Universidade Federal do Pampa

Research Project

Atlas

Marcelo Schmitt Laser
Fernando Lima
Luciano Marchezan

supervised by
Dr. Elder M Rodrigues

April 11, 2017

Introduction

This document will be used to keep track of the Atlas project, a Feature Modeling tool with two primary purposes that both drive its design and set it apart from the existing technology. These purposes are a) to be used in an educational setting as a tool for teaching basic Software Product Line (SPL) concepts and b) to be used as a distributed research tool in maintaining a repository of publicly-accessible feature models.

The purpose of being applied in an educational setting is born from the authors' experience in having great difficulty identifying simple, notation-unbound tools for the explanation of the purposes and functionality of feature models. The existing tools are for the most part bound by particular notations or development environments, and often require download and installation to function, which while usually platform-independent, poses an extra step against the incentives of teaching new SPL notions to inexperienced and possibly unmotivated students.

In regards to its use as a repository, Atlas draws much inspiration from S.P.L.O.T. [8] in attempting to make available a vast number of example feature models, previously validated and analysed, for purposes of serving as proofs of concept or reference specifications. The authors hope to make Atlas a centralizing reference for the education of multiple feature modeling notations through a primarily pedagogical public repository. Furthermore, the authors' experience in distributed research teams drives the need for an account-based private repository to be used in sharing models between contributors, enabling the editing and visualization of models by multiple individuals in geographically separate locations without the need for constant transfer of data, such as through e-mail attachments or cloud-based repositories like Google Drive or Dropbox.

The Atlas Project is part of the overarching Hestia Project, which proposes a pedagogical and research-oriented framework for the dissemination and development of SPL practices. Atlas is the first tool to be implemented in the author's plan of Olympus, a platform to support Hestia as both a Software Product Line (SPL) and Component-Based Software Engineering (CBSE) architectural framework.

The first installments of Atlas are to be developed by a team of undergraduate students from UNIPAMPA (Brazil) who are participating in a research internship programme, and will therefore serve as an experiment for the development of the Hestia framework itself, which is directed towards distributed,

volatile and inexperienced teams. The project experiences will be documented so as to permit later analysis and to serve as input to the main Hestia Project.

We begin this document by describing the context within which this project was conceived and the objectives it hopes to achieve (Chapter 1). We then proceed to elicit the requirements of the system, based on the context and objectives already raised and with the notion in mind of Atlas serving as a subsystem of the Olympus Project (Chapter 2).

Following this, we present the summary of our studies on Feature Model notations, presenting the notations selected for representation on Atlas and the algorithms to enable conversion between notations based on their commonalities (Chapter 3). We then establish the architecture for Atlas, going through the selected technology used for the implementation of the project, the architectural styles and patterns used, the structural and behavioural descriptions of the system, and finally the rationale behind the decisions made [11] (Chapter 4).

Finally, we present the implemented tool, with a focus on its descriptive architecture and functionality, which is analysed based on the testing logs derived from the project's test cases (Chapter 5). At the end of the document, the lessons learned from this project are listed alongside the conclusions drawn from the experience and which will be passed forward into the Hestia Project (Chapter 6).

Contents

Chapter 1

Background

The Atlas project, within the context of the Hestia macro-project, was born from the experiences of our research group over a few years studying Software Product Lines (SPL). It first appeared as the desire from the group to create a single tool that would enable it to more smoothly manage what it perceived as stages of design and development of an SPL, in order to serve as an auxiliary environment in the development of other research projects [2] [3].

The ideas behind this tool quickly grew to a proportion that far exceeded the initial design decisions of the research group, first through a proposal of a software architecture that exceeded the original tool's capabilities [6] and then through the realizations that a framework could be constructed to aid in the several stages of SPL design and development which drew from our research and educational experience, in order not only to aid in particular tasks but to extend the adoption of SPL practices in general [7] [5].

Atlas is conceived as a solution for the particular research scenario lived by this group, and which the authors believe is a common scenario for research groups around the world. This scenario is characterized by contributors distributed in different geographical locations, with different schedules and working hours, with development teams that are inexperienced and volatile (in the sense that contributor turnaround is high) and limited financial resources. The requirements and design decisions of Atlas are representative of this particular scenario, and are aimed at solving the problems encountered by our group over the years. It is our firm belief that we are not alone in these constraints, and that the adoption of SPL practices would greatly benefit from a solution thus directed.

It is important to note that Atlas is meant as only one subsystem within the Olympus integrated environment, and is therefore designed with inter-system interactions in mind. While the authors believe that the contributions of this tool have merit in their own right, the requirements and design decisions taken throughout this document illustrate the objective of later integrating Atlas to other tools within the larger scope of the Hestia macro-project.

The following section lists the overall objectives of both the Atlas project

and the Atlas tool, in order to serve as a foundation for the decisions taken throughout this project.

1.1 Objectives

In accordance to the context given above, the objectives of the Atlas tool can be listed as follows:

- To provide a feature modeling environment that will be ideal for educational settings in which students may have little to no experience with SPL practices and may be initially unmotivated regarding the subject;
- To provide a feature model repository that will serve as a database of prevalidated example feature models in various notations, presenting various examples of the commonalities and peculiarities of each supported notation;
- To provide a feature modeling environment that is accessible to inexperienced users, possibly from fields other than software engineering or computer science in general, therefore furthering the adoption of SPL practices by a wider group of practitioners.

The following list presents the objectives of the Atlas project, within the context of the Hestia macro-project:

- To examine the needs of geographically distributed research groups in regards to sharing and labeling of data related to feature models, in order to further explore solutions for geographically distributed SPL research in particular, and computer science research in general;
- To evaluate the perceived architectural practices in both design and implementation that will serve as the basis for the definition of the Hestia architectural style and its related patterns, in order to refine the ongoing research in this subject;
- To research and explore the fields of Human-Computer Interaction and Graphical User Interfaces, invaluable to the success of any Integrated Development Environment (IDE) or Computer-aided Software Engineering (CASE) tool;
- To establish the basis of the Olympus integrated environment, experiencing the architectural peculiarities of distributed and GUI-intensive web systems.

Chapter 2

Requirements and Design Decisions

Building on the objectives presented in Chapter 1, this chapter presents the requirements elicited and design decisions taken in the Atlas project. As with any architecture-centric software project, it is important to note that the requirements and design decisions are subject to change over the course of the project [12].

2.1 Solution Requirements

The requirements, as of December 9th, 2016, in accordance with the currently established goals and with a strong basis on the experiences drawn from the first instance of Atlas [7], are as follows:

- RQ01: The proposed solution must be Web-based, so as to be readily accessible in different environments without the need for installation or preparation of the environment;
- RQ02: The proposed solution must make use exclusively of open-source and free technology, so as to be costless as an education solution, given the possible lack of resources that a student or institution may have to dispose of in this field;
- RQ03: The proposed solution must present an user interface that requires little effort to understand. This effort will be measured with qualitative evaluations and further research of Human-Computer Interaction standards;
- RQ04: The proposed solution must be extensible to allow for the support of new feature model notations;

- RQ05: The proposed solution must be extensible to allow for the support of feature model notation conversions, based on the commonalities and peculiarities of different notations, in order for users to more easily visualize these commonalities and peculiarities and identify the relationships between different notations;
- RQ06: The architecture of the proposed solution must follow well-defined architectural styles and patterns, so as to have an easily accessible documentation to anyone with knowledge of software architecture;
- RQ07: The proposed solution must dispose of a feature model validation system, linked to each of the available notations, in order to permit the rapid identification of flaws and assist in self-teaching scenarios;
- RQ08: The proposed solution must include a data persistence system to maintain a repository of feature models accessible to the public;
- RQ09: There must exist a set of publicly available tutorials and a manual to the tool, as well as publicly available architectural documentation and code, so as to disseminate its use and facilitate its adoption.

The requirements so far listed can be taken as general requirements, believed by the authors to be the basic foundation to any feature modeling tool that is intended as educational. In addition to these, the following requirements are raised specifically for Atlas, in order to prepare it for later assimilation within the Olympus environment:

- RQ10: The proposed solution must be designed in a modular manner, with well-specified interfaces and communication protocols, in order to be easily adaptable to operate within larger software environments;
- RQ11: The proposed solution must function independently, being possible to instantiate and operate it without the need for other, third-party tools.

2.2 Project Design Decisions

The previous section presented the elicited requirements that the authors believe are essential both to a general feature modeling educational tool and to the specific needs of a tool integrated within a larger software environment. This section presents the design decisions specific to this project, which will be taken in the design and implementation of Atlas to meet the presented requirements.

Chapter 3

Feature Modeling Notations

This chapter will present the Feature Modelling notations studied by this project, including their common characteristics and peculiarities. While Atlas' architecture is being designed to be independent of notations and to be easily extensible by external contributors, the authors will maintain a "master" version of the tool. This chapter presents the notations that will be supported by the authors' version, including the algorithms that are used to convert between notations. The names of the notations are given based on the generally-used naming conventions found in the literature, and are often not explicitly given within the original authors' description.

3.1 Feature-Oriented Domain Analysis - FODA

3.2 Generative Programming - GP

3.3 Cardinality-Based - CB

3.4 Gurp-Bosch-Svahnberg - GBS

3.5 FeatuRSEB

Chapter 4

Architecture

This chapter will define the architecture of the tool being proposed. It will begin by describing the technology to be utilised in its implementation, so as to narrow down the possible architectural styles and patterns that can be applied in its construction.

4.1 Technology

In order to properly define Atlas' architecture, it is necessary to first study the technology that is available for its implementation, as this should narrow down the options of architectural styles and patterns that can be applied [12]. This section presents the areas of technology that must be studied in order to identify the appropriate tools, languages and other instruments to be used in Atlas' implementation.

4.1.1 Diagramming Libraries

The primary function of Atlas is that of permitting the graphical diagramming of Feature Models within a Web-based environment. In order to minimise effort, as well as standing in accordance with the precepts of Component-Based Software Engineering (CBSE) [4] [1], it is necessary to identify the currently available libraries that may suit this particular function.

In order to facilitate the research into this subject, certain constraints are established upon this requirement, as follows:

- The library's output must be HTTP-compatible, so as to be usable within a Web-environment;
- The library must be free for use, as per established requirements;
- The library must have sufficient source material to permit its use without trial-and-error implementation;

- The library should be proven to work on all modern Web Browsers (Mozilla Firefox, Google Chrome, Internet Explorer, Safari).

4.1.2 Client-side Programming Language - CSPL

Atlas will require use of a programming language engineered for Web environments, so as to permit easy parsing of its client-side service with existing Web Browser technologies. The choices of language will be largely limited by the choices of Diagramming Libraries studied, and should be researched in conjunction with them. The choice of client-side programming language is constrained in the same manner as the Diagramming Libraries.

4.1.3 Database Management System - DBMS

One of the greatest contributions of the Atlas project is the notion of a free repository of feature models, as well as private feature model repositories per account, should this functionality be passed by project management. For this to be possible, a Database Management System must be selected to persist and provide access to the models.

There is only one constraint upon the DBMS choice, and that is that the DBMS permit an equivalent representation to those used in the Server-side Programming Language, preferably object-oriented collections or entity-relationship models.

4.1.4 Server-side Programming Language - SSPL

The server-side of Atlas will be largely responsible for the serialising and deserialising of persisted feature models, as well as offering validation of feature models, executing the transformation between notations and controlling access permissions to the repository database. Therefore, the research of this subject should be constrained by certain premises, as follows:

- The language should be capable of processing large data structures, as certain feature models may exceed the tens of thousands of elements and will need to be validated in reasonable time;
- It should be possible to program data structures in an object-oriented form, or some similar form that permits reuse of data structure elements;
- The language must be capable of accessing the DBMS selected for the repository;
- The language must be capable of providing a Web server to communicate through HTTP packets with the Client-side service.

4.1.5 Web Server

A Web server will need to be installed and configured to provide the Server-side services of Atlas. This Web server must be capable of handling both the SSPL and DBMS choices, as it is likely that we will not avail of the resources necessary to establish two Web servers, and a single one will most likely have to handle the server-side services and persistence of the system. The Web servers researched must also be capable of running on either a Linux or Windows platform, as these are more widely available and therefore of easier maintenance and evolution.

4.1.6 Integrated Development Environment - IDE

For the better use of human resources in the project, one or two IDE will have to be selected for use between all development team members. The choice of IDE will evidently be limited by the choice of CSPL and SSPL, but it is not required that a single IDE be used for both server-side and client-side development. Whether it be a single one, or two different ones, the IDE responsible for the client-side development must be capable of handling HTML and CSS markup, so as to centralize the development of the application services of the client-side.

4.2 Architectural Style

This section will present the architectural styles selected for use in the design of Atlas' architecture.

4.2.1 Client-Server

Due to the distributed nature of both Atlas and the overarching Hestia project, it is clear that the primary architectural style used should be the Client-Server style. The further study of this style is necessary for the full design of Atlas' architecture.

4.2.2 Blackboard

At the current time, December 10th, 2016, the authors believe it prudent to assume that a Blackboard-like architectural style be adequate for the implementation of the GUI of the client-side application. Further study of this style, as well as selection of a diagramming technology to be used in the implementation, should clarify whether or not this style is indeed adequate for this application.

4.3 Architecture Views

This section presents the views of the Atlas architecture, the relations between them and the specifications of their elements. These views are at the time presented primarily in UML diagrams [9], along with certain semantically-loose freehand diagrams.

4.3.1 Overview

The overview of Atlas serves to present the highest-level concepts of the system's architecture, including the form of software deployment and the high-level interfaces of each high-level component.

High-level Deployment Diagram

The basic elements of the Overview can be divided between the Client and Server components, as presented in Figure 4.1. The Client is responsible solely for maintaining its own model of the diagram being modeled by the user, as well as the GUI functions, separated between the presentation itself, a Renderer component and a State-Machine component. The Server is responsible for maintaining a persistent repository of models and validating models provided by the Clients.

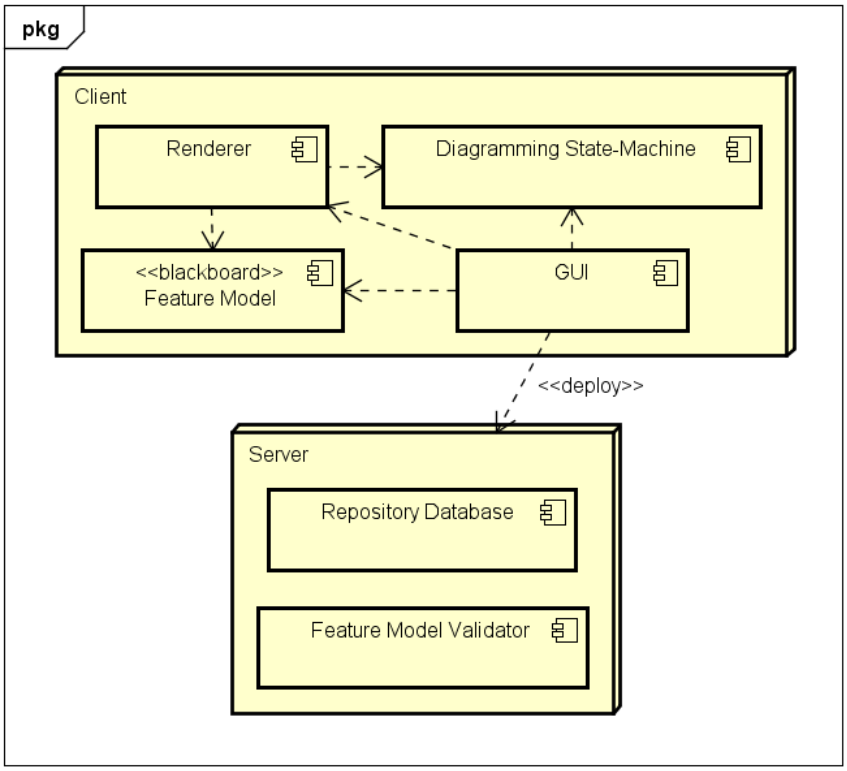

Figure 4.1: High-level Deployment Diagram

High-level Client Component Diagram

The client can be further defined by establishing the interfaces provided and consumed by its components, as well as by defining the GUI as a subsystem and establishing its inner parts. This is presented in Figure 4.2. As can be seen, the GUI subsystem redirects all of its parts' requests to the appropriate interfaces, encapsulating the Model Visualizer, Diagramming Toolbar and System Interface components. Furthermore, both the Renderer and the GUI access the Feature Model through the same *Façade* component; likewise the Diagramming State-Machine. This enables these components (Feature Model; Diagramming State-Machine) to provide services independently of application, expanding their reusability.

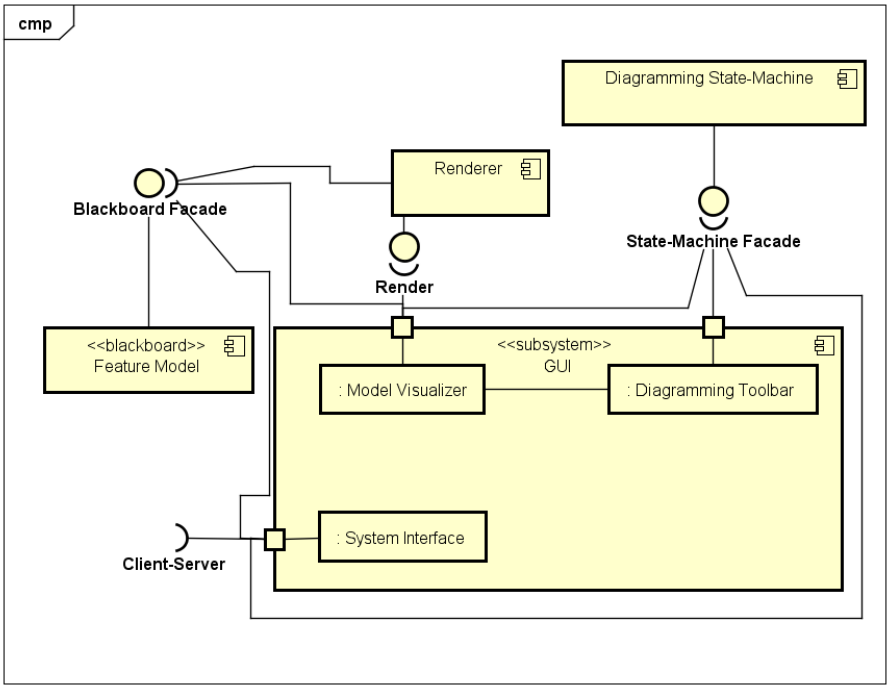

Figure 4.2: High-level Client Component Diagram

High-level Server Component Diagram

The server is not as well-defined as the client, presenting a simple Linkage connector that permits the Client to access it through a single entry-point for both of its functions. This is presented in Figure 4.3.

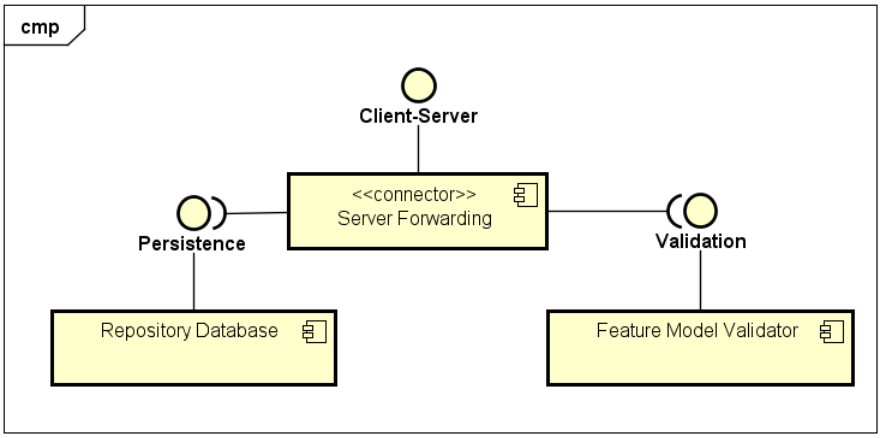

Figure 4.3: High-level Server Component Diagram

4.3.2 State-Machine View

Due to the complexity of the State-Machine, a comprehensive description of it will be provided in this section, in the form of natural language descriptions and UML Statechart diagrams. Different views are provided for each Feature Model notation, given that the operations available in them can differ greatly.

4.4 Client Component Specifications

The client side of the Atlas application is formed by four main components: Diagramming State-Machine; Renderer; Feature Model, and; GUI. The GUI subsystem can be further reduced to the sub-components: Model Visualizer; Diagramming Toolbar, and; System Interface. This section presents the specification of each of these components in natural language, as well as a freehand code-like representation of each component's interfaces, loosely based on the OMG IDL [10].

4.4.1 Diagramming State-Machine

The Diagramming State-Machine component serves to maintain the current state of the diagramming application in accordance to the interactions of the user with the GUI, primarily with the Diagramming Toolbar and Model Visualizer. The states held by this component are then used by the GUI to define which operations are permitted upon the model.

This component provides a single *Façade* type interface offering: a *get* operation for the current state; a *transition* operation for modifying the state, and; permission operations that inform which operations are available from a certain

```
State-Machine Facade{
    property state cState;
    property operation[] aOperations;

    out state currentState(){
        return cState;
        };
    inout state transition(operation t){
        execute t;
        return cState;
        };
    out operation[] availableOperations(){
        return aOperations;
        };
    out boolean isAvailable(operation t){
        return true if (t exists in aOperations)
                    else false;
        };
}
```

Listing 1: Representation of State-Machine Facade

state, or whether or not a certain operation is available. Listing 1 presents a code-like representation of this interface.

4.4.2 Renderer

The Renderer component serves as an encapsulation of the diagramming library functions used to render the visualization of the model. It operates based on a procedure call connector linked to the GUI, which activates the Renderer, and the *Façade* type interface provided by the Feature Model blackboard. This component holds no internal state, being responsible solely for providing operations and accessing the blackboard model. Listing 2 presents a code-like representation of the Render interface.

```
Render{
    out visualization render(){
        execute create visualization v;
        return v;
    }
}
```

Listing 2: Representation of Render interface

The interaction of the Render component with the Feature Model blackboard

is read-only, being responsible solely for creating a visualization from the model.

4.4.3 Feature Model

The Feature Model component serves as a blackboard model to the client application. It holds the entire structure of the model being constructed, as well as any data required for visualization rendering. It is modified in accordance to the user's interaction with the GUI, in accordance to permissions given by the Diagramming State-Machine, and serves as the basis for the actions of the Renderer.

```
Feature Model Facade{
    property model;

    in void alter(operation t, elements[] e){
        execute operation t(e);
    }
    out model view(){
        return model;
    }
}
```

Listing 3: Representation of Blackboard Facade

Listing 3 presents a code-like representation of the Blackboard Facade interface. It is important to note that this representation is highly abstract, assuming that all operations have a single related procedure with a well-specified signature. In practice, operation overloading is probable and proper care must be taken during implementation. It is also important to note that only the GUI has access to the *alter* operation, this being forbidden to the Renderer.

4.4.4 GUI

The GUI is a subsystem within the client application of Atlas, serving as an encapsulation for the inner GUI components. Given that no interfaces are provided by the GUI subsystem, the external components need not know whence requests originate from, interaction with the GUI subsystem which forwards the responses appropriately, based on the port/interface used for the request. The GUI subsystem is responsible for the visualization of the Feature Model, as well as handling the user interactions in what refers to both diagramming actions and system actions, as defined below.

Model Visualizer

The Model Visualizer sub-component is strictly responsible for presenting the visualization of the Feature Model, as provided by the Renderer, executing

tooltip and highlight type operations based on the current state of the State-Machine and cursor position, and acquiring cursor coordinates. The Model Visualizer is therefore not responsible for holding the model being presented, and is encapsulated away from GUI buttons that interact with the State-Machine and the server.

This component's interactions can be divided between the interfaces it consumes:

- The State-Machine is used by the Model Visualizer to identify the which state is currently active, and to execute operations that require cursor coordinate input. The information of active state is used to decide what tooltip and highlight operations must be executed depending on cursor position. The operations executed upon the State-Machine from the Model Visualizer will not forward the cursor coordinates, but rather will use them to modify the model in accordance to the state resulting from the operation. Unlike the Diagramming Toolbar, the Model Visualizer is NOT responsible for identifying whether or not an operation is legal prior to forwarding it to the State-Machine. Rather, it must be capable of treating exceptions, undoing recent operations, and presenting help messages to the user.
- The Renderer is used by the Model Visualizer when, upon executing an action, it identifies that the current state of the State-Machine requires a re-drawing of the visualization.
- The Blackboard is used by the Model Visualizer when, upon executing an action, it identifies that the current state of the State-Machine requires updating the model.

Diagramming Toolbar

The Diagramming Toolbar sub-component serves strictly to encapsulate operations executed upon the State-Machine which do not require cursor coordinate input. Generally speaking, it will operate sequentially, forwarding a (previously-identified as legal) operation to the State-Machine, and then reforming the presented Toolbar in accordance to what operations are legal after the operation is executed. The Diagramming Toolbar is responsible for identifying whether or not an operation is legal prior to forwarding it to the State-Machine.

System Interface

The System Interface sub-component is responsible for all communication between client and server. Therefore, all operations that depend upon the persistence repository or the model validators must be represented in the System Interface GUI. Despite being part of the GUI, the System Interface sub-component is independent from the Model Visualizer and Diagramming Toolbar

sub-components, operating strictly based on the Blackboard state and the operations provided by the server. The exception are the initialization operations (Load Model, New Model), which force change the State-Machine and force starts the Renderer.

Chapter 5

Atlas - Feature Modelling Tool

In this chapter, the design and implementation of Atlas are expanded on based on the technologies and notations previously explored in Chapters 3 and 4 and the requirements elicited in Chapter 2.

5.1 Prototyping

This section serves to define the prototyping stage of the Atlas research, which will be used to better define the capabilities of the technologies raised in Chapter 4 so as to aid in the selection of the ones being used in the implementation of the tool.

5.1.1 Diagramming Prototype

The first prototyping stage of Atlas will be the exploration of the capabilities of the diagramming libraries raised in Subsection 4.1.1. For this prototyping stage, certain diagram dispositions and diagramming functions are raised as critical points based on the research team's experience with previous implementation of diagramming tools, particularly feature modeling tools [7] [2].

Simple FODA Model

Chapter 6

Conclusion

Bibliography

[1] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[2] E. de M. Rodrigues, L. Passos, F. Teixeira, A. F. Zorzo, and R. Saad. On the Requirements and Design Decisions of an In-House Component-based SPL Automated Environment. In *26th International Conference on Software Eng. and Knowledge Eng.*, pages 483–488, Vancouver, CA, 2014.

[3] E. de M. Rodrigues, L. D. Viccari, A. F. Zorzo, and I. M. Gimenes. PLeTs-Test Automation using Software Product Lines and Model Based Testing. In *22nd International Conference on Software Eng. and Knowledge Eng.*, pages 483–488, San Francisco, USA, 2010.

[4] H. Jifeng, X. Li, and Z. Liu. *Component-Based Software Engineering*, pages 70–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[5] M. S. Laser, A. Domingues, and E. M. Rodrigues. Hephaestus - a software product line configurator. In *3rd Latin-American School on Software Engineering*, 2016.

[6] M. S. Laser, E. M. Rodrigues, A. R. P. Domingues, F. M. Oliveira, and A. F. Zorzo. Architectural Evolution of a Software Product Line: an experience report. In *27th SEKE*, pages 217–222, Pittsburgh, USA, 2015.

[7] M. S. Laser, E. M. Rodrigues, C. M. Martins, and F. Oliveira. Atlasspl - a web-based tool for feature modeling. In *2nd Latin-American School on Software Engineering*, pages 54–65, 2015.

[8] M. Mendonça, M. Branco, and D. D. Cowan. S.P.L.O.T.: software product lines online tools. In S. Arora and G. T. Leavens, editors, *OOPSLA Companion*, pages 761–762. ACM, 2009.

[9] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5, 2011.

[10] OMG. OMG Interface Definition Language (OMG IDL), Version 3.5. http://www.omg.org/spec/IDL35/3.5/, 2014.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[12] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.