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. In the practical component of this course, software description articles are analyzed to identify the theoretical concepts in practical examples. 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:

- The course has around one-hundred students per year. Each person is different, and while some students 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 for everyone, it is usually limited to the student's individual notes or the work done in groups.
- The case description is usually a fairly long document, and correlating the concepts learned in theory classes with the practical examples 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 2, 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" [20] 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 [22]: **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, 19].

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 [18]. 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) [17].

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 [17]. 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 [17]. 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 [17]. 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[17]. 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, 16]. 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 [23].

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[21], 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 [23], 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[14, 15], 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 [24]. 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.

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**, **Stimulus**, **Environment**, **Artifact**, **Response** and **Response Measure**. Each Scenario uses a set of **Tactics**, which achieve the quality requirements. Each quality requirement has a set of commonly used tactics.

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, which define 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 **Component & Connector Viewtype**, which documents the system units of execution, and the **Allocation Viewtype**, which documents the relationships between a system's software and its development and execution environments.

6.2 Model

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

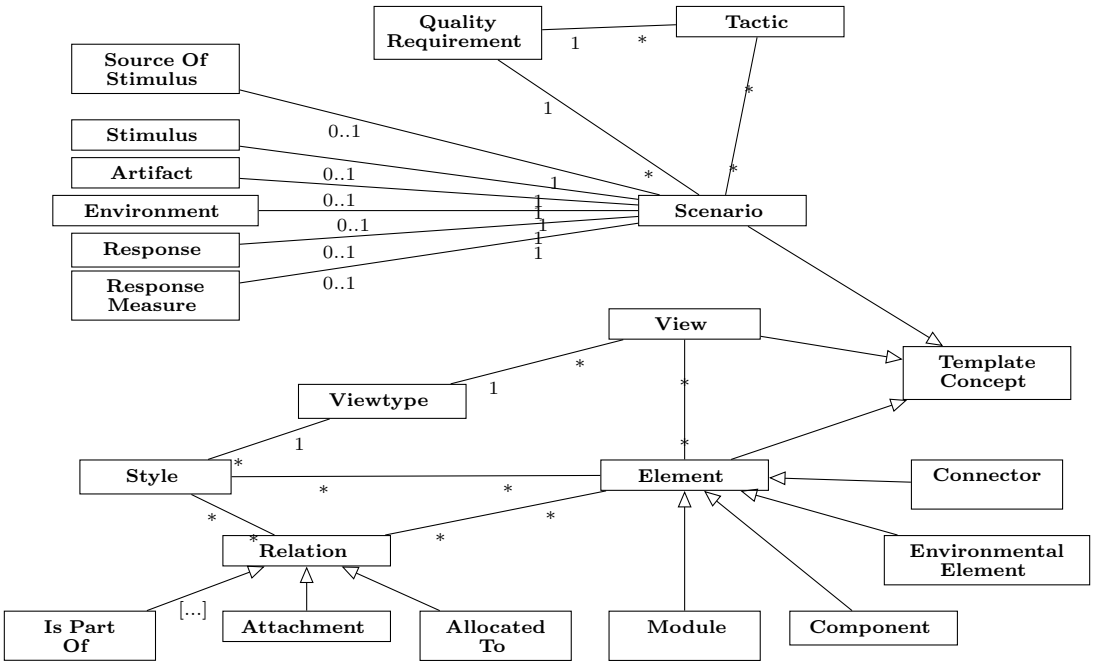


Fig. 1. Domain Model showing the Software Architectures concepts and how they are related

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 Elements 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 concept, as it is easier to see how they relate if they are in the same template, for example, it is easier to see the Scenario elements if they are inside the Scenario template. 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.

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

Section 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 main entities from the system: the Document and the Annotation.

```
<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

7.1 Document

The Document entity in this system corresponds to a software system description article. Figure 3 shows the Document entity in the system.

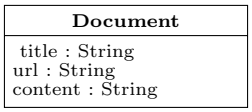


Fig. 3. The Document Entity

As the main purpose of this application is to create a structured representation of the document using the templates described in Section 6.3, the instances of the main concepts are associated with the Document.

7.2 Annotation

An annotation is a portion of text from the article, enriched with a tag. Figure 4 shows the Annotation entity in the system. The "annotation" field saves a JSON representation of the annotation data. Annotations are added to the system by selecting a

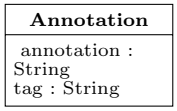


Fig. 4. The Annotation Entity

portion of text in the parsed document and setting it as an annotation. Therefore, they are always associated to the Document in which they were created and, as they describe partially or totally a Software Architectures concept, can be associated not only with the main concepts, but with all the concept entities of the domain model.

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 the creation of an annotation by using the AnnotatorJS interface, the association of that annotation to a domain concept and the redirection to that concept template, where annotations can be moved or deleted and other information, including description text, can be added.

8 Implementation

The application developed follows a Model-View-Controller architecture. 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 from Section 6 and the domain model is specified in the Domain Modeling Language, which was created for the Fénix Framework[2, 3].

Regarding **Annotations**, similarly to what is described in 7.2, in the implemented model Annotations are associated with the Document in which they were created, and to the User that created it.

Scenarios are represented in the implemented model in a very similar way to what is shown in Figure 1. However, the Quality Requirement is not associated with the Tactic, as the set of Quality Requirements and Tactics are not pre-defined in the implemented database, they are added upon Scenario creation. The Scenario is associated with the respective Document and the Annotations that describe it. Scenario elements are also associated with their respective annotations.

The **Modules** are the elements of the Module Viewtype Views. The Module entity in the implemented domain model is associated with the corresponding document upon its creation, and with the set of Annotations that describe it, as described in Section 7.

Modules are related to each others, as seen in Figure 1. Instead of defining these relations in a separate entity, they are represented in the implemented model with different associations from the Module entity to itself, as seen in Figure 5.

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

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

Fig. 5. Is-Part-Of relation between Modules in the implemented model

Components and Connectors are the elements of the Component & Connector Viewtype Views. A Component has a set of Ports, and a Connector as a set of Roles. Figure 6 shows how the Components, Connectors and their respective Ports and Roles are implemented.

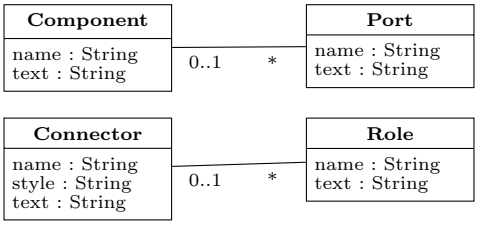


Fig. 6. Component and Connector entities in the implemented model

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.

Components are related with Connectors by the *Attachment* relation, in which Component ports are attached to roles of Connectors. The *Attachment* relation was defined by creating an association between the Port and Role entities, as shown in Figure 7.

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

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

Fig. 7. Attachment relation between Component Ports and Connector Roles in the implemented model

Regarding **Views**, a View is considered a main concept, and therefore has its own dedicated template. Figure 8 shows how it is represented in the implemented model.

The viewtype of a View is defined upon its creation and stored in the "viewtype" attribute, and the styles of a view are not explicitly stated, but they are implicit by the elements that are added to it. 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 9 exemplifies the association with the Module entity in the Domain Modeling Language.

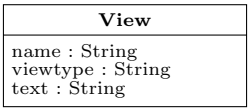


Fig. 8. View entity in the implemented model

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

Fig. 9. Relation between View and the view elements

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

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.

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.

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

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. To add a new document, the controller checks if the database already contains a document with the given URL and then extracts the article contents from that URL to add to the database. 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 other operations handled by this controller, such as redirecting operations to other controllers upon receiving a request to visualize the template of a concept or to associate annotations with domain entities as seen in Figure 10, and navigation to the page containing the structured representation of the document.

The **ScenarioController** 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 database, all associated with the newly created Scenario.

When the user chooses which Scenario to link the Annotation with, the controller verifies the tag associated with the Annotation, and add the Annotation either to the Scenario or to the corresponding element.

A Scenario is initially created without any tactics. Upon receiving the corresponding request, the controller adds a new "Tactic" to the database, associated with the corresponding Scenario. Inside the template, annotations can be associated with the specific tactics.

[1] https://spring.io/

```
@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. 10. Redirecting requests to associate annotations with domain entities in the DocumentController

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.

Similarly to the Scenario Controller, the **ModuleController** handles the creation and deletion of Modules. 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.

As Modules are related to each other, the Module Controller handles adding and removing other Modules to/from the possible relations. The modules to associate are selected within the template.

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. The implementation of the Connector Roles is similar to the implementation of the Ports.

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.

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. The ViewController distinguishes between two different templates when receiving a request to view the View's template: One for Module Viewtype views and one for the Component & Connector Viewtype views, to which only Components and Connectors can be added.

When, inside the template, 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.

8.3 Views

This section describes how the information from the Model is displayed to the user. The developed application uses Thymeleaf[2], a Java template engine for displaying dynamic templates.

Thymeleaf allows the definition of template fragments, which are pieces of code that can be defined in a separate file and be included in any template as desired. Fragments are used in the developed application in the page header, which is included in all templates, and in the concept templates. Figure 11 shows how a fragment is defined. Each fragment must have a unique name and all html code of that fragment is then defined inside the *section* tags.

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

Fig. 11. Defining the Header fragment in Thymeleaf syntax

Including a predefined fragment in a template is as easy as Figure 12 shows.

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

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

Fig. 12. Including a fragment in Thymeleaf syntax

The templates use the information added by the Controllers to the Model Attributes and display it in a structured way. This information is actually Java objects 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, in Figure 13, the quality requirement and the name of a Scenario object are accessed.

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

Fig. 13. How Thymeleaf accesses variables

Thymeleaf also provides automatic binding of forms information to provided objects. For example, Figure 14 shows a form with checkboxes, to select Modules.

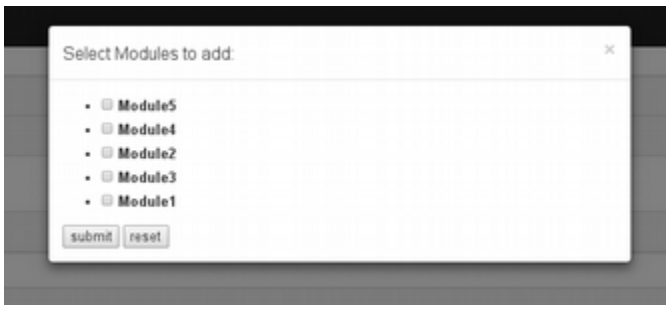


Fig. 14. Example of a Modal window containing a form

When the form is submitted, Thymeleaf will bind the selected values to an object passed, and send that object in the body of the POST request to the Controller, which will process them as seen in Figure **??**.

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.

Conclusions....

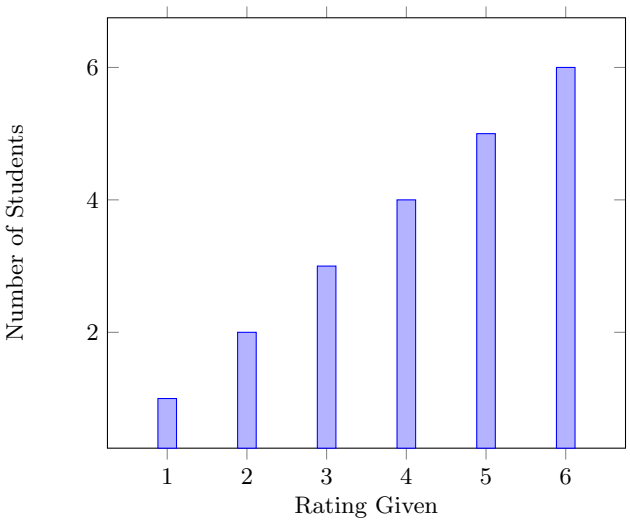
Regarding the adequacy of the application to create Scenarios, Figure ... shows the graph with the ratings given by the students.

Conclusions...

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

10 Future Work

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 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, which was the most important goal of this thesis.

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

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

It is easier to understand Views in a graphical representation, and it would be very useful for the platform to provide a way to create or upload these representations in the future

10.3 Social Elements

Sections 4.1, 4.2 and 4.4 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, such as **Groups**, so students can work on their group assignments, **Discussion Forums** to provide communication and cooperation, a **Reputation System** to motivate students to contribute, and a **Spotlight** with the students with highest reputation scores, to provide recognition of their efforts.

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. Ling Liu and Malcolm Munro. Systematic analysis of centralized online reputation systems. *Decision support systems*, 52(2):438–449, 2012.
15. Ling Liu, Malcolm Munro, and William Song. Evaluation of collecting reviews in centralized online reputation systems. 2010.
16. SF Nadel. The theory of social structure. 1957.
17. Michael Oduor, Tuomas Alahäivälä, and Harri Oinas-Kukkonen. Persuasive software design patterns for social influence. *Personal and Ubiquitous Computing*, 18(7):1689–1704, 2014.
18. 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.
19. 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.
20. 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.
21. 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.
22. Gene Smith. Social software building blocks. *Retrieved November*, 5:2010, 2007.
23. Sokratis Vavilis, Milan Petković, and Nicola Zannone. A reference model for reputation systems. *Decision Support Systems*, 61:147–154, 2014.
24. 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.