontology learning.

Based on the state-of-the-art presented and the context of the Software Architectures course, this document presents an architecture for a collaborative platform where students can create a structure to relate the theory concepts with the text from systems descriptions.

The main goal for this work is to develop the platform with the knowledge structuring elements and discussion methods, to allow students to discuss and organize thoughts. Additionally, other social software elements will be added, such as reputation systems, in order to boost motivation and encourage participation.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003.
2. João Cachopo and António 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*, pages 297–304. ACM, 2006.
3. João Manuel Pinheiro Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Universidade Técnica de Lisboa, 2007.
4. P. Clements. *Documenting Software Architectures: Views and Beyond*. SEI series in software engineering. Addison-Wesley, 2003.
5. Tom Coates. Two cultures of fauxonomies collide. *http://www. plasticbag. org/archives/2005/06/two_cultures_of_fauxonomies_collide/. Last access: May*, 8:2008, 2005.
6. Silviya Dencheva, Christian R Prause, and Wolfgang 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*, pages 1–20. Springer, 2011.
7. Pedro Domingos. Toward knowledge-rich data mining. *Data Mining and Knowledge Discovery*, 15(1):21–28, 2007.
8. Brian J Fogg. Persuasive technology: using computers to change what we think and do. *Ubiquity*, 2002(December):5, 2002.
9. Mathilde Forestier, Anna Stavrianou, Julien Velcin, and Djamel A Zighed. Roles in social networks: Methodologies and research issues. *Web Intelligence and Agent Systems*, 10(1):117–133, 2012.
10. Scott A Golder and Bernardo A Huberman. Usage patterns of collaborative tagging systems. *Journal of information science*, 32(2):198–208, 2006.
11. Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.
12. Rosanna E Guadagno and Robert B Cialdini. Preference for consistency and social influence: A review of current research findings. *Social Influence*, 5(3):152–163, 2010.
13. Eduard Hovy, Roberto Navigli, and Simone Paolo Ponzetto. Collaboratively built semi-structured content and artificial intelligence: The story so far. *Artificial Intelligence*, 194:2–27, 2013.
14. Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
15. Ling Liu and Malcolm Munro. Systematic analysis of centralized online reputation systems. *Decision support systems*, 52(2):438–449, 2012.

16. Ling Liu, Malcolm Munro, and William Song. Evaluation of collecting reviews in centralized online reputation systems. 2010.
17. SF Nadel. The theory of social structure. 1957.
18. Michael Oduor, Tuomas Alahäivälä, and Harri Oinas-Kukkonen. Persuasive software design patterns for social influence. *Personal and Ubiquitous Computing*, 18(7):1689–1704, 2014.
19. Harri Oinas-Kukkonen and Marja Harjumaa. Towards deeper understanding of persuasion in software and information systems. In *Advances in Computer-Human Interaction, 2008 First International Conference on*, pages 200–205. IEEE, 2008.
20. Harri Oinas-Kukkonen and Marja Harjumaa. Persuasive systems design: Key issues, process model, and system features. *Communications of the Association for Information Systems*, 24(1):28, 2009.
21. Roberto Pereira, M Cecilia C Baranauskas, and Sergio Roberto P da Silva. Social software building blocks: Revisiting the honeycomb framework. In *Information Society (i-Society), 2010 International Conference on*, pages 253–258. IEEE, 2010.
22. Christian R Prause and Stefan Apelt. An approach for continuous inspection of source code. In *Proceedings of the 6th international workshop on Software quality*, pages 17–22. ACM, 2008.
23. T Reenskaug and J Coplien. The dci architecture: A new vision of object-oriented programming. artima developer, 2009.
24. Gene Smith. Social software building blocks. *Retrieved November*, 5:2010, 2007.
25. Sokratis Vavilis, Milan Petković, and Nicola Zannone. A reference model for reputation systems. *Decision Support Systems*, 61:147–154, 2014.
26. Wilson Wong, Wei Liu, and Mohammed Bennamoun. Ontology learning from text: A look back and into the future. *ACM Computing Surveys (CSUR)*, 44(4):20, 2012.