Design de Software

TornGes
Software Architecture Document (SAD)

CONTENT OWNERS: João David (49448), Ye Yang (49521)

DOCUMENT NUMBER:	RELEASE/REVISION:	RELEASE/REVISION DATE:
• 1	• 1.0	• 24th November, 2019
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•

Table of Contents

List of Figures

List of Tables

1 Documentation Roadmap

The Documentation Roadmap should be the first place a new reader of the SAD begins. But for new and returning readers, it is intended to describe how the SAD is organized so that a reader with specific interests who does not wish to read the SAD cover-to-cover can find desired information quickly and directly.

Sub-sections of Section 1 include the following.

- Section 1.1 (“Document Management and Configuration Control Information”) explains revision history. This tells you if you’re looking at the correct version of the SAD.

- Section 1.2 (“Purpose and Scope of the SAD”) explains the purpose and scope of the SAD, and indicates what information is and is not included. This tells you if the information you’re seeking is likely to be in this document.

- Section 1.3 (“How the SAD Is Organized”) explains the information that is found in each section of the SAD. This tells you what section(s) in this SAD are most likely to contain the information you seek.

- Section 1.4 (“Stakeholder Representation”) explains the stakeholders for which the SAD has been particularly aimed. This tells you how you might use the SAD to do your job.

- Section **Error! Reference source not found.** (“**Error! Reference source not found.**”) ex plains the *viewpoints* (as defined by IEEE Standard 1471-2000) used in this SAD. For each viewpoint defined in Section **Error! Reference source not found.**, there is a corresponding view defined in Section 3 (“Views”). This tells you how the architectural information has been partitioned, and what views are most likely to contain the information you seek.

- Section 1.5 (“How a View is Documented”) explains the standard organization used to document architectural views in this SAD. This tells you what section within a view you should read in order to find the information you seek.

1.1 Document Management and Configuration Control Information

- Revision Number: 1.0
- Revision Release Date: November 24th, 2019
- Purpose of Revision: To apply and consolidate some of the concepts related to software architecture being taught in the course of Software Design

- Scope of Revision: Not Applicable

1.2 Purpose and Scope of the SAD

This SAD specifies the software architecture for **TornGest** a tournament management system that provides various functionalities regarding the registering of new players, setting up matches between players or teams, viewing and registering match results**.** All information regarding the software architecture may be found in this document, although much information is incorporated by reference to other documents.

What is software architecture? The software architecture for a system[1] is the structure or structures of that system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refers to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. This definition provides the basic litmus test for what information is included in this SAD, and what information is relegated to downstream documentation.

Elements and relationships. The software architecture first and foremost embodies information about how the elements relate to each other. This means that architecture specifically omits certain information about elements that does not pertain to their interaction. Thus, a software architecture is an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. Elements interact with each other by means of interfaces that partition details about an element into public and private parts. Software architecture is concerned with the public side of this division, and that will be documented in this SAD accordingly. On the other hand, private details of elements—details having to do solely with internal implementation—are not architectural and will not be documented in a SAD.

Multiple structures. The definition of software architecture makes it clear that systems can and do comprise more than one structure and that no one structure holds the irrefutable claim to being the architecture. The neurologist, the orthopedist, the hematologist, and the dermatologist all take a different perspective on the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these perspectives are pictured differently and have very different properties, all are inherently related; together they describe the architecture of the human body. So it is with software. Modern systems are more than complex enough to make it difficult to grasp them all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment—which *view* we are taking of the architecture. Thus, this SAD follows the principle

[1] Here, a system may refer to a system of systems.

that documenting a software architecture is a matter of documenting the relevant views and then documenting information that applies to more than one view.

For example, all non-trivial software systems are partitioned into implementation units; these units are given specific responsibilities and are the basis of work assignments for programming teams. This kind of element will comprise programs and data that software in other implementation units can call or access, and programs and data that are private. In large projects, the elements will almost certainly be subdivided for assignment to sub-teams. This is one kind of structure often used to describe a system. It is a very static structure; in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.

Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The set of processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

None of these structures alone is *the* architecture, although they all convey architectural information. The architecture consists of these structures as well as many others. This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

These structures will be represented in the views of the software architecture that are provided in Section 3.

Behavior. Although software architecture tends to focus on structural information, *behavior of each element is part of the software architecture* insofar as that behavior can be observed or discerned from the point of view of another element. This behavior is what allows elements to interact with each other, which is clearly part of the software architecture and will be documented in the SAD as such. Behavior is documented in the element catalog of each view.

1.3 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 ("Documentation Roadmap") provides information about this document and its intended audience**. It provides the roadmap and document overview. Every reader who wishes to find information relevant to the software architecture described in this document should begin by reading Section 1, which describes how the document is organized, which stakeholder viewpoints are represented, how stakeholders are expected to use it, and where information may be found. Section 1 also provides information about the views that are used by this SAD to communicate the software architecture.
- **Section 2 ("Architecture Background") explains why the architecture is what it is.** It provides a system overview, establishing the context and goals for the development. It describes the background and rationale for the software architecture. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been utilized in the architecture. It includes information about evaluation or validation performed on the architecture to provide assurance it meets its goals.
- **Section 3 (Views") and Section 4 ("Relations Among Views") specify the software architecture**. Views specify elements of software and the relationships between them.
- **Sections 5 ("Referenced Materials") and 6 ("Directory") provide reference information for the reader.** Section 5 provides look-up information for documents that are cited elsewhere in this SAD. Section 6 includes a glossary and acronym list.

1.4 Stakeholder Representation

This section provides a list of the stakeholder roles considered in the development of the architecture described by this SAD. For each, the section lists the concerns that the stakeholder has that can be addressed by the information in this SAD.

The main stakeholders to take into account during the development of the architecture are: the end user (in this particular case, the tournament manager), the developers, the code maintainers and the project manager.

The end users will have a top-level view of the workings of the system and how different components are related with each other through the implemented architecture. The main quality attributes concerning this stakeholder will be the availability of the system as it will be frequently accessed by the end user. Usability is another quality attribute to focus on, as the system should be of easy understanding for new users to decrease the learning time required.

To better understand the structuring and functioning of the code implemented between the multiple components, both developers and maintainers will be able to use the architecture representation as guidance to guarantee the quality attributes within the system or to make any amends on previous system implementations. Thus, the main quality attributes for these two stakeholders will be modifiability to ensure the system components can be easily modifiable to change implementations to adapt to environment changes, or to correct any faults that may occur during the normal functioning of the system. Understandability is another quality attribute to have in mind, not only must the components be easily modifiable, but the results of modifications or the components themselves must be easily understandable by both stakeholders. This way the time needed to dissect and work out the relations between components or the inner workings of a component itself will be decreased. Finally, we have testability, the system must be easily executable with all use case scenarios, this will make the debugging process much smoother for both developers and maintainers.

From the architecture, the project manager will be able to see the modular implementation of the system, and the relations between the components. This will make team coordination easier, as the work needed for the implementation is clearer. Knowing the relations between the components will make coordination between different teams possible, as dependencies between components will be clearly visible.

1.5 How a View is Documented

Each view is documented as follows, where the letter *i* stands for the number of the view: 1, 2, etc.:

- Section 3.i: Name of view.
- Section 3.i.1: Primary Presentation. This section shows the elements and the relations between them, using a graphical representation.
- Section 3.i.2: Element Catalog. This section details the elements shown in the primary presentation. For instance, if a diagram shows elements A, B and C, then this section will explain what A, B and C are and their purposes or roles they play.
- Section 3.i.3: Rationale. Rationale explains why the design reflected in the view came to be. The goal of this section is to explain why the design is as it is and to provide a convincing argument that it is sound. The use of a pattern or style in this view should be justified here.

2 Architecture Background

2.1 Problem Background

2.1.1 System Overview

This system is a business application, with the purpose of creating and managing sport events, it was developed using Java EE by a group of alumni in the context of the CSS subject, from the course LEI. There's a total of four basic user cases (UC) listed below. UC1 and UC2 can only be executed through a browser, while UC3 and UC4 are done using a desktop application.

Functionality

The system provides four basic use cases (UC):

- **UC1 (Register Results).** The tournament manager chooses a match from a specific tournament and registers the result of that match.
- **UC2 (View Calendar).** The tournament manager choses a player and views a calendar with all his scheduled matches.
- **UC3 (Create Tournament).** Given a modality (chosen from a list of available modalities), name of the tournament, type of tournament (played as a Team or Individual), number of participants, number of matches between a pair of teams/individuals, tournament start date and its duration in days.
- **UC4 (Register Player).** Given the participant registration number (Team or Individual's registration number) a list of available tournaments will be displayed, allowing for the manager to register the participant.

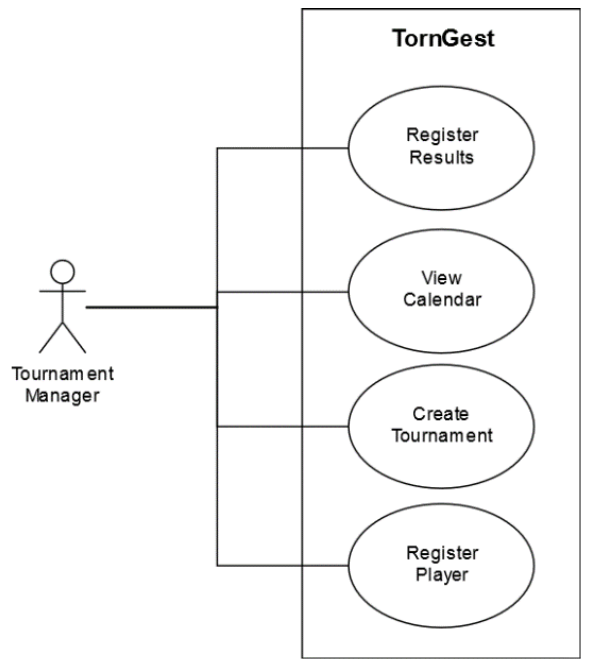

Figure 1 SSD

2.1.2 Goals and Context

The architecture defined for the system will be used mainly to aid future development and code maintainability. All the elements that make up the system itself will be documented as well as the relations among them, this will make implementing new use cases a simpler process just by consulting this SAD. It will also server as a guiding document to ensure the main quality attributes are met after each iteration of the system implementation, mainly: modifiability, maintainability, usability, readability, among others.

2.1.3 Significant Driving Requirements

The quality attribute scenarios (QAS) are listed below, separated by quality attribute.

Modifiability

- **QAS1.** In the future the developers wish to change the web page UI, to do that, they want to modify the less possible, by having the different parts of the project separated in modules, they can inflict changes on them and still have a working system.

Performance

- **QAS2.** The tournament manager views the calendar of games for a chosen player. The user must see the calendar within 5 seconds.

- **QAS3.** The tournament manager adds a player to a previously created tournament, the manager must see a notification within 3 seconds, telling him if the player was successfully added or if there was some kind of error, for example, he was already registered, or the modality that the player practices is not the same one being playing in the tournament.

Availability

- **QAS4.** The system must be available to the user 24/7, if any of the servers is down, a system administrator must be notified within 60 seconds, and the system must be repaired. The state of the system after recovering must be equal to the one before the crash, keeping the data consistent.

Usability

- **QAS5.** The system must be intuitive to the end user (tournament manager), he must be able to learn each UC in less than 30 minutes.

2.2 Solution Background – Architectural Approaches

The system implemented functions as a client-server architecture, the clients being both the Web and GUI applications. To better modularize the system's implementation and to decrease class coupling, a layered architecture was adopted. This way we can clearly separate which functionalities a component should have depending on the logical layer it will reside on, promoting modifiability.

Given the layered architecture and the implementation of the system in modules, we can not only increase modifiability, but also readability and maintainability. Having components implemented in modules makes following the control flow of the execution much easier for developers or maintainers since the same modules will be separated into different logical layers, making it also possible to discern what layer a component is residing in without looking at the architecture documentation. Code maintainability is also simplified as module dependencies are clear, making the impact of adjustments or corrections to the system predictable. This also improves testability since the relations between components are known, we can test parts of the system giving certain inputs that may test the documented component relations.

2.2.1 SonarQube Analysis Results

After running the system under a SonarQube analysis we come across various statistics that influence nonfunctional quality attributes. Regarding maintainability, the implementation does not show any major issues that may make the code unmanageable, however some minor bugs and code smells increase the technical debt to 2 days, meaning it will take that extra amount of time to correct any small issues within the code in the future. Given the statistics, we can see that most of the code smells are aggregated in the torngest-business and torngest-web-client packages, this will narrow down the problems to be solved by code maintainers or developers.

Regarding reliability, we can see that the rating is significantly lower compared to other aspects measured. Some of the issues occur in the torngest-web-client package, most of them are related to lack of attributes inside of the table headers which may make then hard to read when presented on the interface to the user. In the torngest-business package, we can see a handler class (RegistarParticipanteHandler) with high Cognitive Complexity on two different methods. This in turn will make maintaining this piece of code harder as it is more difficult to understand the control flow of the method. This issue can be solved by encapsulating portions of the logic handling, in this handler, in different methods, thus increasing readability and decreasing cognitive complexity which in turn will increase system maintainability.

The last measurement was the security of the system, this is also related the safety quality attribute. The main vulnerabilities detected by SonarQube were due to lack of correct error handling. As with most simple system implementations, error handling is taken quite lightly, a simple try-catch and printing the error message for easy debugging. However, for a system that is going to be deployed to large scale use, where there may be many clients, some of which malicious, printing the direct error stack may supply them with crucial information that may lead to system exploitation. These vulnerabilities however are easily corrected by either logging the error to a log file or by simply printing a generic error message, so the user gets some feedback.

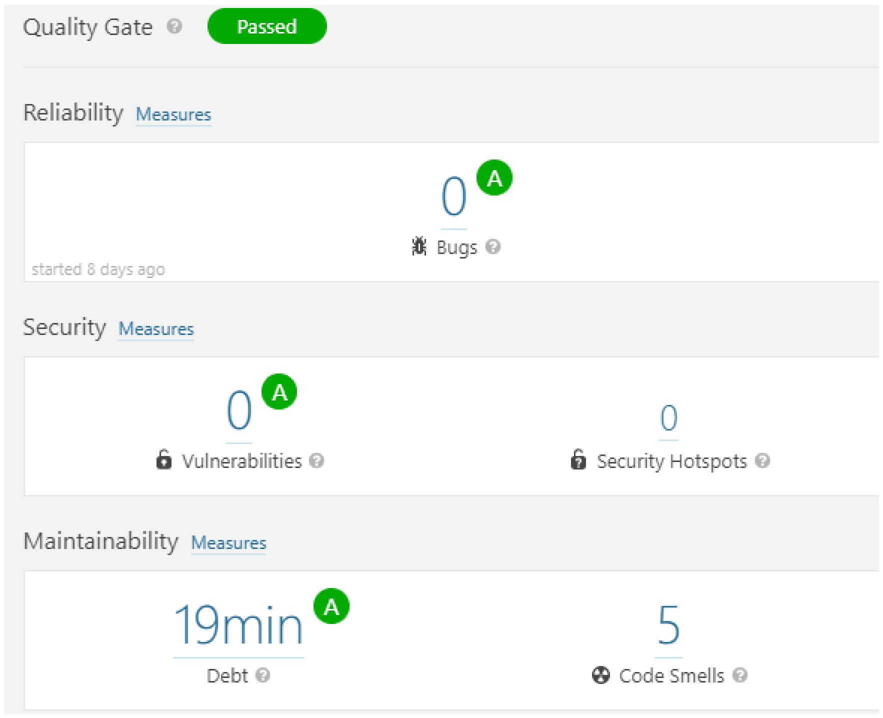

Figure 2 - SonarQube statistics post corrections

After solving some issues signaled by SonarQube, we were able to increase both system security and maintainability. Most noticeably we see a significant decrease on technical debt, having corrected some major bugs early in the development, we were able to decrease future time consumption on code correction.

3 Views

This section contains the views of the software architecture. A view is a representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. Concretely, a view shows a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.

Architectural views can be divided into three groups, depending on the broad nature of the elements they show. These are:

- Module views. Here, the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. Modules are assigned areas of functional responsibility, and are assigned to teams for implementation. There is less emphasis on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as: What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use? What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

- Component-and-connector views. Here, the elements are runtime components (which are principal units of computation) and connectors (which are the communication vehicles among components). Component and connector structures help answer questions such as: What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes?

- Allocation views. These views show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation structures answer questions such as: What processor does each software element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of the software element to development teams?

These three kinds of structures correspond to the three broad kinds of decisions that architectural design involves:

- How is the system to be structured as a set of code units (modules)?

- How is the system to be structured as a set of elements that have run-time behavior (components) and interactions (connectors)?

- How is the system to relate to non-software structures in its environment (such as CPUs, file systems, networks, development teams, etc.)?

Often, a view shows information from more than one of these categories. However, unless chosen carefully, the information in such a hybrid view can be confusing and not well understood.

3.1 TornGest Module Layered View

3.1.1 Primary Presentation

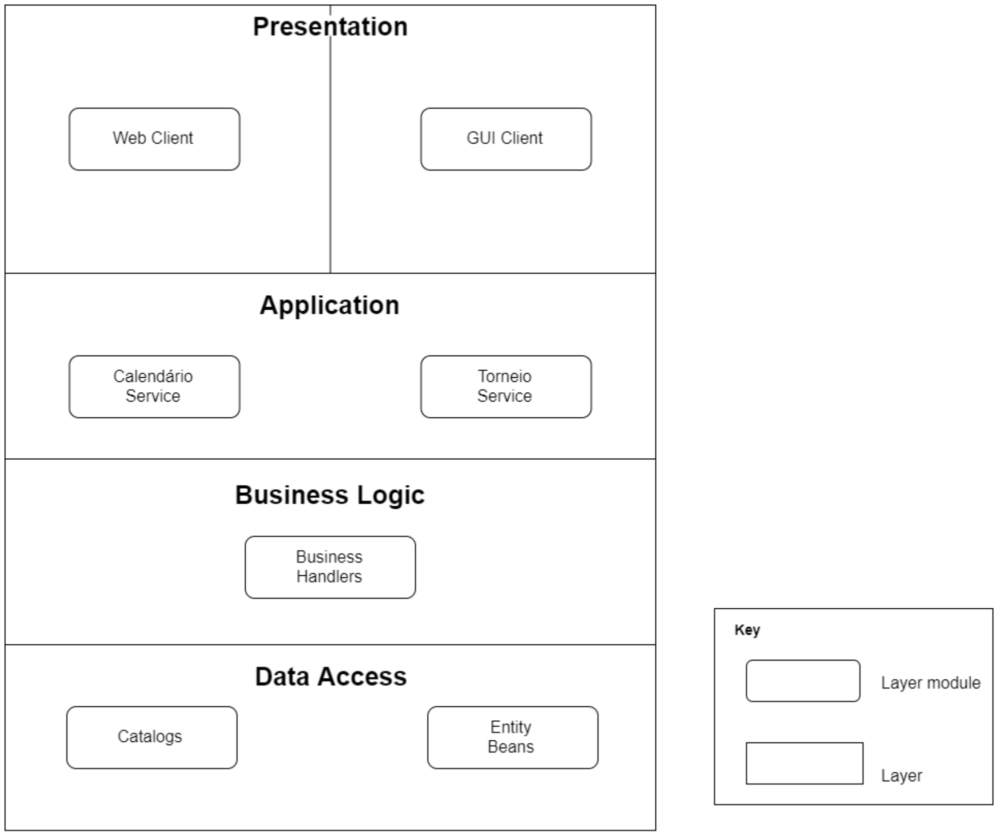

Figure 3 - TornGest Module Layered View

3.1.2 Element Catalog

- The presentation layer will be composed of 2 clients, a web client and a desktop GUI client. Each will accommodate 2 of the 4 total use cases to be implemented in our system. Both clients will have access to the business layer from which all the data will be retrieved and sent to.

- The application layer will expose the interfaces of 2 main services from which the main methods will be called from. These services will have access to the business logic layer which will take care of all the logical operations given the input from the services with handler classes.

- Each handler class will have access to a set of database entity catalogs from which all data enquiries will be done.

- To access the database, the entity catalogs will use JPQL Queries which are defined within the database entity classes. The entities themselves will define the tables in the remote database, i.e. the columns and their respective values.

3.1.3 Rationale

This view was defined to aid all the stakeholders in question as it provides a very high-level view of the system implementation. Maintainers and developers will be able to extract the general control flow of the program, making development or debugging simpler. Each layer has a certain functionality that no other layers should have, making it clear where a certain module should be implemented.

3.2 TornGest Module Decomposition View

3.2.1 Primary Presentation

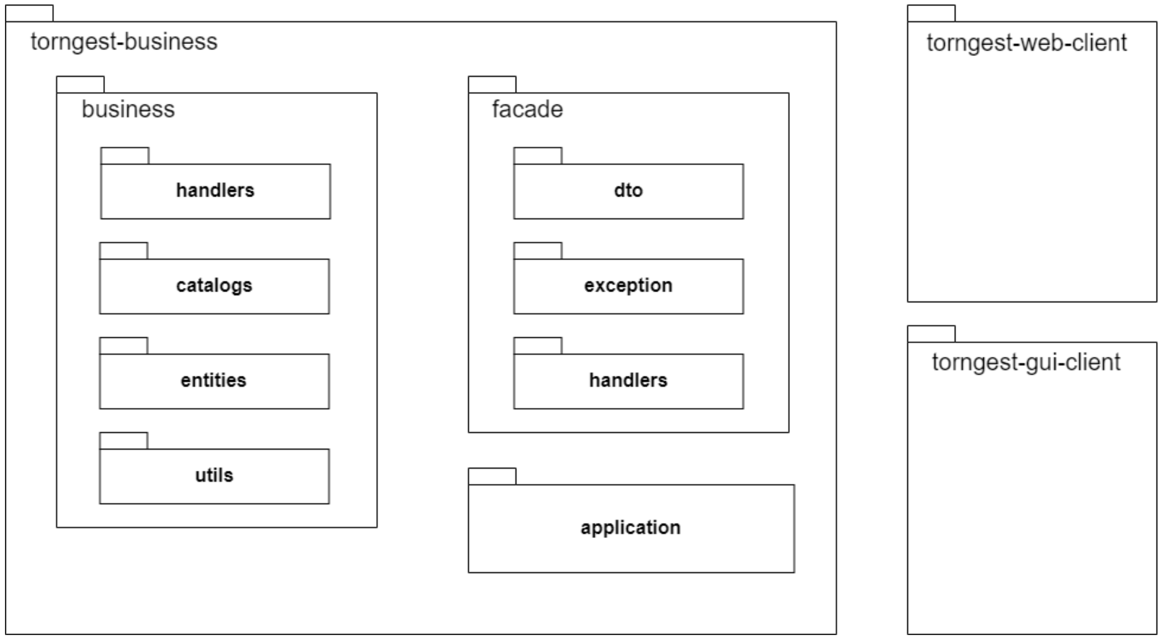

Figure 4 - TornGest Module Decomposition View

3.2.2 Element Catalog

- torngest-business – This module contains all the business logic necessary for the 4 use cases implemented in the system

- business – this package contains the main processing logic required for any type of input from the user

- handlers – the handlers package contains a set of handlers which will take care of the information treating required for every use case implemented. These will have access to entity catalogs from which data will be retrieved and will also have methods to persist data in the database.

- entities – the entities package contains all the database entities which define the columns and their respective values. Also defined within the classes of this package are JPQL queries used to access data from the database.

- utils – this package contains a helper class used to calculate the elo after a match

- facade – this package contains the main implementation of the façade design pattern, mainly used to mask the underlying business logic complexity

- dto – this package contains all the DTOs (Data Transfer Objects) required to send requested data through method calls on both the GUI and Web clients

- exception – the exceptions package contains custom exceptions which are thrown according to the business logic implementation

- application – this package contains the implementation of both main services required for the correct functioning of the system. The CalendarioService which will deal with all date viewing functionalities and TorneioService which will deal with viewing set matches, creating new matches between players and adding new players to the database

- torngest-web-client – This module contains the web client accessed through a browser which will let the user, store the result of a given match and view the upcoming matches of a give player or team

- torngest-gui-client – This module contains the GUI interface which will access the business logic through RMI calls

3.2.3 Rationale

This view promotes modifiability and maintainability. By decomposing the main modules in smaller modules, we have smaller more concise units of implementation, this helps to lower coupling between classes therefore boosting modifiability. It helps futures developers and maintainers to reassess the way the modules were implemented helping them to either integrate new functorialities the system may need, or to correct already existing bugs.

3.3 TornGest Module Uses View

3.3.1 Primary Presentation

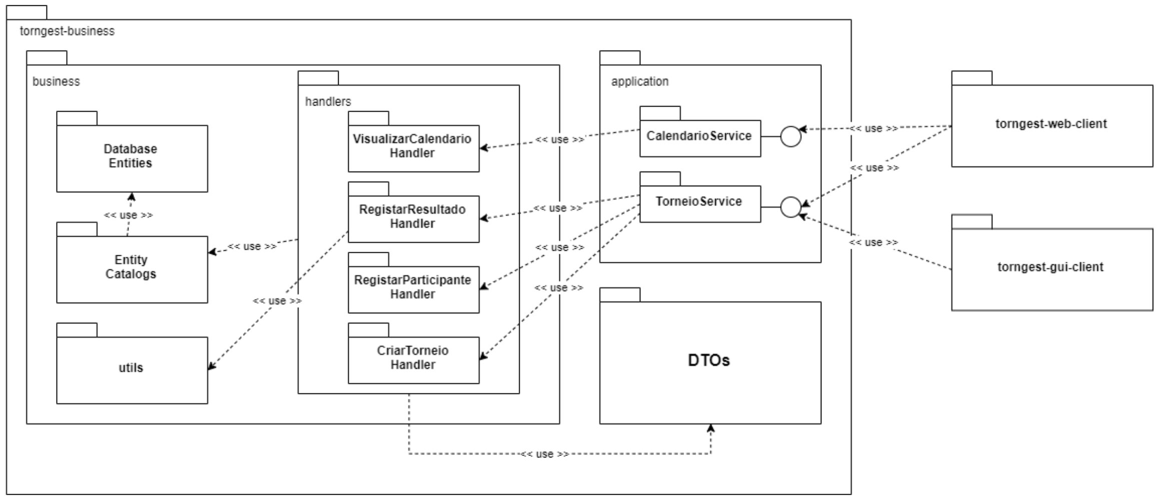

Figure 5 - TornGest Module Uses View

3.3.2 Element Catalog

- torngest-business – this module contains all the classes related to business logic handling give the input from the available clients. It implements 4 main use cases:
    - Create tournament
    - Register tournament participant
    - Register tournament scores
    - View participant schedule
- business – this module contains the main classes for all the logic handling necessary
- Database Entities – this package contains all the entities that define the table within the MySQL database, i.e. their columns and respective values. These classes also have a set of JPQL queries which can be used to add and retrieve data from the database.
- Entity Catalogs – this package contains all the entity catalogs following the information expert design patter. Each catalog will be responsible for accessing a certain entity from the database given any restrictions necessary. Database access will be done in the form of TypedQueries, granting safe access by disabling SQL injections.
- handlers – this package contains the 4 handlers which implement each one of the use cases defined. The handlers will do most of the logic processing and value checking before getting data from or sending data to the database through the catalogs.
- utils – this package contains a simple helper class which calculates the score of a given tournament, it is only used by the RegistaResultado Handler.
- Application – this module contains the main services accessed by both the GUI and Web clients. These services will expose all the functions implemented in the system via a remote interface which will be injected to both clients.

3.3.3 Rationale

This view serves as an aid mainly to developers, but also to code maintainers and project managers. It shows a more in-depth view of the modules implemented and the dependencies between each other. These dependencies are to be considered in future iterations of system implementation or for debugging purposes. By seeing which modules depend on which, we can see clearly the general control flow of the system, increasing the testability.

3.4 TornGest Entities Module Class View

3.4.1 Primary Presentation

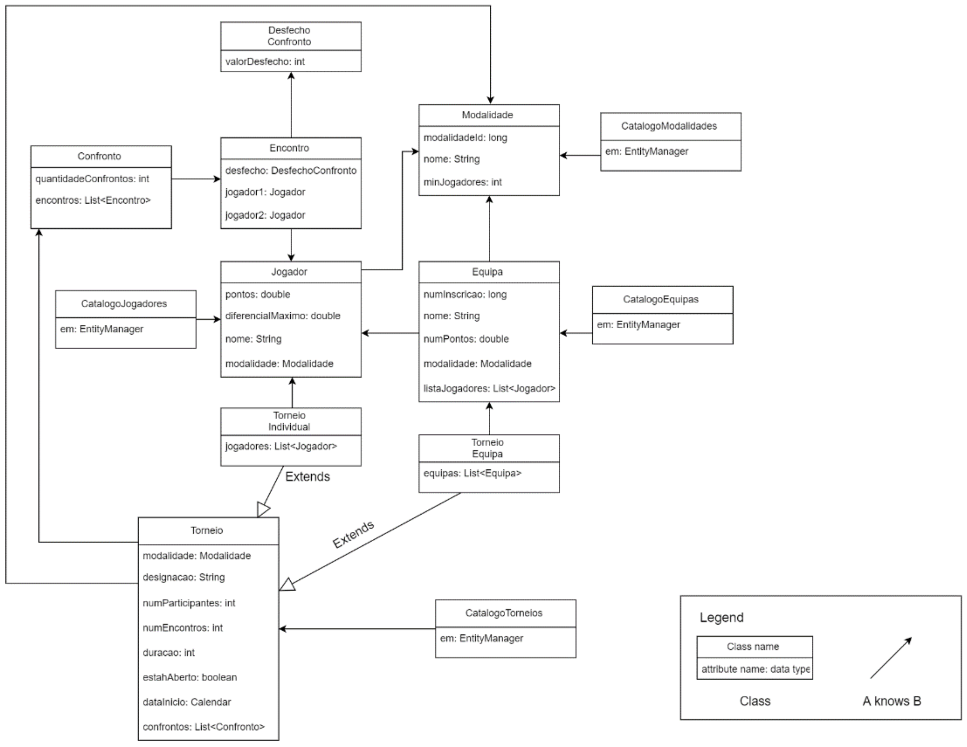

Figure 6 - TornGest Entities Module Class View

3.4.2 Element Catalog

All the classes represented above are entity classes, except for the Catalogo ones (that use TypedQuery to query the database to look for data hold by the entities). Let's start with the entity Torneio class, represents a tournament, since this system supports matches between teams and individuals, there are two kinds of tournaments, TorneioEquipa(keeps a list of registered teams) and TorneioIndividual(keeps a list of registered individuals) respectively, this two classes extend the

Torneio class. Each instance of Torneio has a Modality that will be played and a list of Confronto. Confronto represents all the “matches” between every intervenient in the tournament, it keeps a list of Encontro “matches”, each Encontro instance keeps track the two players that will face each other, and a DesfechoEncontro, an enumerator that has the three possible values for the outcome of the “match” (vitoria, empate and derrota).

An individual is represented only by the Jogador class, each Jogador has one Modalidade that he plays, while the Teams are represented by various players, the class Equipa represents a team, and it has a list of Jogador that belong to that team, and like the Jogador class, also has a Modalidade played by that team.

3.4.3 Rationale

The Domain Model was used because it deals better with increasing complexity, the pattern provides an object-oriented way of dealing with complicated logic. Instead of having one procedure that handles all business logic for a user action there are multiple objects and each of them handles a slice of domain logic that is relevant to it. The Data Mapper JPA was also used to persist the data, this patter is advantageous because the fact that domain objects don't need to know that there is a database present, making it easy to introduce changes to database structure or domain logic.

3.5 Web Client multi-tier C&C View

3.5.1 Primary Presentation

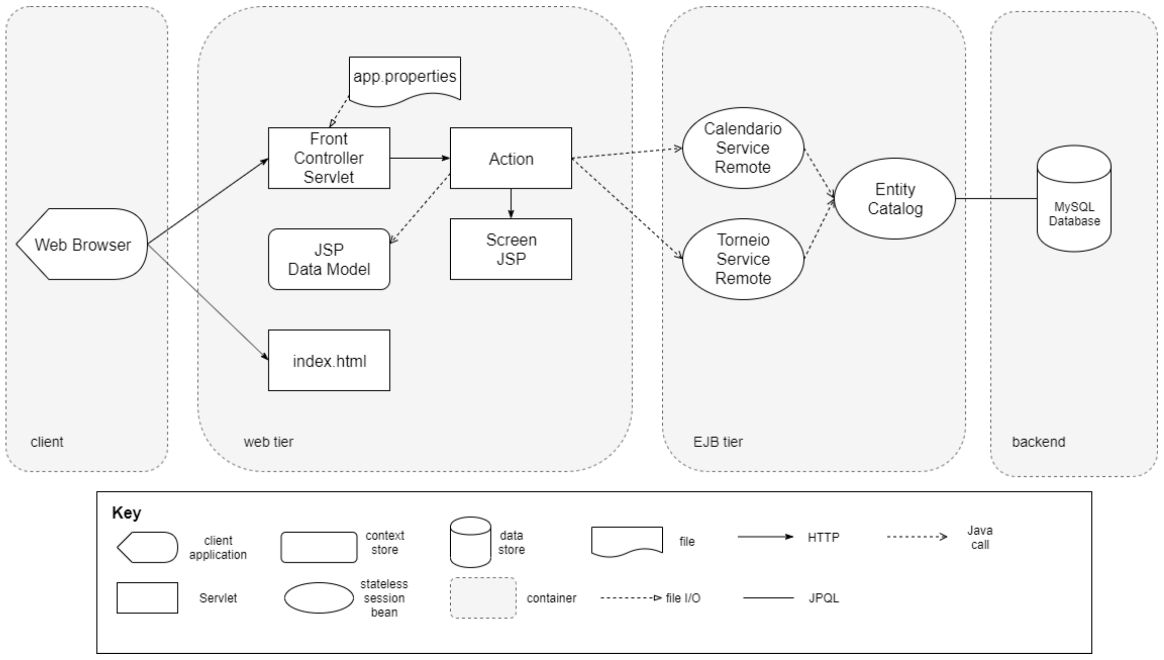

Figure 7 - Web Client multi-tier C&C View

3.5.2 Element Catalog

- Web Browser – the application used by the user to serve as a client to the web server

- Front Controller Servlet – this servlet is responsible for mapping all HTTP requests from the web application to their respective action object, from which the input or output data will be handled and displayed. For every action there is a path defined within app.properties file, which is loaded into the servlet upon initialization, thus mapping all action objects to their corresponding paths. Any action that is not mapped by the servlet will be categorized as an unknown action. When the application is first accessed, the app.properties file will be loaded into the front controller's map, routing all the URLs to a specific screen JSP.

- app.properties – this file contains the path mappings to all the available actions for all the use cases implemented in the web client.

- index.html – this file displays the initial welcome page. It contains 2 URLs which are associated to use cases view player schedule and store match results.

- Action – this component handles the contents of incoming http requests from the web client. The data to and from the database will be sent and retrieved by a specific action depending on the use case in question. It will have access to the 2 main services from within the business logic, which are injected as stateless EJBs. It will also be responsible to forwarding the HTTP request to the corresponding JSP file. When the client presses a button to conclude input insertion, the request will be sent to the front controller which will map the URL to the respective Action component.

- Screen JSP – these JSP files are part of the web screen. The data required to fill in the fields, if any, is sent via HTTP requests from the action servlets. These requests will contain the data model which contain all the fields necessary for the current JSP, whether it be displaying data, or filling in data to be sent to the backend for further request processing.

- JSP Data Model – this component will be used by a specific screen JSP. It contains all the fields that may be displayed on the loading of the JSP and the fields that will be filled in through the screen JSP, by calling the appropriate action when submitting the data. After filling in the necessary values, upon clicking on a submission button, a JSP Data Model will be populated with the respective values filled in. This model is passed on to the Action component responsible for all the data handling necessary.

- Calendario Service Remote – this component is a remote stateless session bean. It will be accessed by a specific action that takes care of the view player schedule use case. The action component calls methods on this component via Java calls, with the values extracted directly from the Data Models or computed from them.

- Torneio Service Remote – this component is also a remote stateless session bean accessed for the register match result use case, both remotes are injected in the actions and invoked through java calls. It functions the same way as Calendario Service Remote through Java calls by the Action component.

- Entity Catalog – this component will take care of all the database accesses needed through JPQL queries. Whenever a user requests information from the database, these components will handle all the query setup necessary.

- MySQL Database – this component represents the main datastore from which all the data will be accessed from and stored to.

3.5.3 Rationale

This view is useful for both implementation and deployment purposes. The end user can access this view to see what is accessed when using the web application, giving some more in depth information about the inner workings of the system. This may increase system usability as the user acquires more information. It also boosts system maintainability, by seeing which components interact with each other and how they do it during runtime eases the testing and debugging process. For instance, if the screen is displaying wrong date regarding the upcoming matches of a certain player, then we know that either the Action component is not processing the input data correctly, or the session beans that conduct business logic are not validating the data correctly before retrieving information from the database.

This view also boosts modifiability, the developer can extract from this view information regarding the components and connections to access the web application to, for example, register a player into a tournament, and may want to add additional functionalities to that use case, or add a new use case all together following the flow control of the components.

3.6 GUI multi-tier C&C View

3.6.1 Primary Presentation

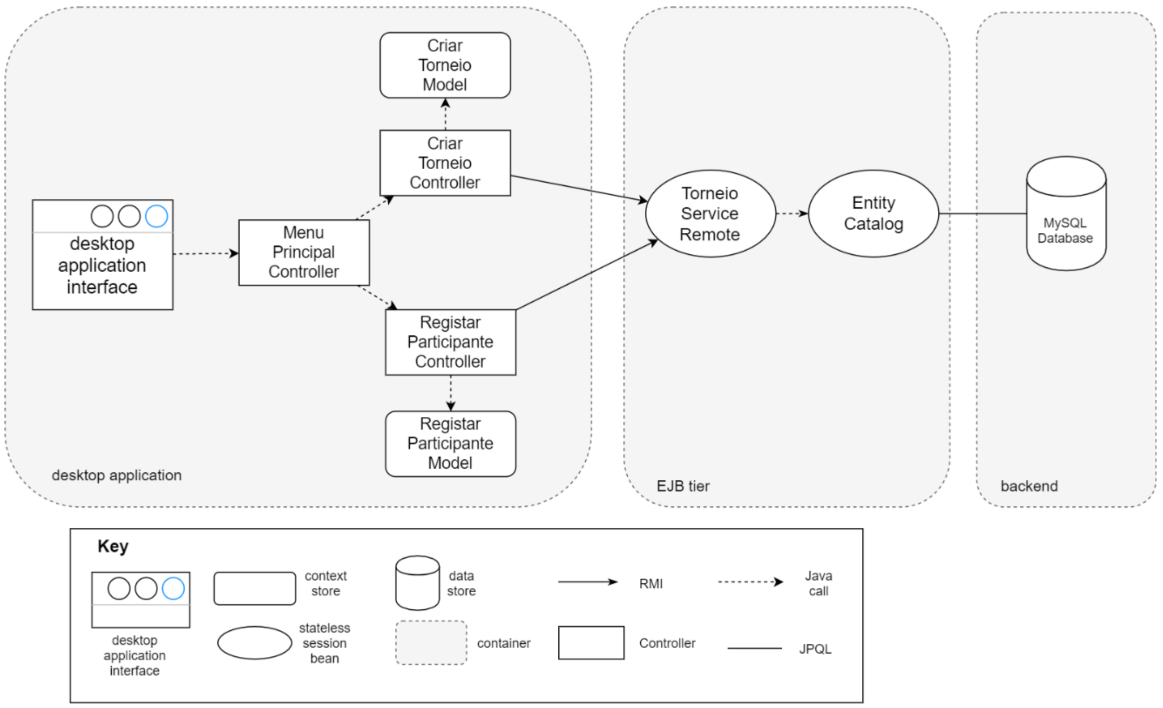

Figure 8 - GUI multi-tier C&C View

3.6.2 Element Catalog

- The Desktop application interface follows the MVC (Model View Controller) design pattern, each page that it is seen in the interface is controlled by a different controller

- MenuPrincipalController - The controller that is first loaded when the application is booted, from this scene we can chose to perform one of the two implemented UC, which will be handled by a different controller
- CriarTorneioController - The controller responsible for UC3 (Create Tournament), it has control of the CriarTorneioModel.
- RegistarParticipanteController - The controller responsible for UC4 (Register Player), it has control of the RegistarParticipanteModel.
- Model - Each model is responsible for holding user input information, that will be then used by the respective Controller.
- Torneio Service Remote – this component is a remote stateless session bean accessed by the controllers using an interface, both controllers use it to perform the user case
- Entity Catalog – this component will take care of all the database accesses needed through JPQL queries. Whenever a user requests or stores information to the database, the transaction will always be processed by an entity catalog.
- MySQL Database – this component represents the main datastore from which all the data regarding match scores from the web client will be stored.

3.6.3 Rationale

For the desktop GUI, the MVC (Model View Controller) pattern was used, this way the partition of duties helps the developer in future developments. It has low coupling behavior among the models, views, and controllers. This allows developers to work simultaneously on the interface, reducing the amount of conflict changes, increasing the productivity of the work.

3.7 TornGest Deployment Allocation View

3.7.1 Primary Presentation

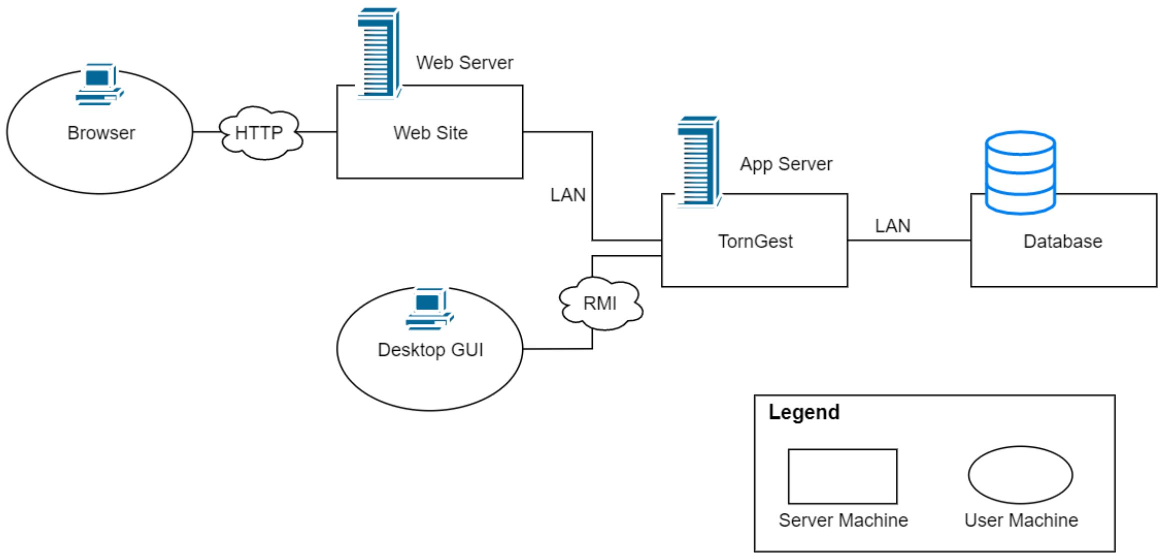

Figure 9 - TornGest Deployment Allocation View

3.7.2 Element Catalog

The user has two ways of interacting with TornGest Application, the first one is through a web browser, that serves as a client to the Web Server, they communicate using HTTP, the WebServer then forwards those requests trough a LAN connection to the App Server. The other way is by having a desktop GUI installed on its own machine, that communicates directly to the app server using RMI. The App server is connected via LAN to another server, that is responsible for persisting the data.

3.7.3 Rationale

Scalability

The system has horizontal scalability, it uses three different servers, one running the web server, one for the app server and another one for the database. This way we don't overload the same server with huge amount of work, it copes better with higher traffic than having a single server machine hosting all the servers. Also, the user has two ways to interact with the system, through a browser and a desktop GUI, this way, if the web server is down for some reason (maintenance), the user can still use the system.

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective and design handle on a system, and each is valid and useful. Although the views give different system perspectives, they are not independent. Elements of one view will be related to elements of other views, and we need to reason about these relations. For example, a module in a decomposition view may be manifested as one, part of one, or several components in one of the component-and-connector views, reflecting its runtime alter-ego. In general, mappings between views are many to many. Section 4 describes the relations that exist among the views given in Section 3. As required by ANSI/IEEE 1471-2000, it also describes any known inconsistencies among the views.

4.1 General Relations Among Views

Deployment View	**Module View Decomposition**
element: Web Site	module: torngest-web-client
element: TornGest	module: torngest-business
element: Desktop GUI	module: torngest-gui-client
element: Browser	N/A
element: Database	N/A

Table 1 - Relation between Deployment and Decomposition

A direct mapping between the deployment allocation view and module decomposition view is useful during implementation of new functionalities. By knowing where a certain piece of the software will be run, it is possible to implement in a way that best fits its environment. For example, if we wanted to add new functionalities to the desktop GUI, we would have to take into account that the application will be run on the clients personal computer, meaning that resource consumption and performance of the implementation should be taken into account.

5 Referenced Materials

Barbacci 2003	Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs)*, Third Edition (CMU/SEI-2003-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr016.html>.
Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice,* second edition, Addison Wesley Longman, 2003.
Clements 2001	Clements, Kazman, Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley Longman, 2001.
Clements 2002	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.

6 Directory

6.1 Glossary

Term	Definition
software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
view	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
view packet	The smallest package of architectural documentation that could usefully be given to a stakeholder. The documentation of a view is composed of one or more view packets.
viewpoint	A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and audience for a view, and the techniques for its creation and analysis [IEEE 1471]. Identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques,

	consistency checking techniques, etc., used by any conforming view.